Sean Welch - x23285508 - E-commerce System

Task 9 - Agile Methodology:

For the implementation of the E-commerce system that has been outlined in part one of this report, I will follow a Scrum methodology. Scrum can be defined as a management framework that teams can use to self organise and work towards a common goal [1]. This kind of project management structure is especially pertinent in software development areas like E-commerce as it allows for adaptable and incremental development cycles which make the teams more fluid when it comes to changing product requirements [2]. Indeed the fluid approach is often built upon the philosophy of empiricism, in that true knowledge comes from experience, and that experience is gained from an iterative approach to product development [2]. At its core the main objective of scrum is to deliver value in a collaborative way which thus allows the team surrounding the development of software to solve complex problems [3]. However, in our E-commerce system it is crucial to point out the various detailed factors that go into a scrum methodology. For example, scrum methodologies are often characterised by some archetypal principles and values.

First of all the principles involved in scrum include transparency, reflection and adaptation [1]. In this E-commerce system, it is pertinent that the team involved will offer transparency when working in the team by ensuring that all team members are aware of the overarching challenges and goals of their fellow scrum members [1]. Moreover, with regard to reflection, it is crucial that in this scrum based system that clear and predefined reflectiontion points are built into the framework so that the scrum leaders, product owner and development team review progress and allow the product manager to continually adjust and refine goals for the future [1]. As aforementioned, the final main principle of scrum, adaptation, is also crucial to this e-commerce system. Indeed, once reflection points have been reached, it is crucial that the team adapt to changing customer requirements by prioritising tasks and ensuring optimal delivery of those tasks within a hierarchy [1].

Following on, there are also important values that must be adopted by a team in a scrum methodology in order to ensure the governing principles are followed. In this e-commerce system, it is crucial that the team is committed to the project and the deliverance of the time based tasks making up that project [3]. It is also pertinent that each individual team member shows the courage to ask difficult questions to ensure the objectives of the team are understood by all [3]. Focus is another important value that promotes the prioritisation of tasks laid out in a hierarchy of the backlog [3]. Lastly, the two final values that must be exemplified by all scrum members in this e-commerce system and are openness and respect; these values will ensure that team members are not rigidly stuck in their preexisting ideas and are open to change while also permitted open dialog and collaboration between team members [3].

As outlined prior, scrum is a more modern and adaptive approach to software development lifecycles; however, there are other more rigid approaches like for example the Waterfall approach [4]. Scrum and waterfall are often portrayed as being polar opposite approaches to SDLC due to the fact that waterfall is often based on a methodology whereby customer requirements are constant and there is a very linear approach to product development with a clear start and endpoint [4].

This is of course very different from scrum in the context of our e-commerce system as it is designed to be adaptable and resilient to change. To differentiate further, in a waterfall methodology, project phases are clearly laid out from the beginning and follow a rigid, linear pattern; these phases are often characterised by requirement gathering, followed by specification and design of the product [5]. Next is

implementation and testing where code is created to the aforementioned requirement specifications [5]. Finally there is the maintenance stage where the code and product have been delivered and issues are fixed as they arise in production [5].
Given these differences outlined above, it is clear to see that the e-commerce system in this report would benefit greatly from the iterative and adaptable approach of a scrum methodology. A waterfall approach is a sub-optimal way to organise a management framework for our product given that in a dynamic and competitive software landscape, an e-commerce system will constantly have to evolve in order to continuously integrate and develop new features for a growing customer base.

Task 10 - Methodology Artefacts:

Artefacts are a crucial tool in managing and implementing an agile methodology in an effective manner. Within the context of our e-commerce system, they will provide an actionable way to visualise, manifest and iterate through our scrum based system. Indeed there are three main artefacts that will prove critical to the successful implementation of our agile and scrum methodology: user stories, product backlogs and burndown charts.
Firstly, user stories are the building blocks of any scrum implementation; in essence they are short an informal message conveyed from the persona of a user of the software that details how a certain software feature is to provide value to the user [28]. Indeed, oftentimes user stories are used to bridge the gap between the technical requirements of a project and the needs of the user in the real world [29].
In scrum, user stories are added to sprints and burned down over the duration of said sprint; this helps keep the development teams focus on the user, which in turn provides many benefits [28]. From a developers perspective, such benefits include: enhances collaboration and drive to propose creative solutions to complex problems [28]. From a project managers point of view, user stories allow scrum teams to give better estimation of how long a sprint may require, and thus allow for better forecasting for future requirements of a project [28].
Finally, with regard to user stories, it is often pertinent to focus them on the INVEST principles; that is user stories should be independent of each other; they should be negotiable and adaptable to changing requirements; they should be valuable to the user; they should be estimable in terms of the time requirement; they should also be small in their scope to avoid over complication; and lastly, they should be testable [30]. Below, I have attached some examples of user stories from the systems Azure Devops organisation.

USER STORY 13*

13 User Management - Registration

Sean Welch | 0 comments | Add tag

State	New	Area	E-commerce System
Reason	New	Iteration	E-commerce System\Iteration 1

Description

As a new user, I wan't to be easily able to register with my details and not have to worry about losing any information on my account

Acceptance Criteria

We need to implement a smooth and reliable process for our user to register their details with us.
Some options include: Firebase authentication. This one will be reliable, and the quickest to implement.
However, it could lock us into using other firebase products in the future.
Some other more modern and secure approached include implement the Oauth2 protocol to authenticate users with other providers and then use their client id to validate the session.
If neither of these options work, we could always roll our own auth by hashing user passwords with a salt code on the sever, and then adding a Json Web token to their browser as a HTTP only cookie to authenticate requests to the backend.
However, this will probable yield the least secure options and most cumbersome for users causing them to have to re-authenticate more often

USER STORY 16*

16 Shopping Cart - Checkout Experience

Unassigned | 0 comments | Add tag

State	New	Area	E-commerce System
Reason	New	Iteration	E-commerce System\Iteration 1

Description

As a user, I want to have a smooth cart experience. When I add products to my cart, I want them to stay there for an extended period of time in case I come back later to purchase

Acceptance Criteria

In order to implement such a feature were going to need to use some sort of client side state to hold information about the user's cart and then have this state persist across sessions.
For the client side state, there are many options available to us, especially if we're going to implement some sort of Single Page Application Framework like React or Angular.
For example, if using react we can use many librays such as nanostores, react context, or my personal preference Zustand.
This will allow us to access and modify the state of the shopping cart from anywhere we wish to implement the logic on the client side app.
However, in order to get this to persist across user sessions, we probably have to integrate zustand with some sort of web storage API such as local storage.
For example, as soon as client state is stored in the cart with zustand, we can then also pass this state as json string to localstorage. When we need to access that state on a page, we first check for the item in localstorage, as this will be the most up to date version. The local storage item will persist across page refreshes and thus can be used to implement this user story with easy

USER STORY 17*

17 Payment Options - Checkout Experience

Sean Welch | 0 comments | Add tag

State	New	Area	E-commerce System
Reason	New	Iteration	E-commerce System\Iteration 1

Description

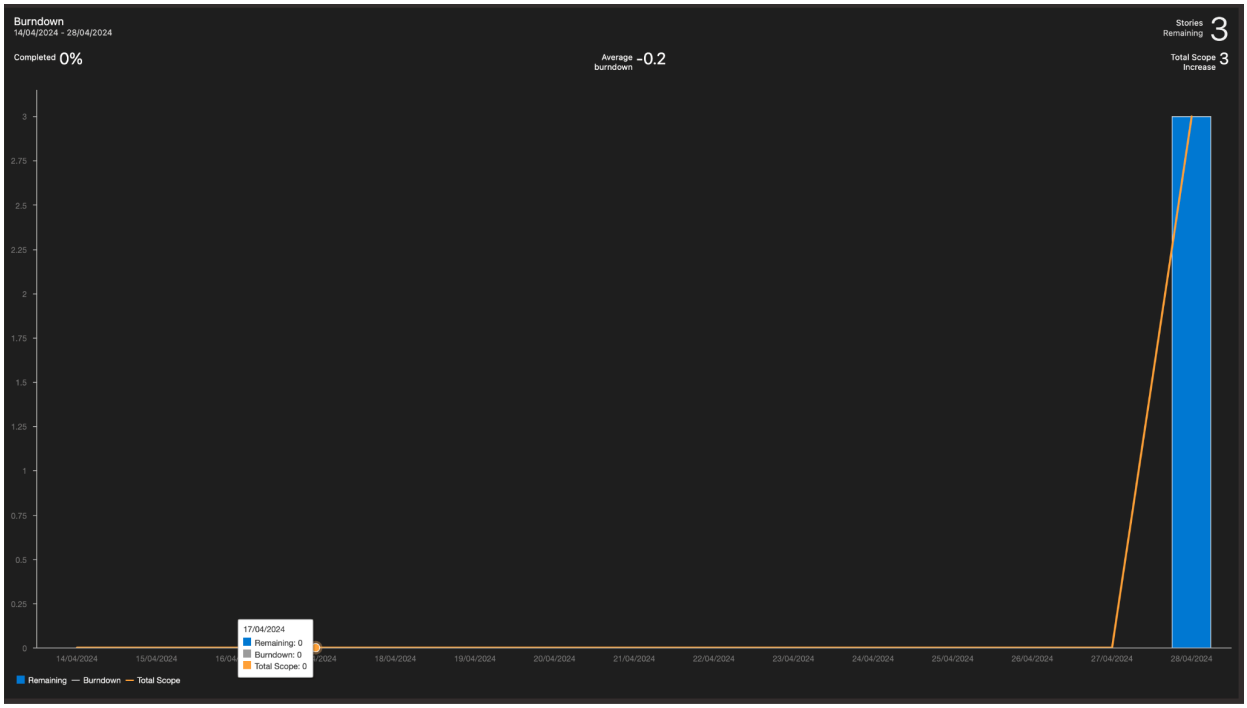
As a user, I want a smooth checkout experience when it comes to payments. I'd like the option to choose from various providers such as Apple/Google Pay, Card Payments and perhaps even Bank Transfers. I want the checkout Experience to be reliable and not take too long to confirm, so I don't get anxious about my payment being declined or fraudulent.

Acceptance Criteria

In order to implement this feature for our users, there's only one choice for use really; we're going to use Stripe's library of payment APIs to do this.
Stripe's third party payment solutions are tried and tested and will help us conform to the wants of our users. The Checkout experience is sleek and modern; their API is robust and reliable meaning everything will load and process in an effective manner provided our User's have a decent internet connection and are working from a modern browser.
I've attatched a link to some really helpful documentations for implementing Stripe's tailored checkout experience: https://docs.stripe.com/checkout/embedded/quickstart. They even give us a quickstart guide of how to implement their API into our server in Java

```
public class Server {

  public static void main(String[] args) {
    port(4242);

    // This is a public sample test API key.
    // Don’t submit any personally identifiable information in requests made with this key.
    // Sign in to see your own test API key embedded in code samples.
    Stripe.apiKey = "sk_test_zzPhAh8sZkhmI4JDtzTNnhGl";

    staticFiles.externalLocation(
        Paths.get("public").toAbsolutePath().toString());

    Gson gson = new Gson();

    post("/create-checkout-session", (request, response) -> {
        String YOUR_DOMAIN = "http://localhost:4242";
        SessionCreateParams params =
          SessionCreateParams.builder()
            .setUiMode(SessionCreateParams.UiMode.EMBEDDED)
            .setMode(SessionCreateParams.Mode.PAYMENT)
            .setReturnUrl(YOUR_DOMAIN + "/return.html?session_id={CHECKOUT_SESSION_ID}")
            .addLineItem(
              SessionCreateParams.LineItem.builder()
                .setQuantity(1L)
                // Provide the exact Price ID (for example, pr_1234) of the product you want to sell
                .setPrice("{{PRICE_ID}}")
                .build())
            .build();

      Session session = Session.create(params);

      Map<String, String> map = new HashMap();
      map.put("clientSecret", session.getRawJsonObject().getAsJsonPrimitive("client_secret").getAsString());


      return map;
    }, gson::toJson);

    get("/session-status", (request, response) -> {
      Session session = Session.retrieve(request.queryParams("session_id"));

      Map<String, String> map = new HashMap();
      map.put("status", session.getRawJsonObject().getAsJsonPrimitive("status").getAsString());
      map.put("customer_email", session.getRawJsonObject().getAsJsonObject("customer_details").getAsJsonPrimitive("email").getAsString());

      return map;
    }, gson::toJson);
  }
}
```

Moving on, another important artefact for the process of scrum is that of a product backlog. This artefact can be described as a list of prioritised work that is created by the product manager for the implementation by the engineering team and is usually derived from the product roadmap and its requirements [31]. Crucially, it must be noted that although the product manager is the one to comprise the product backlog and is responsible for its proper organisation, it is the developer team that works through this backlog at their own capacity throughout scrum iterations [31].
Indeed, the product backlog artefact is another crucial artefact for the implementation of a scrum based agile methodology, and a well organised and prioritised product backlog can allow for faster and more effective iteration in sprint cycles; in turn this allows for more frequent feature releases and easier forecasting into future requirements [31]. However, it is not just the scrum team who benefits from an effective product backlog. Crucially, the product backlog can help set the expectation of other stakeholders involved in the organisation; in essence, it makes engineering time a fixed asset by showing other stakeholders the work involved in the implementation of new features and the sprint cycles [31].
Finally, it is also important to note the importance of Burndown Charts in the implementation of a scrum based system. Burndown Charts provide a way to measure and graphically represent the amount of work that has been completed in a sprint cycle, and also represent the total work remaining [32]. Indeed, Burndown Charts can be used to predict the team's likelihood of completing their work in the allotted sprint time; this is of course a valuable insight not only for the developer team - as it makes them aware of the aforementioned scope creep issue - but also for other stakeholders as it can show the actionable results throughout a sprint cycle and allow for readjustments if necessary [32]. There are various steps involved in making a Burndown Chart, such as: setting time estimations, tracking daily progress, computing actual effort and of course obtaining the final dataset used to plot the burndown chart [32]. All of which can be seen in the example used below form our Azure Devops organisation.



Task 11 - Potential Risks:

Indeed, Agile (and more specifically, Scrum) is not a foolproof strategy, and in a real world system that has to adapt to both a changing economic and consumer landscape, many challenges can arise to the proper functioning of the system [12]. So too can there be challenges in the effectiveness and quality of communication management in such E-commerce systems.

Firstly, there exists pertinent legal and compliance that is directly applicable to a business or system operating in an e-commerce landscape. For example, payments are involved in every business transaction and thus, the payment card industry has a hand to play in some aspect of any system, especially e-commerce [9]. For example, in the regard of our system, it is crucial that transactions are handled carefully to comply with PCI data security regulations and also that customer's card information is stored securely in such a manner that is compliant with modern cybersecurity protocols and regulations [13]. One way to mitigate this risk, as our system will do, is to integrate a robust, secure and battle tested payment provider like Stripe or Paypal. Again, in our case, the choice will be Stripe to their modern API solutions, as well as their intuitive and smart user interface that will allow our system to easily comply with the aforementioned regulations [13].

Moving on, with regard to agile and scrum in a more generalist manner, there are some pressing issues that must be addressed if our agile methodology is to be implemented in an effective manner; for example, one pressing issue that all agile systems have is scope creep and team collaboration [10]. Scope creep describes a phenomena in an agile based system whereby due to the iterative and flexible nature of a scrum, the project may get out of hand without clearly defined scope and direct feedback [11].

To explain further, as mentioned the goal of agile is to quickly adapt to changing customer needs; however, when a team is constantly adding new features and requirements to a project, it is very easy for tasks to get left behind and for these the latter affect the smooth running of the project [10]. This is why an effective communication strategy is crucial to mitigate such a risk. Indeed, one way to implement such a protocol that can be seen across agile systems is through the use of sprints and daily standups [11]. Through these mechanisms, the team can more effectively communicate by having a common goal to work towards for the sprint period, and then through daily standups, communicate any issues they are having moving towards that goal.

Indeed another major risk to scrum and agile development is that of technical debt [11]. Technical debt defines a situation whereby, as a project grows in scale, the codebase becomes harder to manage and adapt to changing requirement and new features; this is usually caused by poor architectural design in the beginning phases of a systems development, as well as poor coding practices and design through the various development cycles [10]. As aforementioned, in order to properly mitigate the problem of technical debt, there needs to be proper quality management practices put in place; for example these often include proper architectural design at the beginning of the project, as well as a continuous integration and continuous development (CI/CD) pipeline throughout the sprint cycles [11].

To explain further, in order to mitigate problems with architectural design, a modular and microservices based system can help. This means that each feature of the code is implemented in a distinct manner such that interconnected pieces will not cause an issue; whereas a microservices architecture involves splitting up cloud architecture into small components to reduce complexity and cost [11]. Moreover, in order to introduce quality management throughout the systems development, automated testing can be built into the CI/CD pipelines to ensure code and ideas being introduced into the system adhere to certain quality standards [14].

Task 12 - Class Implementation:

For the class implementation of the Checkout use case described in part one of the project report, I decided to implement a Terminal/Text User interface developed in the console [18]. This small application serves as a prototype for the e-commerce checkout experience and has been implemented using the Model View Presenter pattern [15].
In the application the majority of java classes serve as the model layer, storing data in memory and exposing various simple methods. However, the EcommerceService layer acts as the presenter layer of the application, which controls the various model layers and applies the necessary business logic for the EcommerceTUI or the view layer to take over and present the data to the end user [15].
For the most part, references for this code were taken from W3Schools to refresh the basics of Java based syntax and class implementation [16], [17]. However, for more complex business logic, I took reference to some of my own personal projects, which are available on my Github for transparency: https://github.com/sean-david-welch [19]. Some repositories may be private but could be made available upon request.

// Status Enum

```
public enum Status {
   FULFILLED, SHIPPED, CANCELLED;
}
```

// Shipping class

```
public class Shipping {
   String id;
   String address;
   String deliveryCost;
   Status status;

   public Shipping(String id, String address, String deliveryCost, Status status) {
       this.id = id;
       this.address = address;
       this.deliveryCost = deliveryCost;
       this.status = status;
   }

   public String getId() {
       return id;
   }

   public void setId(String id) {
       this.id = id;
   }

   public String getAddress() {
       return address;
   }

   public void setAddress(String address) {
       this.address = address;
   }

   public String getDeliveryCost() {
       return deliveryCost;
   }

   public void setDeliveryCost(String deliveryCost) {
```

```
        this.deliveryCost = deliveryCost;
    }

    public Status getStatus() {
        return status;
    }

    public void setStatus(Status status) {
        this.status = status;
    }

    public void updateStatus(Status newStatus) {
        this.status = newStatus;
    }
}
```

// Order Class

```
public class Order {
    String id;
    Cart cart;
    Shipping shipping;
    Status status;

    public Order(String id, Cart cart, Shipping shipping, Status status) {
        this.id = id;
        this.cart = cart;
        this.shipping = shipping;
        this.status = status;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Cart getCart() {
        return cart;
    }

    public void setCart(Cart cart) {
        this.cart = cart;
    }

    public Shipping getShipping() {
        return shipping;
    }

    public void setShipping(Shipping shipping) {
        this.shipping = shipping;
    }

    public Status getStatus() {
        return status;
    }

    public void setStatus(Status status) {
        this.status = status;
    }

    public void updateStatus(Status newStatus) {
        this.status = newStatus;
        System.out.println("Order ID " + id + " status updated to: " + status);
```

```
    }

    public void cancelOrder() {
        if (status == Status.FULFILLED) {
            System.out.println("Order ID " + id + " cannot be cancelled as it is already fulfilled.");
        } else {
            this.status = Status.CANCELLED;
            System.out.println("Order ID " + id + " has been cancelled.");
        }
    }
}
```

// Merchant Class

```
public class Merchant {
    String id;
    String name;

    public Merchant(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void manageOrder(Order order) {
        if (order.getStatus() == Status.CANCELLED) {
            System.out.println("Order ID " + order.getId() + " has been cancelled. No further actions
are possible.");
        } else {
            System.out.println("Managing order ID " + order.getId() + ".");
        }
    }

    public void shipOrder(Order order) {
        if (order.getStatus() == Status.FULFILLED) {
            order.updateStatus(Status.SHIPPED);
            System.out.println("Order ID " + order.getId() + " has been shipped.");
        } else {
            System.out.println("Order ID " + order.getId() + " is not ready for shipping.");
        }
    }
}
```

// Product Class

```
import java.util.UUID;

public class Product {
    String id;
    String name;
    String description;
```

```
    float price;

    public Product(String id, String name, String description, float price) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public void buyNow(Cart cart, int quantity) {
        CartItem cartItem = new CartItem(UUID.randomUUID().toString(), this, quantity);
        cart.addItem(cartItem);
    }
}
```

// Customer Class

```
import java.util.HashMap;

public class Customer {
    String id;
    String name;
    String email;
    Shipping shipping;
    Cart shoppingCart;

    public Customer(String id, String name, String email, Shipping shipping, Cart shoppingCart) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.shipping = shipping;
        this.shoppingCart = shoppingCart;
    }
```

```
public Customer() {}

private final HashMap<String, String> registeredCustomers = new HashMap<>();
private boolean isLoggedIn = false;

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public Shipping getShipping() {
    return shipping;
}

public void setShipping(Shipping shipping) {
    this.shipping = shipping;
}

public Cart getShoppingCart() {
    return shoppingCart;
}

public void setShoppingCart(Cart shoppingCart) {
    this.shoppingCart = shoppingCart;
}

public void register(String email, String password) {
    if (registeredCustomers.get(email) == null) {
        registeredCustomers.put(email, password);
        System.out.println("Registration successful for: " + email);
    } else {
        System.out.println("Email already registered!");
    }
}

public void login(String email, String password) {
    String storedPassword = registeredCustomers.get(email);
    if (storedPassword != null && storedPassword.equals(password)) {
        isLoggedIn = true;
        System.out.println("Logged in successfully as: " + email);
    } else {
        System.out.println("Login failed. Incorrect email or password.");
    }
}

public void placeOrder(Order order) {
    if (isLoggedIn) {
```

```
            order.updateStatus(Status.FULFILLED);
            System.out.println("Order placed successfully.");
        } else {
            System.out.println("You need to login first.");
        }
    }

    public void initiateCheckout() {
        if (isLoggedIn) {
            if (!shoppingCart.items.isEmpty()) {
                System.out.println("Initiating checkout...");
                System.out.println("Checkout complete. Total:€ " + shoppingCart.getTotal());
            } else {
                System.out.println("Your shopping cart is empty.");
            }
        } else {
            System.out.println("You need to login first.");
        }
    }
}
```

// Cart Item Class

```
public class CartItem {
    String id;
    Product product;
    int quantity;

    public CartItem(String id, Product product, int quantity) {
        this.id = id;
        this.product = product;
        this.quantity = quantity;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void updateQuantity(int input) {
        int newQuantity = this.quantity + input;
        if (newQuantity >= 0) {
            this.quantity = newQuantity;
        } else {
            System.out.println("Cannot reduce quantity below zero.");
            this.quantity = 0;
```

```
        }
    }
}
```

// Cart Class

```
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class Cart {
    String id;
    List<CartItem> items;
    int total;

    public Cart() {
        this.id = UUID.randomUUID().toString();
        this.items = new ArrayList<>();
        this.total = 0;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public List<CartItem> getItems() {
        return items;
    }

    public void setItems(List<CartItem> items) {
        this.items = items;
    }

    public int getTotal() {
        this.total = 0;
        for (CartItem item : this.items) {
            this.total += (int) (item.getProduct().getPrice() * item.getQuantity());
        }
        return total;
    }

    public void setTotal(int total) {
        this.total = total;
    }

    public void addItem(CartItem cartItem) {
        this.items.add(cartItem);
        this.total += (int) (cartItem.getProduct().getPrice() * cartItem.getQuantity());
    }

    public void removeItem(String itemId) {
        items.removeIf(item -> item.getId().equals(itemId));
    }

    public void clearCart() {
        this.items.clear();
        this.total = 0;
    }
}
```

// Ecommerce Service layer to interact with classes (Presenter)

```
import java.util.*;

public class EcommerceService {
   private final Cart cart = new Cart();
   private final Customer customer = new Customer();
   private final List<Product> products = new ArrayList<>();

   public EcommerceService() {
       products.add(new Product(UUID.randomUUID().toString(), "Whey Protein", "Premium whey protein for
muscle growth, 2 lbs", 49.99f));
       products.add(new Product(UUID.randomUUID().toString(), "Multivitamins", "Complete daily
multivitamins for overall health", 29.99f));
       products.add(new Product(UUID.randomUUID().toString(), "Omega-3 Capsules", "High-strength fish
oil omega-3 capsules", 19.99f));
   }

   public void registerUser(String email, String password) {
       customer.register(email, password);
   }
   public void loginUser(String email, String password) {
       customer.login(email, password);
   }

   public List<CartItem> viewCart() {
       List<CartItem> emptyList = Collections.emptyList();

       if (cart.getItems().isEmpty()) {
           System.out.println("Your cart is empty.");
           return emptyList;
       }

       return cart.getItems();
   }

   public int getCartSubtotal() {
       return cart.getTotal();
   }

   public List<Product> viewProducts() {
       return products;
   }

   public void addProductToCart (String productName, int quantity) {
       for (Product product: products) {
           if (Objects.equals(productName, product.name)) {
               String cartItemId = UUID.randomUUID().toString();
               CartItem cartItem = new CartItem(cartItemId, product, quantity);

               cart.addItem(cartItem);
               System.out.println("Added " + quantity + " of " + productName + " to the cart.");
               customer.setShoppingCart(cart);
               return;
           }
       }
       System.out.println("Product '" + productName + "' not found.");
   }

   public void initiateCheckout() {
       customer.initiateCheckout();
   }
}
```

// EcommerceTUI layer - Terminal User Interface (View)

```
import java.util.List;
import java.util.Scanner;

public class EcommerceTUI {
   private final Scanner scanner;
   private final EcommerceService ecommerceService;

   public EcommerceTUI(Scanner scanner, EcommerceService ecommerceService) {
       this.scanner = scanner;
       this.ecommerceService = ecommerceService;
   }

   public void registerUser() {
       System.out.println("Please enter your email:");
       String email = scanner.nextLine();
       System.out.println("Please enter your password:");
       String password = scanner.nextLine();

       ecommerceService.registerUser(email, password);
   }

   public void loginUser() {
       System.out.println("Please enter your email:");
       String email = scanner.nextLine();
       System.out.println("Please enter your password:");
       String password = scanner.nextLine();

       ecommerceService.loginUser(email, password);
   }

   public void viewCart() {
       List<CartItem> items = ecommerceService.viewCart();


       if (!items.isEmpty()) {

           System.out.println("Items in your cart:");
           System.out.printf("%-30s %-10s %-10s%n", "Product Name", "Quantity", "Price");

           for (CartItem item : items) {
               Product product = item.getProduct();
               System.out.printf("%-30s %-10d €%.2f%n",
                       product.getName(),
                       item.getQuantity(),
                       product.getPrice() * item.getQuantity());
           }

           int subtotal = ecommerceService.getCartSubtotal();
           System.out.println("Total: €" + subtotal);
       }
   }

   public void viewProducts() {
       List<Product> products = ecommerceService.viewProducts();

       if (products.isEmpty()) {
           System.out.println("No products available");
           return;
       }

       System.out.println("Available Products:");
       int index = 1;
       for (Product product : products) {
           System.out.printf("%d. Name: %-30s Description: %-50s Price: €%.2f%n",
```

```
                    index++,
                    product.getName(),
                    product.getDescription(),
                    product.getPrice());
        }
        System.out.println("--------------------");
    }


    public void addProductToCart () {
        viewProducts();

        System.out.println("Enter the name of the product you want to add to the cart:");
        String productName = scanner.nextLine().trim();
        System.out.println("--------------------");


        System.out.println("Enter the quantity:");
        System.out.println("--------------------");
        int quantity;
        try {
            quantity = Integer.parseInt(scanner.nextLine().trim());
            if (quantity <= 0) {
                System.out.println("Quantity must be greater than zero.");
                return;
            }
        } catch (NumberFormatException e) {
            System.out.println("Invalid input for quantity. Please enter a number.");
            return;
        }

        ecommerceService.addProductToCart(productName, quantity);
        System.out.println("--------------------");
    }

    public void initiateCheckout() {
        ecommerceService.initiateCheckout();
    }

    private void viewMainOptions() {
        String[] mainOptions = new String[] {
                "Register User", "Login User",
                "View Cart", "View Products",
                "Add Product To Cart", "Initiate Checkout"
        };

        System.out.println("--------------------");
        System.out.println("Please choose from the options... Enter 0 to view them again.... Press 7 to
quit");
        System.out.println("--------------------");
        for (int i = 0; i < mainOptions.length; i++) {
            System.out.printf("%s. %s%n", (i + 1), mainOptions[i]);
        }
        System.out.println("--------------------");
    }

    public void ecommerceController() {
        this.viewMainOptions();

        while (true) {
            String inputLine = scanner.nextLine().trim();
            int input = -1;
            try {
                input = Integer.parseInt(inputLine);
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
```

```
                continue;
            }
            System.out.println("--------------------");

            if (input == 0) {
                this.viewMainOptions();
                continue;
            } else if (input == 7) {
                break;
            }

            switch (input) {
                case 1:
                    this.registerUser();
                    break;
                case 2:
                    this.loginUser();
                    break;
                case 3:
                    this.viewCart();
                    break;
                case 4:
                    this.viewProducts();
                    break;
                case 5:
                    this.addProductToCart();
                    break;
                case 6:
                    this.initiateCheckout();
                    break;
                default:
                    System.out.println("Invalid option. Try Again or enter 0 to see the options! Press 7
to quit!");
                    System.out.println("--------------------");
            }
        }
    }

}
```

// Main Class Entry Point

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class EcommerceApp {
    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            EcommerceService ecommerceService = new EcommerceService();
            EcommerceTUI ecommerceTUI = new EcommerceTUI(scanner, ecommerceService);

            while (true) {
                try {
                    ecommerceTUI.ecommerceController();

                    System.out.println("Enter 'quit'/'q' to exit or press Enter to continue.");
                    String input = scanner.nextLine();

                    if ("quit".equalsIgnoreCase(input) || "q".equalsIgnoreCase(input)) {
                        break;
                    }
                } catch (InputMismatchException err) {
                    System.out.println("Input is invalid, try again. Error: " + err.getMessage());
                    scanner.nextLine();
                } catch (Exception err) {
                    System.out.println("An error occurred, try again: " + err.getMessage());
```

```
                }
            }
        } catch (Exception err) {
            System.out.println("A critical error occurred, exiting the application: " +
err.getMessage());
        }
    }
}
```

Makefile for easy compilation and running in the terminal:

```
build:
    @javac -d target src/*.java

run:
    @java -cp target EcommerceApp

deploy: build run

zip:
    zip -r seanwelch.zip . -x references.txt -x ".git/*" -x ".git" -x .gitignore
```

Task 13 - Test Scenarios:

The importance of testing in the software development life cannot be understated. Indeed, when dealing with agile software development practices this holds even more true; in order to keep the project on track and not run into the issues previously outlined (scope creep and technical debt) it is crucial that a robust and resilient testing process be carried out [25]. For the purposes of this e-commerce system and application, the main type of tests that will be carried out are unit testing and integration testing. Through the combination of these two testing methodologies, it will make it far easier to identify issues and bugs early on, allow for smoother refactoring in the future, faster pivoting to new customer requirements and thus the effortless integration of new features [20].

Our first testing methodology is unit testing, which in essence, involves breaking down the application into the smallest possible features or implementations and testing these minor parts to ensure they work correctly [22]. Unit testing is ubiquitous throughout software development and all major languages often have testing built into their runtime like in the example of Golang; or they have external frameworks built by the community to make testing easier to implement and more robust, like that of Junit in Java, Pytest with Python or Jest in Javascript [23]. By integrating unit tests in our system's codebase, it will make it far easier to detect bugs early on and thus instil a culture of quality code in our organisation [22].

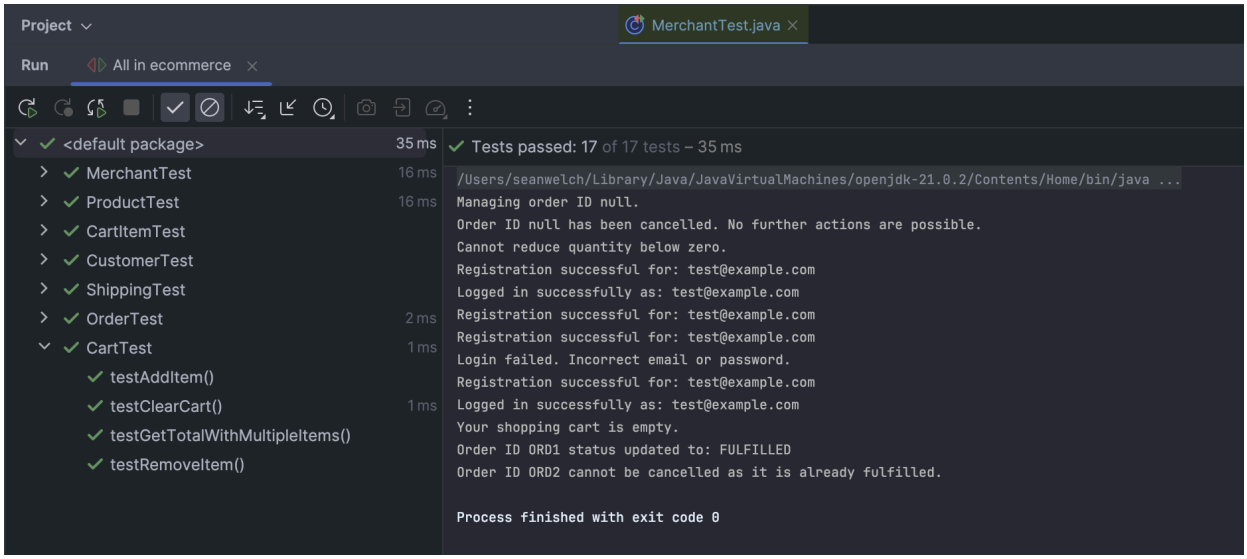
For the most part, there are important best practices that should be followed when implementing unit tests in our e-commerce system. For example, test should be appropriately named to ensure they ease of association with the relevant code; testing should be made as simple as possible to make sure complications are not made in the process; and finally, a metric should be asserted as to how much of the codebase should be under testing - the most common case being 80% coverage [24].

Moving on from unit testing, as our system grows in scale and complexity, it is critical that some form of integration testing be implemented in order to ensure that the system has test coverage as a whole. Integration testing involves writing tests that validate how different software components interact as a whole [20]. This often involves separating your codebase into different modules and testing how such modules interact within a predefined dataflow of your application, or how different components within

such modules interact with each other [20]. To illustrate this in a practical sense, we could look at our aforementioned checkout use case. To perform an integration test within this framework may involve testing the flow and transformation of data amongst all the various classes outlined previously. Moreover, in a real world scenario, if there was a call to an external payment provider, the integration test may involve mocking this api call to ensure the correct data is returned to our system in order to facilitate the whole application flow [23].

Oftentimes, integration testing will culminate in what is known as end-to-end testing. This is often the case when there is a client-facing application that is separate from the backend server layer of the system; To be precise, it involves verifying the function of your application from start to finish in a real world scenario [27]. In essence, this means writing test cases based on the potential of user interaction, and how this interaction will affect the client application, all the way through to the backend server, as well as the database [27].

To conclude, it is important that these kinds of tests are easy to implement within our system. For example, if tests are hard to write and perform, there is no point in implementing them as it will only cut down the velocity of the development cycle [26]. This is where an automated continuous integration and development pipeline comes into play that can build the test environment and run tests whenever a change is made to the codebase [26]. There are various tools available to developers in order to do this, and are often integrated within a Git based workflow to ensure that everytime code is committed to a repository, that the build environment and associated tests are run, either resulting in a base or a fail [22]. Such tools often used by modern systems, include Jenkins Pipelines, Gitlab runners, Github actions as well as Azure Devops. To elucidate examples of how unit testing can be implemented, I have included various test cases built for the classes in the previous section below. After running such test, these are the results I received:



// Shipping Class Test

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

import org.example.Shipping;
import org.example.Status;
import org.junit.jupiter.api.Test;

public class ShippingTest {

   @Test
```

```
    public void testUpdateStatus() {
        Shipping shipping = new Shipping("SHIP1", "123 Main St", "$5.00", Status.SHIPPED);
        Status newStatus = Status.FULFILLED;

        shipping.updateStatus(newStatus);

        assertEquals(newStatus, shipping.getStatus(), "Shipping status should be updated");
    }
}
```

// Product Test

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

import org.example.Cart;
import org.example.CartItem;
import org.example.Product;
import org.junit.jupiter.api.Test;

public class ProductTest {

    @Test
    public void testBuyNow() {
        Product product = new Product("P1", "Product Name", "Description", 10.00f);
        Cart cart = new Cart();

        product.buyNow(cart, 2);

        assertEquals(1, cart.getItems().size(), "One item should be added to the cart");
        CartItem cartItem = cart.getItems().getFirst();

        assertEquals(product, cartItem.getProduct(), "Product should match");
        assertEquals(2, cartItem.getQuantity(), "Quantity should be 2");
    }
}
```

// Order Test

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

import org.example.Cart;
import org.example.Order;
import org.example.Shipping;
import org.example.Status;
import org.junit.jupiter.api.Test;

public class OrderTest {

    @Test
    public void testUpdateStatus() {
        Order order = new Order("ORD1", new Cart(), new Shipping(), Status.SHIPPED);
        Status newStatus = Status.FULFILLED;

        order.updateStatus(newStatus);

        assertEquals(newStatus, order.getStatus(), "Order status should be updated");
    }

    @Test
    public void testCancelOrderFulfilled() {
        Order order = new Order("ORD2", new Cart(), new Shipping(), Status.FULFILLED);

        order.cancelOrder();

        assertEquals(Status.FULFILLED, order.getStatus(), "Order status should remain FULFILLED");
    }
```

```
}
```

// Merchant Test

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

import org.example.Merchant;
import org.example.Order;
import org.example.Status;
import org.junit.jupiter.api.Test;

public class MerchantTest {

   @Test
   public void testManageOrderCancelled() {
       Merchant merchant = new Merchant("1", "Merchant 1");
       Order cancelledOrder = new Order();
       cancelledOrder.setStatus(Status.CANCELLED);

       merchant.manageOrder(cancelledOrder);

       assertEquals(Status.CANCELLED, cancelledOrder.getStatus(), "Order status should be CANCELLED");
   }

   @Test
   public void testManageOrderActive() {
       Merchant merchant = new Merchant("M1", "Merchant 1");
       Order activeOrder = new Order();
       activeOrder.setStatus(Status.SHIPPED);

       merchant.manageOrder(activeOrder);

       assertEquals(Status.SHIPPED, activeOrder.getStatus(), "Order status should be SHIPPED");
   }

   @Test
   public void testShipOrderFulfilled() {
       Merchant merchant = new Merchant("M1", "Merchant 1");
       Order fulfilledOrder = new Order();

       merchant.shipOrder(fulfilledOrder);

       assertEquals(Status.FULFILLED, fulfilledOrder.getStatus(), "Order status should be FULFILLED");
   }
}
```

// Customer Test

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

import org.example.Cart;
import org.example.Customer;
import org.junit.jupiter.api.Test;

public class CustomerTest {

   @Test
   public void testCustomerRegistration() {
       Customer customer = new Customer();
       String email = "test@example.com";
       String password = "password123";

       customer.register(email, password);

       assertTrue(customer.registeredCustomers.containsKey(email), "Email should be registered");
       assertEquals(password, customer.registeredCustomers.get(email), "Password should match");
```

```
    }

    @Test
    public void testCustomerLoginSuccess() {
        Customer customer = new Customer();
        String email = "test@example.com";
        String password = "password123";

        customer.register(email, password);
        customer.login(email, password);

        assertTrue(customer.isLoggedIn, "Customer should be logged in");
    }

    @Test
    public void testCustomerLoginFailIncorrectPassword() {
        Customer customer = new Customer();
        String email = "test@example.com";
        String password = "password123";
        String wrongPassword = "wrongpassword";

        customer.register(email, password);
        customer.login(email, wrongPassword);

        assertFalse(customer.isLoggedIn, "Customer should not be logged in");
    }


    @Test
    public void testInitiateCheckoutLoggedInEmptyCart() {
        Customer customer = new Customer();
        String email = "test@example.com";
        String password = "password123";
        Cart emptyCart = new Cart();

        customer.register(email, password);
        customer.login(email, password);
        customer.setShoppingCart(emptyCart);
        customer.initiateCheckout();

        assertEquals(0, emptyCart.getItems().size(), "Shopping cart should be empty");
    }
}
```

// CartItem test

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

import org.example.CartItem;
import org.example.Product;
import org.junit.jupiter.api.Test;

public class CartItemTest {

    @Test
    public void testUpdateQuantityPositive() {
        Product product = new Product("1", "Product", "Description", 10.00f);
        CartItem item = new CartItem("1", product, 2);

        item.updateQuantity(3);
        assertEquals(5, item.getQuantity(), "Quantity should be updated to 5");
    }

    @Test
    public void testUpdateQuantityZero() {
        Product product = new Product("1", "Product", "Description", 10.00f);
```

```
        CartItem item = new CartItem("1", product, 2);

        item.updateQuantity(0);
        assertEquals(2, item.getQuantity(), "Quantity should remain 2");
    }

    @Test
    public void testUpdateQuantityNegative() {
        Product product = new Product("1", "Product", "Description", 10.00f);
        CartItem item = new CartItem("1", product, 2);

        item.updateQuantity(-3);

        assertEquals(0, item.getQuantity(), "Quantity should be set to zero");
    }
}
```

// Cart Test

```
import static org.junit.jupiter.api.Assertions.*;
import org.example.Cart;
import org.example.CartItem;
import org.example.Product;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class CartTest {
    private Cart cart;
    private CartItem item1;
    private CartItem item2;

    @BeforeEach
    public void setUp() {
        cart = new Cart();
        Product product1 = new Product("1", "Product1", "description 1", 10.00F);
        Product product2 = new Product("2", "Product2", "description 2", 15.00F);
        item1 = new CartItem("1", product1, 2);
        item2 = new CartItem("2", product2, 3);
    }

    @Test
    public void testAddItem() {
        cart.addItem(item1);
        assertEquals(1, cart.getItems().size(), "Items should match");
        assertEquals(20, cart.getTotal(), "Total should be 20");
    }

    @Test
    public void testRemoveItem() {
        cart.addItem(item1);
        cart.addItem(item2);
        cart.removeItem(item1.getId());
        assertEquals(1, cart.getItems().size(), "Should have 1 item left");
        assertEquals(45, cart.getTotal(), "Total should be 45 after removing item1");
    }

    @Test
    public void testGetTotalWithMultipleItems() {
        cart.addItem(item1);
        cart.addItem(item2);
        assertEquals(65, cart.getTotal(), "Total should sum to 65");
    }

    @Test
    public void testClearCart() {
        cart.addItem(item1);
```

```
        cart.addItem(item2);
        cart.clearCart();
        assertEquals(0, cart.getItems().size(), "Cart should be empty");
        assertEquals(0, cart.getTotal(), "Total should be 0 after clear");
    }
}
```

References:

[1] Amazon.com. [Online]. Available: https://aws.amazon.com/what-is/scrum/#:~:text=It%20describes%20a%20set%20of,problems%20cost%20effectively%20and%20sustainably. [Accessed: 22-Apr-2024].

[2] Atlassian, What is Scrum? 2018.

[3] "What is Scrum?," Scrum.org. [Online]. Available: https://www.scrum.org/resources/what-scrum-module. [Accessed: 22-Apr-2024].

[4] P. Sahithi and P. Kumar, "Implementing Scrum and Kanban Approaches for E-Commerce Web Application," An Agile Framework, vol. 9, pp. 2321–0613, 2021.

[5] S. Barnes, "Scrum vs. Waterfall: Which is right for your project?," Float.com, 29-Mar-2023. [Online]. Available: https://www.float.com/resources/scrum-vs-waterfall/. [Accessed: 22-Apr-2024].

[6] "Stripe-hosted page," Stripe.com. [Online]. Available: https://docs.stripe.com/checkout/quickstart. [Accessed: 24-Apr-2024].

[7] "Checkout," Paypal.com. [Online]. Available: https://developer.paypal.com/docs/checkout/. [Accessed: 24-Apr-2024].

[8] "Payment gateway benefits for businesses," Stripe.com. [Online]. Available: https://stripe.com/ie/resources/more/five-key-benefits-of-payment-gateways-for-businesses. [Accessed: 24-Apr-2024].

[9] EcomPedia, "The risk management in E-commerce: A complete guide," Ecompedia, 02-Oct-2023. .

[10] “Applying Scrum methodology on your e-commerce project,” Baldwin, 26-Feb-2015. [Online]. Available: https://www.baldwin.agency/blog/management/applying-scrum-methodology-on-your-e-commerce-project/. [Accessed: 26-Apr-2024].

[11] Belitsoft.com. [Online]. Available: https://belitsoft.com/custom-application-development-services/agile-methodology. [Accessed: 26-Apr-2024].

[12] J. K. Bickford, P. Malik, S. Bochtler, B. O’Malley, and B. Rehberg, “Making risk management work for agile,” BCG Global, 01-Aug-2019. [Online]. Available: https://www.bcg.com/capabilities/digital-technology-data/making-risk-management-work. [Accessed: 26-Apr-2024].

[13] “What is PCI DSS compliance? 12 requirements,” Stripe.com. [Online]. Available: https://stripe.com/ie/guides/pci-compliance. [Accessed: 26-Apr-2024].

[14] “Quality Department,” The GitLab Handbook. [Online]. Available: https://handbook.gitlab.com/handbook/engineering/quality/. [Accessed: 27-Apr-2024].

[15] R. RISHU_MISHRA Follow, “MVP (model view presenter) architecture pattern in android with example,” GeeksforGeeks, 28-Oct-2020. [Online]. Available: https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/. [Accessed: 27-Apr-2024].

[16] “Introduction to Java,” W3schools.com. [Online]. Available: https://www.w3schools.com/java/java_intro.asp. [Accessed: 27-Apr-2024].

[17] “Java classes and objects,” W3schools.com. [Online]. Available: https://www.w3schools.com/java/java_classes.asp. [Accessed: 27-Apr-2024].

[18] C. Simmonds, Mastering embedded Linux programming-second edition. Birmingham, England: Packt Publishing, 2017.

[19] S. Welch, Sean-david-welch - repositories. .

[20] “Integration testing: A detailed guide,” BrowserStack, 10-Mar-2023. [Online]. Available: https://www.browserstack.com/guide/integration-testing. [Accessed: 27-Apr-2024].

[21] P. Tylee, “The importance of testing and how to do it properly,” Linkedin.com, 31-Oct-2023. [Online]. Available: https://www.linkedin.com/pulse/importance-testing-how-do-properly-peter-tylee-lahjc/. [Accessed: 27-Apr-2024].

[22] G. Andrades, “What is Unit Testing? 6 Best Practices to Do it Right,” ACCELQ Inc, 26-Oct-2023. [Online]. Available: https://www.accelq.com/blog/unit-testing/. [Accessed: 27-Apr-2024].

[23] K. Nalawade, “How to write unit tests in java,” freecodecamp.org, 03-Apr-2023. [Online]. Available: https://www.freecodecamp.org/news/java-unit-testing/. [Accessed: 27-Apr-2024].

[24] Baeldung.com. [Online]. Available: https://www.baeldung.com/java-unit-testing-best-practices. [Accessed: 27-Apr-2024].

[25] D. Radigan, “Agile methodology testing best practices & why they matter,” Atlassian. [Online]. Available: https://www.atlassian.com/agile/software-development/testing. [Accessed: 27-Apr-2024].

[26] “What is CI CD testing? How Does It Work?,” Testsigma Blog, 24-Jul-2023. [Online]. Available: https://testsigma.com/blog/ci-cd-testing/. [Accessed: 27-Apr-2024].

[27] Marketing Ranorex, “End-to-end testing vs integration testing explained,” Ranorex, 01-Feb-2024. [Online]. Available: https://www.ranorex.com/blog/end-to-end-testing-vs-integration-testing-explained/. [Accessed: 27-Apr-2024].

[28] M. Rehkopf, “User stories,” Atlassian. [Online]. Available: https://www.atlassian.com/agile/project-management/user-stories. [Accessed: 28-Apr-2024].

[29] Gareth, “45 user story examples to inspire your agile team,” Parabol, 27-Jun-2023. [Online]. Available: https://www.parabol.co/blog/user-story-examples/. [Accessed: 28-Apr-2024].

[30] “User stories in agile - how to write with examples,” ActiveCollab. [Online]. Available: https://activecollab.com/blog/project-management/user-stories-in-agile. [Accessed: 28-Apr-2024].

[31] D. Radigan, Product backlog grooming. 2019.

[32] M. Rehkopf, “Jira burndown chart: step-by-step tutorial,” Atlassian. [Online]. Available: https://www.atlassian.com/agile/tutorials/burndown-charts. [Accessed: 28-Apr-2024].