



# APPLIED SOFTWARE ENGINEER

## Coursework 2

**University of West London**

**Academic Year:** 2023/24

**Course:** BSc Computer Science

**Module:** Applied Software Engineer

**Assignment:** Coursework 2

**Module Leader:**

**Student:** Nunzio Emanuele Sgroi

**Student ID:**

Case Study  
**Hill & Knowlton Looks for a New Knowledge Management System**

## Table of Contents

Table of Figures.....	1
<b>Task 1: Implementation</b> .....	2
<b>Task 2: Design Patterns</b> .....	5
<b>Task 3: Object Constraint Language (OCL)</b> .....	8
Preconditions and Postconditions .....	9
Constraints .....	11
<b>Task 4: Unit Tests</b> .....	12
Testing Examples.....	12
Testing Data and TDD Approach.....	15
System and Test Launch Instructions .....	15
Traceability Matrix .....	16
<b>References</b> .....	17
<b>Appendix</b> .....	17
Interfaces .....	17
Classes.....	17
Command Patterns .....	26
Unit Test.....	29

## Table of Figures

Figure 1 - Class Diagram.....	2
Figure 2 - Command Pattern Diagram .....	5
Figure 3 - Output of Workflow Simulator .....	8
Figure 4 - OCL: Employee make a purchase using Beenz.....	9
Figure 5 - OCL: Apply Beenz credits and adjust total price.....	9
Figure 6 - OCL: Add item to cart.....	10
Figure 7- OCL: Complete payment and empty cart .....	10
Figure 8 - OCL: Employee Beenz balance invariant.....	11
Figure 9 - OCL: Cart items integrity invariant.....	11
Figure 10 - OCL: Checkout completion invariant .....	12
Figure 11 - Traceability Matrix .....	16

## Task 1: Implementation

The software application developed for the use case "Purchasing Items in the hk.net Store Using Beenz Credits" was implemented following the class diagram from Coursework 1. This section details how the design was translated into a functioning Java application, adhering to object-oriented principles.

The structure of the application, follows what was designed in the class diagram. Every class and interface, was aligned with the specifications laid out in the diagram.

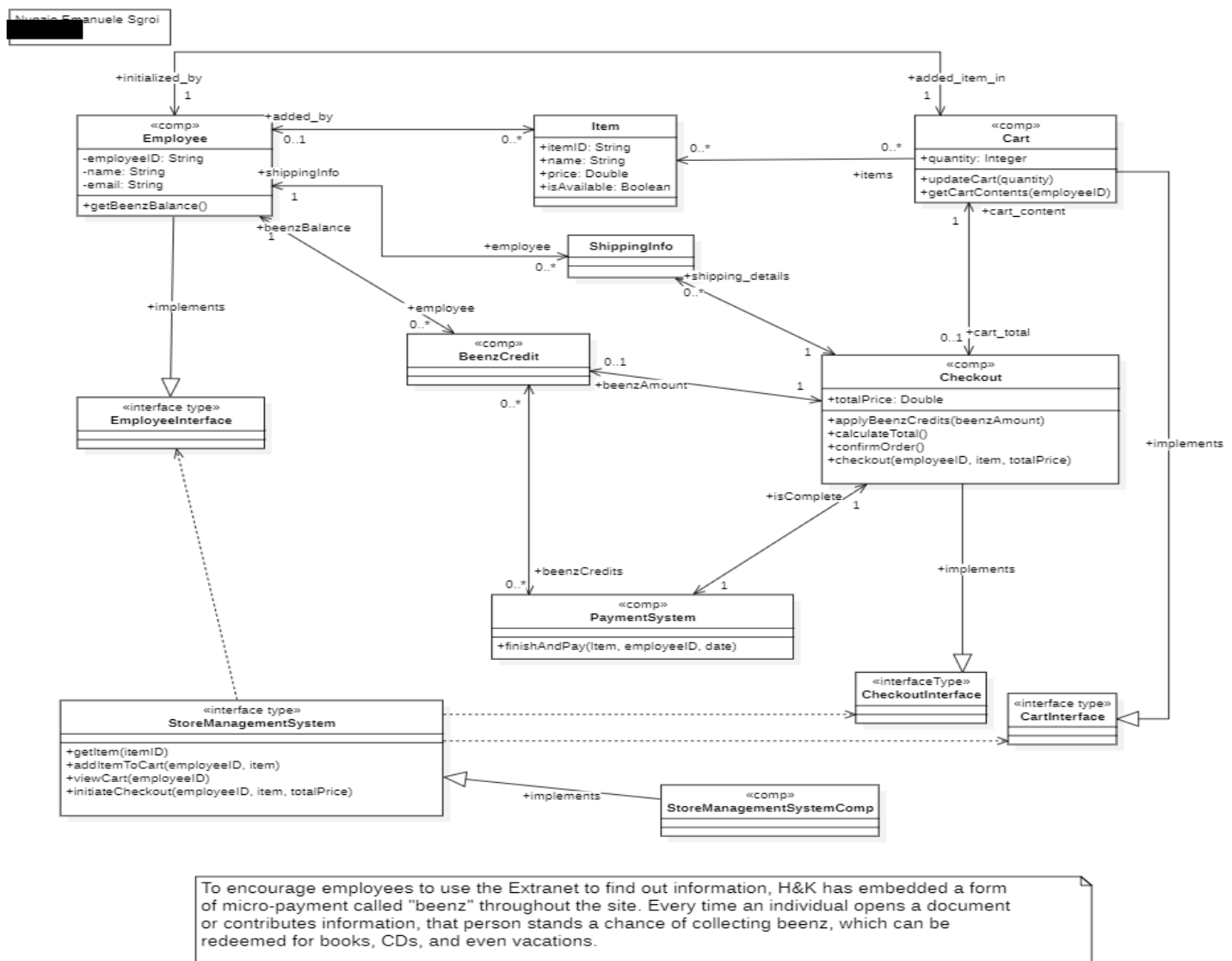


Figure 1 - Class Diagram

**Note:** Full code snippet is reported in Appendix section

### Encapsulation

Encapsulation was applied across all classes. For instance, in the Employee class, information such as employee ID, name, and email were encapsulated within private fields, with public getters and setters facilitating controlled access.

```

public class Employee implements EmployeeInterface{
    private String employeeID;
    private String name;
    private String email;

    //constructor
    public Employee(String employeeID, String name, String email) {
        this.employeeID = employeeID;
        this.name = name;
        this.email = email;
    }

    // Establishing relationships (getter and setter)
    public String getEmployeeID() {
        return employeeID;
    }
    public String getName() {
        return name;
    }
    public String getEmail() {
        return email;
    }
}

```

## Inheritance and Interface Implementation

The implementation of interfaces and the use of inheritance were crucial in building a solid architectural foundation for the system. The Employee class, as an example, adhered to EmployeeInterface, ensuring uniformity in method definitions across various class implementations. Same logic was applied to every class implementing an interface

```

public class Employee implements EmployeeInterface{ ... }

```

## Polymorphism

Polymorphism played a key role in making the system flexible. For instance, both Checkout and PaymentSystem used the CheckoutInterface. This meant that various payment methods could be handled in the same way, showing how polymorphism is useful in object-oriented programming.

```

public class PaymentSystem implements CheckoutInterface{ ... }
public class Checkout implements CheckoutInterface { ... }

```

## Association

Association was implemented through references between related classes (attributes). For example, Employee held a reference to Cart, enabling the establishment of a meaningful relationship between these entities. Each reference, is then followed by the getter and setter that allow the exchange of information between classes associated.

```

public class Employee implements EmployeeInterface{
    //references
    private Cart cart;

    // Establishing relationships (getter and setter)
    public void setCart(Cart cart){ this.cart = cart; }
    public Cart getCart(){ return this.cart; }
}

```

## Abstraction

In the application, the use of interfaces effectively demonstrates the concept of abstraction. This object-oriented concept allows the system to focus on what operations are performed, rather than how they are implemented. For example, the CheckoutInterface defines abstract methods for checkout processes, however, it is important to note that similar logic was applied in various interfaces (see Appendix).

```
public interface CheckoutInterface {
    void applyBeenzCredits(double beenzAmount);
    double calculateTotal(double totalPrice);
    void confirmOrder();
    void checkout(String employeeID, Item item, double totalPrice);
}
```

## System Class

The StoremanagementSystemComp class, acts as the core of the application, as shown in the class diagram, and it was developed to handle tasks like getting items, updating carts and initiating the checkout process. This work shows a clear use of object-oriented programming, making the application strong and easy to maintain. By implementing the StoreManagementSystem interface, this class exemplifies the use of inheritance, further reinforcing object-oriented design within the application's framework.

```
public class StoreManagementSystemComp implements StoreManagementSystem {
    private Map<String, Employee> employees = new HashMap<>();

    public void addEmployee(Employee employee) {
        employees.put(employee.getEmployeeID(), employee);
    }

    @Override
    public Item getItem(String itemID) {
        // Implementation to get an Item
        return null;
    }

    @Override
    public void addItemToCart(String employeeID, Item item) {
        // Implementation to add item to cart
    }

    @Override
    public Cart viewCart(String employeeID) {
        // Implementation to view cart
        return null;
    }

    @Override
    public void initiateCheckout(String employeeID, Item item, double
totalPrice) {
        // Implementation to initiate checkout
    }
}
```

```
public interface StoreManagementSystem {
    Item getItem(String itemID);
}
```

```

void addItemToCart(String employeeID, Item item);
Cart viewCart(String employeeID);
void initiateCheckout(String employeeID, Item item, double totalPrice);
}

```

## Task 2: Design Patterns

The Command Pattern is a behavioural design pattern in software engineering. It encapsulates requests or commands as objects, effectively separating the entity that issues the command from the one that executes it. This pattern enhances system modularity and scalability, as it allows commands to be issued and executed independently. The implementation of this pattern in a project typically aims to improve system structure, making it easier to adapt and expand in response to evolving requirements or functionalities.

### Command Pattern Structure

In this project, the Command Pattern was structured to encapsulate specific actions or requests within the software as objects. This approach transforms various operations like adding items to a cart or processing payments into executable commands. The pattern consists of four main components: the **Command interface**, **ConcreteCommand** classes (e.g., AddItemToCartCommand), the **Invoker** class (managing command execution), and **Receivers** (performing the actual operations).

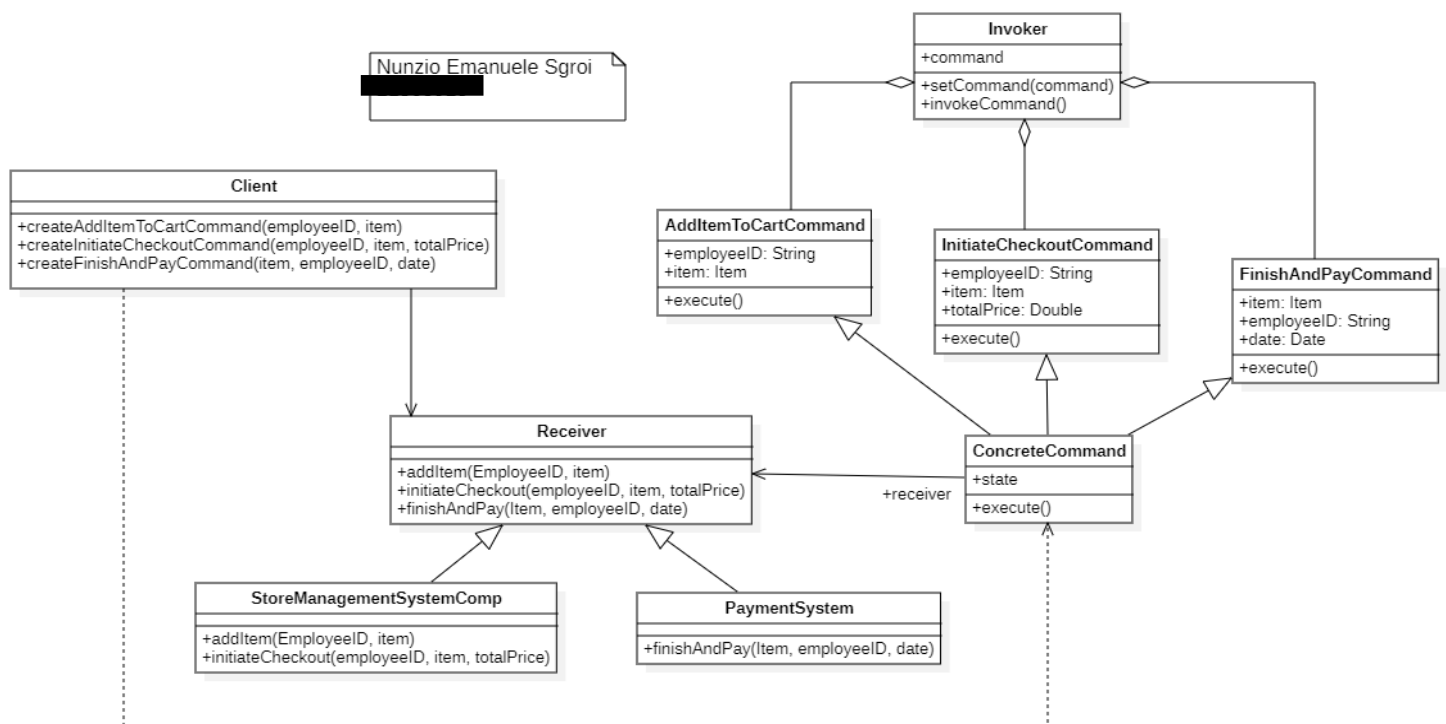


Figure 2 - Command Pattern Diagram

**Note:** Full code snippet is reported in Appendix section

## Command Interface

The Command interface is crucial in the Design Pattern, acting as a template for command objects. It defines the **execute()** method, ensuring uniformity across commands. This setup allows different commands to be executed by the invoker and facilitates their modification and extension

```
public interface Command {
    void execute();
}
```

## Concrete Commands

The **AddItemToCartCommand**, **InitiateCheckoutCommand**, and **FinishAndPayCommand** are concrete implementations of the Command interface. Each represents a specific action in the system:

- **AddItemToCartCommand**: Manages adding items to a shopping cart.
- **InitiateCheckoutCommand**: Initiates the checkout process.
- **FinishAndPayCommand**: Completes the payment process.

These commands interact with receiver classes like **StoreManagementSystemComp** or **PaymentSystem** to execute their respective operations. They encapsulate the command logic and delegate execution to the receivers, maintaining a clear separation of command initiation and execution.

Below is an example of implementation of AddItemToCart:Command:

```
public class AddItemToCartCommand implements Command {
    private StoreManagementSystemComp storeManagementSystem;
    private String employeeID;
    private Item item;
    public AddItemToCartCommand(StoreManagementSystemComp
storeManagementSystem, String employeeID, Item item) {
        this.storeManagementSystem = storeManagementSystem;
        this.employeeID = employeeID;
        this.item = item;
    }
    @Override
    public void execute() {
        System.out.println("Add item to cart command"); // for testing
        storeManagementSystem.addItemToCart(employeeID, item);
    }
}
```

## Invokers

The **CommandInvoker** class plays a central role in the Command Pattern. It acts as an intermediary that handles the execution of command objects. The primary responsibility of the **CommandInvoker** is to hold a command and trigger its execution at the appropriate time.

```
public class CommandInvoker {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
}
```

```

public void executeCommand() {
    if (command != null) {
        command.execute();
    }
}
}

```

## Receivers

In the implemented Command Pattern, **StoreManagementSystemComp** and **PaymentSystem** serve as the receivers. They are responsible for executing the actual business logic associated with each command. For instance:

- **StoreManagementSystemComp** handles actions like adding items to the cart and initiating the checkout process.
- **PaymentSystem** takes care of the final payment and order completion steps.

Each concrete command delegates its execution to these receivers, ensuring that the action it represents is carried out effectively.

```

@Override
public void execute() {
    System.out.println("Add item to cart command"); // for testing
    storeManagementSystem.addItemToCart(employeeID, item);
}

```

```

@Override
public void execute() {
    System.out.println("Finish and pay command"); // for testing
    paymentSystem.finishAndPay(item, employeeID, date);
}

```

```

@Override
public void execute() {
    System.out.println("Initiate checkout command"); // for testing
    storeManagementSystem.initiateCheckout(employeeID, item, totalprice);
}

```

**Note:** Please, refer to task 1 and/or appendix for the implementation of the classes mentioned above.

## Workflow Simulation

To demonstrate the practical application of the Command Pattern, I developed a Java class called **ShoppingWorkflowSimulator**. By creating and executing specific command instances through the **CommandInvoker**, this class showcases how different user actions can be encapsulated as commands and handled uniformly. This is the reason why I included messages printed in console within the execute method in the concrete command classes.

```

public class ShoppingWorkflowSimulator {
    public void simulateShoppingProcess() {
        // Objects for the commands
        StoreManagementSystemComp storeSystem = new

```

```

StoreManagementSystemComp();
    Employee employee = new Employee("empID", "Emanuele Sgroi",
"test@example.com");
    Item item = new Item("itemID", "Item Name", 20.99, true);
    // Simulate adding item to cart
    Command addItemCommand = new AddItemToCartCommand(storeSystem,
employee.getEmployeeID(), item);
    CommandInvoker invoker = new CommandInvoker();
    invoker.setCommand(addItemCommand);
    invoker.executeCommand();
    // Simulate initiating checkout
    Command checkoutCommand = new InitiateCheckoutCommand(storeSystem,
employee.getEmployeeID(), item, 20.0);
    invoker.setCommand(checkoutCommand);
    invoker.executeCommand();
    // Simulate finishing payment
    Command paymentCommand = new FinishAndPayCommand(new
PaymentSystem(), item, employee.getEmployeeID(), new Date());
    invoker.setCommand(paymentCommand);
    invoker.executeCommand();
}
public static void main(String[] args) {
    System.out.print("Nunzio Emanuele Sgroi - 21508918 \n");
    System.out.print("-----\n");
    ShoppingWorkflowSimulator simulator = new
ShoppingWorkflowSimulator();
    simulator.simulateShoppingProcess();
}
}

```

```

> Task :ShoppingWorkflowSimulator.main()
Nunzio Emanuele Sgroi - ██████████
-----
Add item to cart command
Initiate checkout command
Finish and pay command

BUILD SUCCESSFUL in 249ms
2 actionable tasks: 2 executed
09:34:56: Execution finished ':ShoppingWorkflowSimulator.main()'.

```

Figure 3 - Output of Workflow Simulator

### Task 3: Object Constraint Language (OCL)

This section implements OCL specifications from Coursework 1 into the Java application. OCL is pivotal for establishing business rules and ensuring the application adheres to necessary operational constraints and maintains data integrity.

## Preconditions and Postconditions

### Employee make a purchase using Beenz & Apply Beenz credits and adjust total price

**Operation:** Employee make a purchase using Beenz

- **Precondition:** The employee must have a Beenz balance greater than or equal to the amount of Beenz they wish to spend, and the amount being spent should be greater than zero.
- **Postcondition:** After the purchase, the employee's Beenz balance is reduced by the amount of Beenz spent.

**context** Checkout::applyBeenzCredits(beenzAmount: Double)

**pre:** self.cart.employee.beenzBalance >= beenzAmount and beenzAmount > 0

**post:** self.cart.employee.beenzBalance = self.cart.employee.beenzBalance@pre - beenzAmount

**Description:** This OCL statement ensures that employees can only use the Beenz Credits if the Beenz Balance is greater than 0. After using the Beenz Credits to make a purchase, the amount Beenz balance is recalculated.

Figure 4 - OCL: Employee make a purchase using Beenz

**Operation:** Apply Beenz credits and adjust total price

- **Precondition:** The amount of Beenz Credits to be applied must not exceed the employee's current Beenz balance and must be a positive value.
- **Postcondition:** The checkout's total price is updated to reflect the deduction of the applied Beenz Credits from the initial total price.

**context** Checkout::applyBeenzCredits(beenzAmount: Double)

**pre:** beenzAmount <= self.cart.employee.beenzBalance and beenzAmount > 0

**post:** self.totalPrice = self.totalPrice@pre - beenzAmount

**Description:** This OCL statement ensures that the amount of Beenz Credits an employee wishes to use is available in their account and is a positive number. Upon applying these credits, the total price in the checkout is reduced accordingly to reflect the new payable amount.

Figure 5 - OCL: Apply Beenz credits and adjust total price

In the **Checkout** class, the **applyBeenzCredits** method addresses two OCL rules. First, it ensures the employee's Beenz balance is positive before proceeding with a transaction. This aligns to the rule requiring a sufficient Beenz balance for purchases. Second, the method deducts the specified amount from the balance, aligning with the rule for adjusting total price after applying Beenz credits. Both OCL specifications are seamlessly integrated into a singular functional process.

```
@Override
public void applyBeenzCredits(double beenzAmount) {
    //OCL
    // Precondition check
    if (employee.getBeenzBalance() >= beenzAmount && beenzAmount > 0) {
        employee.setBeenzBalance(employee.getBeenzBalance() - beenzAmount);
        // Update totalPrice after applying Beenz Credits
        this.totalPrice -= beenzAmount;
    } else {
        // Logic where preconditions are not met
    }
}
```

## Add item to cart

<p><b>Operation:</b> Add item to cart</p> <ul style="list-style-type: none"> <li>• <b>Precondition:</b> The item must be available.</li> <li>• <b>Postcondition:</b> The item is added to the cart.</li> </ul> <p><b>context</b> StoreManagementSystem::addItemToCart(employeeID: String, item: Item)</p> <p><b>pre:</b> item.isAvailable</p> <p><b>post:</b> employee.cart.items-&gt;includes(item)</p> <p><b>Description:</b> In this OCL statement, the precondition checks that the Item instance passed to the “addItemToCart” operation is available, while the postcondition ensures that after the operation, the Item is included in the items collection of the Cart instance.</p>
--

Figure 6 - OCL: Add item to cart

In the **StoreManagementSystemComp** class, the **addItemToCart** method checks if the Item is available (**item.isAvailable()**). If the item is available, it is added to the employee's cart, ensuring the item is now part of the cart's collection.

```
@Override
public void addItemToCart(String employeeID, Item item) {
    Employee employee = employees.get(employeeID);
    Cart cart = employee.getCart();
    if (employee != null && item.isAvailable()) {
        cart.addItem(item);
    } else {
        // Logic if item not available or employee not found
    }
}
```

## Complete payment and empty cart

<p><b>Operation:</b> Complete Payment and Empty Cart.</p> <ul style="list-style-type: none"> <li>• <b>Precondition:</b> Not applicable</li> <li>• <b>Postcondition:</b> The payment is marked as complete, and the cart associated with the checkout is emptied of all items.</li> </ul> <p><b>context</b> PaymentSystem::finishAndPay(item: Item, employeeID: String, date: Date)</p> <p><b>post:</b> self.checkout.isComplete = true</p> <p><b>post:</b> self.checkout.cart.items-&gt;forAll(i   i.quantity = 0)</p> <p><b>Description:</b> The “finishAndPay” operation marks the checkout associated with the payment system as complete and sets the quantity of items in the cart to zero, effectively emptying the cart.</p>
---

Figure 7- OCL: Complete payment and empty cart

In the **PaymentSystem** class, the **finishAndPay** method has been adapted to fulfil these conditions. Upon execution, it sets the checkout status to complete and iterates through the cart items, setting their quantities to zero, effectively clearing the cart.

```

public void finishAndPay(Item item, String employeeID, Date date) {
    Checkout checkout = this.getCheckout();
    // Mark payment as complete
    checkout.setIsComplete(true);
    // Empty the cart
    Cart cart = checkout.getCart();
    for (Item cartItem : cart.getItems()) {
        cartItem.setQuantity(0);
    }
}

```

## Constraints

### Employee Beenz balance invariant

- Every employee's account balance of Beenz Credits must always be non-negative.

**context** Employee

**invariant:** self.beenzBalance >= 0

Figure 8 - OCL: Employee Beenz balance invariant

In the Employee class, the setBeenzBalance method has been adjusted to enforce this rule. It checks that the Beenz balance is non-negative before updating the balance.

```

public void setBeenzBalance(double beenzBalance) {
    //OCL --> ensure the Beenz balance is never negative
    if (beenzBalance >= 0) {
        this.beenzBalance = beenzBalance;
    } else {
        // Logic where the balance would be negative
    }
}

```

### Cart items integrity invariant

- A cart should always reference the items it contains, ensuring there are no orphaned items not linked to a cart.

**context** Cart

**invariant:** self.items->forAll(i | i.cart = self)

Figure 9 - OCL: Cart items integrity invariant

The addItem method in the Cart class has been modified to ensure this relationship is maintained. Whenever an item is added to the cart, the cart reference in the item is set accordingly. This guarantees that each item is consistently linked back to its respective cart.

```

public void addItem(Item item) {
    //OCL --> Cart items integrity invariant
    if (item != null) {
        items.add(item);
        item.setCart(this); // Set the cart reference in the item
    }
}

```

## Checkout completion invariant

<ul style="list-style-type: none"> <li>• If a checkout's payment process is marked as complete, then certain conditions must be satisfied, such as payment confirmation received, and the cart must be empty</li> </ul> <p><b>context</b> Checkout</p> <p><b>invariant:</b> self.isComplete implies (self.paymentConfirmed and self.cart.isEmpty())</p>
---

Figure 10 - OCL: Checkout completion invariant

The **Checkout** class has been updated with two boolean attributes, **isComplete** and **paymentConfirmed**. Methods like **completeCheckout** enforce the invariant by checking that payment is confirmed and the cart is empty before setting **isComplete** to true.

```
private boolean isComplete = false;
private boolean paymentConfirmed = false;
@Override
public void confirmOrder() {
    this.paymentConfirmed = true;
}
public void completeCheckout() {
    if (paymentConfirmed && cart.quantity == 0) {
        this.isComplete = true;
    } else { // Logic where payment isn't confirmed or cart isn't empty}}
}
```

## Task 4: Unit Tests

In the project, each part was tested individually to make sure it worked correctly on its own. This involved creating separate test classes for different Java classes, like **EmployeeTest** and **CheckoutTest**, using **JUnit** as the main tool. The focus was on checking if the getters and setters functioned properly and ensuring the relationships between objects were correct. This approach not only ensured that each part of the application was reliable but also reflected an understanding of fundamental software testing techniques, essential for the overall integrity of the application.

### Testing Examples

**Note:** Full code snippet is reported in Appendix section

The first stage of the unit testing in this application was centred around verifying class functionalities, particularly through testing getters, setters, and data manipulation methods. This ensured object relationships were properly maintained. For instance, the **EmployeeTest** class tested the accuracy of employee data handling and the correct linkage of associated entities like **Cart**. This method was consistently applied across all classes, validating the effectiveness of their data management processes and inter-class associations.

```
@Test
void testNameSetterGetter() {
    Employee employee = new Employee("id4", "Emanuele",
"test@example.com");
    employee.setName("Nunzio");
    assertEquals("Nunzio", employee.getName());
}
@Test
void testCartSetterGetter() {
    Employee employee = new Employee("id5", "Emanuele",
```

```

"test@example.com");
    Cart cart = new Cart(2);
    employee.setCart(cart);
    assertEquals(cart, employee.getCart());
}

```

The second stage of testing focused on Object Constraint Language (OCL) compliance. This involved creating unit tests for specific methods to verify adherence to established business rules and constraints. The following examples illustrate how these tests were implemented, showcasing the application's alignment with OCL specifications.

1. **Employee's Beenz Balance Test:** Test checks if the balance cannot be negative, adhering to the OCL rule highlighted in the previous task.

```

@Test
void testNegativeBeenzBalance() {
    Employee employee = new Employee("id1", "Emanuele", "test@example.com");
    employee.setBeenzBalance(-100);
    assertTrue(employee.getBeenzBalance() >= 0);
}

```

2. **Cart Item Link Test:** Verifies that items added to the cart are linked correctly.

```

@Test
void testAddItem() {
    Cart cart = new Cart(5);
    Item item = new Item("item1", "Item1", 10.99, true);
    cart.addItem(item);
    assertTrue(cart.getItems().contains(item));
    assertEquals(cart, item.getCart()); // Verifying the back-reference
    from item to cart
}

```

3. **Apply Beenz Credits Test in Checkout:** Ensures Beenz credits are applied correctly under specified conditions.

```

@Test
void testApplyBeenzCredits() {
    Checkout checkout = new Checkout(100.0, false);
    Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
    checkout.setEmployee(employee);
    employee.setBeenzBalance(50);
    checkout.applyBeenzCredits(30);
    assertEquals(20, employee.getBeenzBalance(), 0.001);
    assertEquals(70, checkout.getTotalPrice(), 0.001);
}

```

4. **Test for Initiating Checkout Under Correct Conditions:** Verifies that checkout initiates only if specific preconditions are met, adhering to the OCL rules.

```

@Test
void testInitiateCheckout() {
    StoreManagementSystemComp system = new StoreManagementSystemComp();
    Employee employee = new Employee("empId", "Test", "test@example.com");
    Item item = new Item("item2", "Item2", 20.0, true);
}

```

```

Checkout checkout = new Checkout(20.0, true);
employee.setCart(new Cart(2));
employee.getCart().setCheckout(checkout);
system.addEmployee(employee);
system.initiateCheckout(employee.getEmployeeID(), item, 20.0);
assertEquals(20.0, checkout.getTotalPrice(), 0.001);
}

```

##### 5. Test for Completing Payment and Emptying Cart: Confirms that after payment, the cart is emptied and the payment is marked as complete.

```

@Test
void testFinishAndPay() {
    PaymentSystem paymentSystem = new PaymentSystem();
    Checkout checkout = new Checkout(100.0, true);
    Cart cart = new Cart(1);
    Item item = new Item("item1", "Item1", 10.0, true);
    cart.addItem(item);
    checkout.setCart(cart);
    paymentSystem.setCheckout(checkout);
    paymentSystem.finishAndPay(item, "id1", new Date());
    assertTrue(checkout.isComplete());
    assertEquals(0,
cart.getItems().stream().mapToInt(Item::getQuantity).sum()); // Check if
cart is empty
}

```

Tests on command and invoker structures were also carried out to validate their functionality. These tests ensured that commands executed correctly via the invoker, confirming the proper use of the Command Pattern. For example, using JUnit's BeforeEach annotation, I prepared consistent test conditions for each test run.

```

@BeforeEach
void setUp() {
    StoreManagementSystemComp system = new StoreManagementSystemComp();
    Employee employee = new Employee("id", "Test User", "test@email.com");
    system.addEmployee(employee);
    Item item = new Item("item1", "Test Item", 10.0, true);
    Command addItemCommand = new AddItemToCartCommand(system,
employee.getEmployeeID(), item);
    CommandInvoker invoker = new CommandInvoker();
    invoker.setCommand(addItemCommand);
}

```

Another testing method involved intentionally failing tests by inputting incorrect data into functions. This strategy tested the system's robustness against unexpected inputs. An example of this testing method included initiating the checkout process with an inaccurate price, evaluating the system's response to this error. up.

```

@Test
void testInitiateCheckoutWrongPrice() {
    StoreManagementSystemComp system = new StoreManagementSystemComp();
    Employee employee = new Employee("empId", "Test", "test@example.com");
    Item item = new Item("item2", "Item2", 20.0, true);
    Checkout checkout = new Checkout(20.0, true);
    employee.setCart(new Cart(2));
}

```

```

employee.getCart().setCheckout(checkout);
system.addEmployee(employee);
system.initiateCheckout(employee.getEmployeeID(), item, 20.0);
assertEquals(30.0, checkout.getTotalPrice(), 0.001); // This assertion
is expected to fail
}

```

## Testing Data and TDD Approach

The testing phase employed realistic data sets to simulate actual user scenarios, enhancing the applicability and reliability of the tests. This approach was very important for validating the software's performance and handling under real-world conditions.

```

@Test
void testEmailSetterGetter() {
    Employee employee = new Employee("id2", "Emanuele",
    "test@example.com");
    employee.setEmail("newemail@example.com");
    assertEquals("newemail@example.com", employee.getEmail());
}

```

Furthermore, the development process was guided by the principles of **Test-Driven Development (TDD)**. In TDD, test cases are created prior to writing the functional code, setting a clear path for development and ensuring that each code segment fulfils its intended purpose from the start. This methodology not only streamlined the coding process but also significantly reduced the potential for errors and rework. Furthermore, TDD encourages simple designs and inspires confidence, both of which were reflected in the project's development cycle (Unadkat, 2023).

## System and Test Launch Instructions

Below are the step-by-step instructions for setting up and running the developed software application and its unit tests.

### Prerequisites:

- Java Development Kit (JDK) version 11 or higher is required.
- An Integrated Development Environment (IDE) compatible with Gradle and JUnit, such as IntelliJ IDEA, should be installed.
- Access to the project's source code, obtainable from <https://github.com/Emanuele-Sgroi/Software-Design-and-Testing>

### Environment Setup:

1. Open the project in the chosen IDEA.
2. Ensure Gradle is synchronized and the project structure is correctly recognized by the IDE.

### Running the System:

Initiate the main application class, `StoreManagementSystemComp`, using the IDE's run feature.

### Executing Tests:

1. In the project structure, locate the `src/test/java` directory.
2. Execute all tests within the test package or individual test classes as needed.
3. Observe the test results in the IDE's test output window.

Detailed reports from these tests can be found in the “test-results” directory, created after the test execution.

### Traceability Matrix

Requirement	TC ID	Source Code File(s)	Test Case(s)	Verified	Result	Expected
Add items to cart	System_management_01	StoreManagementSystemComp.java, Cart.java, Item.java	testAddItemToCart() in StoreManagementSystemCompTest.java	Yes	Pass	Yes
Initiate the checkout	System_management_02	StoreManagementSystemComp.java, Cart.java, Item.java, Employee.java	testInitiateCheckout() in StoreManagementSystemCompTest.java	Yes	Pass	Yes
Initiate the checkout with wrong price number	System_management_03	StoreManagementSystemComp.java, Cart.java, Item.java, Employee.java	testInitiateCheckoutWrongPrice() in StoreManagementSystemCompTest.java	Yes	Fail	Yes
Employee access Beenz balance	Employee_01	Employee.java	testBeenzBalance() in EmployeeTest.java	Yes	Pass	Yes
Retrieve employee ID	Employee_02	Employee.java	testIdSetterGetter() in EmployeeTest.java	Yes	Pass	Yes
Retrieve wrong employee ID	Employee_03	Employee.java	testIdSetterGetterFail() in EmployeeTest.java	Yes	Fail	Yes
Testing relationship between Employee and Cart	Employee_04	Employee.java, Cart.java	testCartSetterGetter() in EmployeeTest.java	Yes	Pass	Yes
Check if cart is emptied at finish and pay	Payment_system_01	PaymentSystem.java, Cart.java, Checkout.java	testFinishAndPay() in PaymentSystemTest.java	Yes	Pass	Yes
Check if cart is NOT emptied at finish and pay	Payment_system_02	PaymentSystem.java, Cart.java, Checkout.java	testFinishAndPayFail() in PaymentSystemTest.java	Yes	Fail	Yes
Testing relationship between Employee and PaymentSystem	Payment_system_03	PaymentSystem.java, Employee.java	testSetGetEmployee() in PaymentSystemTest.java	Yes	Pass	Yes
Testing relationship between BeenzCredit and PaymentSystem	Payment_system_04	PaymentSystem.java, BeenzCredit.java	testSetGetBeenzCredit() in PaymentSystemTest.java	Yes	Pass	Yes
Testing relationship between Checkout and PaymentSystem	Payment_system_05	PaymentSystem.java, Checkout.java	testSetGetCheckout() in PaymentSystemTest.java	Yes	Pass	Yes
Apply specific number of Beenz Credit	Checkout_01	Checkout.java, Employee.java	testApplyBeenzCredits() in CheckoutTest.java	Yes	Pass	Yes
Testing wrong number of Beenz Credit applied	Checkout_02	Checkout.java, Employee.java	testApplyBeenzCreditsFail() in CheckoutTest.java	Yes	Fail	Yes
Test confirm order TRUE	Checkout_03	Checkout.java	testConfirmOrder() in CheckoutTest.java	Yes	Pass	Yes
Test confirm order FALSE	Checkout_04	Checkout.java	testConfirmOrderFail() in CheckoutTest.java	Yes	Fail	Yes
Complete checkout	Checkout_05	Checkout.java	testCompleteCheckout() in CheckoutTest.java	Yes	Pass	Yes
Check that the Cart contains item	Cart_01	Cart.java, Item.java	testAddItem() in CartTest.java	Yes	Pass	Yes
Check that the Cart DO NOT contains item	Cart_02	Cart.java, Item.java	testAddItemFail() in CartTest.java	Yes	Fail	Yes
Test add item to cart command execution	Command_01	AddItemToCartCommand.java	execute() in AddItemToCartCommandTest.java	Yes	Pass	Yes
Test finish and pay command execution	Command_02	FinishAndPayCommand.java	execute() in FinishAndPayCommandTest.java	Yes	Pass	Yes
Test Initiate checkout command execution	Command_03	InitiateCheckoutCommand.java	execute() in InitiateCheckoutCommandTest.java	Yes	Pass	Yes

Figure 11 - Traceability Matrix

## References

Unadkat, J. (2023, June 14). *What is Test Driven Development (TDD)?* Retrieved from browserstack.com: <https://www.browserstack.com/guide/what-is-test-driven-development#:~:text=In%20layman's%20terms%2C%20Test%20Driven,unit%20test%20creation%2C%20and%20refactoring.>

## Appendix

### Interfaces

#### StoreManagementSystem

```
package com.ase_assignment;

public interface StoreManagementSystem {
    Item getItem(String itemID);
    void addItemToCart(String employeeID, Item item);
    Cart viewCart(String employeeID);
    void initiateCheckout(String employeeID, Item item, double totalPrice);
}
```

#### EmployeeInterface

```
package com.ase_assignment;

public interface EmployeeInterface {
    double getBeenzBalance();
}
```

#### CartInterface

```
package com.ase_assignment;

public interface CartInterface {
    void updateCart(int quantity);
    Cart getCartContents(String employeeID);
}
```

#### CheckoutInterface

```
package com.ase_assignment;

public interface CheckoutInterface {
    void applyBeenzCredits(double beenzAmount);
    double calculateTotal(double totalPrice);
    void confirmOrder();
    void checkout(String employeeID, Item item, double totalPrice);
}
```

### Classes

#### StoreManagementSystemComp

```

package com.ase_assignment;

import java.util.HashMap;
import java.util.Map;

public class StoreManagementSystemComp implements StoreManagementSystem {
    private Map<String, Employee> employees = new HashMap<>();

    public Map<String, Employee> getEmployees () {
        return employees;
    }

    public void setEmployees (Map<String, Employee> employees) {
        this.employees = employees;
    }

    public void addEmployee (Employee employee) {
        employees.put (employee.getEmployeeID (), employee);
    }

    @Override
    public Item getItem (String itemID) {
        // Implementation to get an Item
        return null;
    }

    @Override
    public void addItemToCart (String employeeID, Item item) {
        Employee employee = employees.get (employeeID);
        Cart cart = employee.getCart ();
        if (employee != null && item != null && item.isAvailable ()) {
            cart.addItem (item);
        } else {
            // Logic if item not available or employee not found
        }
    }

    @Override
    public Cart viewCart (String employeeID) {
        // Implementation to view cart
        return null;
    }

    @Override
    public void initiateCheckout (String employeeID, Item item, double
totalPrice) {
        // Implementation to initiate checkout
    }
}

```

## Employee

```

package com.ase_assignment;

public class Employee implements EmployeeInterface {
    private String employeeID;
    private String name;
    private String email;
    //references
    private Cart cart;
    private Item item;
    private ShippingInfo shippingInfo;
    private BeenzCredit beenzCredit;
    private double beenzBalance;
}

```

```
//constructor
public Employee(String employeeID, String name, String email) {
    this.employeeID = employeeID;
    this.name = name;
    this.email = email;
}

// Establishing relationships (getter and setter)
public String getEmployeeID() {
    return employeeID;
}
public void setEmployeeID(String employeeID) {
    this.employeeID = employeeID;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public void setEmail(String email) {
    this.email = email;
}
public String getEmail() {
    return email;
}

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

//Item
public void setItem(Item item){
    this.item = item;
}
public Item getItem(){
    return this.item;
}

//ShippingInfo
public void setShippingInfo(ShippingInfo shippingInfo){
    this.shippingInfo = shippingInfo;
}
public ShippingInfo getShippingInfo(){
    return this.shippingInfo;
}

//BeenzCredit
public void setBeenzCredit(BeenzCredit beenzCredit){
    this.beenzCredit = beenzCredit;
}
public BeenzCredit getBeenzCredit(){
    return this.beenzCredit;
}

@Override
```

```

public double getBeenzBalance() {
    return this.beenzBalance;
}

public void setBeenzBalance(double beenzBalance) {
    //OCL --> ensure the Beenz balance is never negative
    if (beenzBalance >= 0) {
        this.beenzBalance = beenzBalance;
    } else {
        // Logic where the balance would be negative
    }
}
}

```

### BeenzCredit

```

package com.ase_assignment;

public class BeenzCredit {
    //references
    private Employee employee;
    private Checkout checkout;
    private PaymentSystem paymentSystem;

    // Establishing relationships (getter and setter)

    //Employee
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
    public Employee getEmployee() {
        return employee;
    }

    //Checkout
    public void setCheckout(Checkout checkout) {
        this.checkout = checkout;
    }
    public Checkout getCheckout() {
        return checkout;
    }

    //PaymentSystem
    public void setPaymentSystem(PaymentSystem paymentSystem) {
        this.paymentSystem = paymentSystem;
    }
    public PaymentSystem getPaymentSystem() {
        return paymentSystem;
    }
}

```

### Cart

```

package com.ase_assignment;

import java.util.ArrayList;
import java.util.List;

public class Cart implements CartInterface {
    int quantity;
    //references
}

```

```

private Employee employee;
private List<Item> items = new ArrayList<>();
private Checkout checkout;

// Constructor
public Cart(int quantity) {
    this.quantity = quantity;
}

// Establishing relationships (getter and setter)

//Employee
public void setEmployee (Employee employee){
    this.employee = employee;
}
public Employee getEmployee(){
    return this.employee;
}

//Item

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

public List<Item> getItems () {
    return this.items;
}
public void addItem(Item item) {
    //OCL --> Cart items integrity invariant
    if (item != null) {
        items.add(item);
        item.setCart(this); // Set the cart reference in the item
    }
}

//Checkout
public void setCheckout (Checkout checkout) {
    this.checkout = checkout;
}
public Checkout getCheckout () {
    return checkout;
}

@Override
public void updateCart(int quantity) {
    System.out.print("updateCart() implementation");
}

@Override
public Cart getCartContents (String employeeID) {
    return null;
}
}

```

## Checkout

```

package com.ase_assignment;

public class Checkout implements CheckoutInterface {

```

```
private double totalPrice;
//references
private Cart cart;
private ShippingInfo shippingInfo;
private BeenzCredit beenzCredit;
private PaymentSystem paymentSystem;
private Employee employee;
private boolean isComplete = false;
private boolean paymentConfirmed = false;

// Constructor
public Checkout(double totalPrice, boolean isComplete) {
    this.totalPrice = totalPrice;
    this.isComplete = isComplete;
}

// Establishing relationships (getter and setter)

public double getTotalPrice() {
    return totalPrice;
}

public void setTotalPrice(double totalPrice) {
    this.totalPrice = totalPrice;
}

public boolean isComplete() {
    return isComplete;
}

public void setComplete(boolean complete) {
    isComplete = complete;
}

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

public Cart getCart() {
    return cart;
}

//ShippingInfo
public void setShippingInfo(ShippingInfo shippingInfo) {
    this.shippingInfo = shippingInfo;
}

public ShippingInfo getShippingInfo() {
    return shippingInfo;
}

//BeenzCredit
public void setBeenzCredit(BeenzCredit beenzCredit) {
    this.beenzCredit = beenzCredit;
}

public BeenzCredit getBeenzCredit() {
    return beenzCredit;
}

//PaymentSystem
public void setPaymentSystem(PaymentSystem paymentSystem) {
    this.paymentSystem = paymentSystem;
}
}
```

```

public PaymentSystem getPaymentSystem() {
    return paymentSystem;
}

//Employee
public void setEmployee(Employee employee) {
    this.employee = employee;
}
public Employee getEmployee() {
    return employee;
}

@Override
public void applyBeenzCredits(double beenzAmount) {
    //OCL
    // Precondition check
    if (employee.getBeenzBalance() >= beenzAmount && beenzAmount > 0) {
        employee.setBeenzBalance(employee.getBeenzBalance() -
beenzAmount);
        // Update totalPrice after applying Beenz Credits
        this.totalPrice -= beenzAmount;
    } else {
        // Logic where preconditions are not met
    }
}

@Override
public double calculateTotal(double totalPrice) {

    return totalPrice;
}

@Override
public void confirmOrder() {
    this.paymentConfirmed = true;
}

@Override
public void checkout(String employeeID, Item item, double totalPrice) {
    // Implementation here
}

public void setIsComplete(boolean isComplete) {
}

public void completeCheckout() {
    if (paymentConfirmed && cart.quantity == 0) {
        this.isComplete = true;
    } else {
        // Logic where payment isn't confirmed or cart isn't empty
    }
}
}

```

## Item

```

package com.ase_assignment;

public class Item {
    private String itemID;
    private String name;
}

```

```
private double price;
public boolean isAvailable;
//references
private Employee employee;
private Cart cart;

// Constructor
public Item(String itemID, String name, double price, boolean
isAvailable) {
    this.itemID = itemID;
    this.name = name;
    this.price = price;
    this.isAvailable = isAvailable;
}

// Establishing relationships (getter and setter)

public String getItemID() {
    return itemID;
}

public void setItemID(String itemID) {
    this.itemID = itemID;
}

public String getName() {
    return name;
}

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

public double getPrice() {
    return price;
}

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

//Employee
public void setEmployee(Employee employee){
    this.employee = employee;
}
public Employee getEmployee(){
    return this.employee;
}

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

public Cart getCart() {
    return cart;
}

public boolean isAvailable() {
    return true;
}
```

```

public void setQuantity(int quantity) {
}

public int getQuantity() {
    return 0;
}
}

```

## PaymentSystem

```

package com.hknet;

import java.util.Date;

public class PaymentSystem implements CheckoutInterface{
    //references
    private Checkout checkout;
    private BeenzCredit beenzCredit;
    private Employee employee;

    //Checkout
    public void setCheckout(Checkout checkout) {
        this.checkout = checkout;
    }
    public Checkout getCheckout() {
        return checkout;
    }

    //BeenzCredit
    public void setBeenzCredit(BeenzCredit beenzCredit) {
        this.beenzCredit = beenzCredit;
    }
    public BeenzCredit getBeenzCredit() {
        return beenzCredit;
    }

    //Employee
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
    public Employee getEmployee() {
        return employee;
    }

    @Override
    public void applyBeenzCredits(double beenzAmount) {
        // Implementation for applying Beenz credits
    }

    @Override
    public double calculateTotal(double totalPrice) {
        // Implementation for calculating the total price
        return 0; // Placeholder return
    }

    @Override
    public void confirmOrder() {
        // Implementation for confirming the order
    }
}

```

```

@Override
public void checkout(String employeeID, Item item, double totalPrice) {
    // Implementation for checking out
}

public void finishAndPay(Item item, String employeeID, Date date) {
    // Logic for finishing payment
}
}

```

## ShippingInfo

```

package com.ase_assignment;

public class ShippingInfo {

    //references
    private Employee employee;

    // Establishing relationships (getter and setter)

    //Employee
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
    public Employee getEmployee() {
        return employee;
    }
}

```

## Command Patterns

### CommandInvoker

```

package com.ase_assignment.invokers;

import com.ase_assignment.commands.Command;

public class CommandInvoker {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void executeCommand() {
        if (command != null) {
            command.execute();
        }
    }
}

```

### Command Interface

```

package com.ase_assignment.commands;

public interface Command {
    void execute();
}

```

### AddItemToCartCommand

```

package com.ase_assignment.commands;

import com.ase_assignment.StoreManagementSystemComp;
import com.ase_assignment.Item;

public class AddItemToCartCommand implements Command {
    private StoreManagementSystemComp storeManagementSystem;
    private String employeeID;
    private Item item;
    public AddItemToCartCommand(StoreManagementSystemComp
storeManagementSystem, String employeeID, Item item) {
        this.storeManagementSystem = storeManagementSystem;
        this.employeeID = employeeID;
        this.item = item;
    }
    @Override
    public void execute() {
        System.out.println("Add item to cart command"); // for testing
        storeManagementSystem.addItemToCart(employeeID, item);
    }
}

```

### FinishAndPayCommand

```

package com.ase_assignment.commands;

import com.ase_assignment.PaymentSystem;
import com.ase_assignment.Item;

import java.util.Date;

public class FinishAndPayCommand implements Command {
    private PaymentSystem paymentSystem;
    private Item item;
    private String employeeID;
    private Date date;

    public FinishAndPayCommand(PaymentSystem paymentSystem, Item item,
String employeeID, Date date) {
        this.paymentSystem = paymentSystem;
        this.item = item;
        this.employeeID = employeeID;
        this.date = date;
    }

    @Override
    public void execute() {
        System.out.println("Finish and pay command"); // for testing
        paymentSystem.finishAndPay(item, employeeID, date);
    }
}

```

### InitiateCheckoutCommand

```

package com.ase_assignment.commands;

import com.ase_assignment.Item;
import com.ase_assignment.StoreManagementSystemComp;

public class InitiateCheckoutCommand implements Command{
    private StoreManagementSystemComp storeManagementSystem;

```

```

private String employeeID;
private Item item;
private Double totalprice;
public InitiateCheckoutCommand(StoreManagementSystemComp
storeManagementSystem, String employeeID, Item item, Double totalprice){
    this.storeManagementSystem = storeManagementSystem;
    this.employeeID = employeeID;
    this.item = item;
    this.totalprice = totalprice;
}
@Override
public void execute() {
    System.out.println("Initiate checkout command"); // for testing
    storeManagementSystem.initiateCheckout(employeeID, item,
totalprice);
}
}

```

### ShoppingWorkflowSimulator

```

package com.ase_assignment;

import com.ase_assignment.commands.AddItemToCartCommand;
import com.ase_assignment.commands.FinishAndPayCommand;
import com.ase_assignment.commands.InitiateCheckoutCommand;
import com.ase_assignment.commands.Command;
import com.ase_assignment.invokers.CommandInvoker;

import java.util.Date;

public class ShoppingWorkflowSimulator {
    public void simulateShoppingProcess() {
        // Objects for the commands
        StoreManagementSystemComp storeSystem = new
StoreManagementSystemComp();
        Employee employee = new Employee("empID", "Emanuele Sgroi",
"test@example.com");
        Item item = new Item("itemID", "Item Name", 20.99, true);
        // Simulate adding item to cart
        Command addItemCommand = new AddItemToCartCommand(storeSystem,
employee.getEmployeeID(), item);
        CommandInvoker invoker = new CommandInvoker();
        invoker.setCommand(addItemCommand);
        invoker.executeCommand();
        // Simulate initiating checkout
        Command checkoutCommand = new InitiateCheckoutCommand(storeSystem,
employee.getEmployeeID(), item, 20.0);
        invoker.setCommand(checkoutCommand);
        invoker.executeCommand();
        // Simulate finishing payment
        Command paymentCommand = new FinishAndPayCommand(new
PaymentSystem(), item, employee.getEmployeeID(), new Date());
        invoker.setCommand(paymentCommand);
        invoker.executeCommand();
    }
    public static void main(String[] args) {
        System.out.print("Nunzio Emanuele Sgroi - 21508918 \n");
        System.out.print("-----\n");
        ShoppingWorkflowSimulator simulator = new
ShoppingWorkflowSimulator();
        simulator.simulateShoppingProcess();
    }
}

```

```

}
}

```

## Unit Test

### StoreManagementSystemCompTest

```

package com.ase_assignment;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

public class StoreManagementSystemCompTest {
    private StoreManagementSystemComp system;
    private Employee employee;
    private Item item;
    private Checkout checkout;

    @BeforeEach
    void setUp() {
        system = new StoreManagementSystemComp();
        employee = new Employee("id1", "Emanuele", "test@example.com");
        Cart cart = new Cart(1);
        employee.setCart(cart);
        system.addEmployee(employee);

        item = new Item("item1", "Item1", 10.0, true);
    }

    @Test
    void testAddItemToCart() {
        system.addItemToCart(employee.getEmployeeID(), item);
        assertTrue(employee.getCart().getItems().contains(item));
    }

    @Test
    void testInitiateCheckout() {
        StoreManagementSystemComp system = new StoreManagementSystemComp();
        Employee employee = new Employee("empId", "Test",
"test@example.com");
        Item item = new Item("item2", "Item2", 20.0, true);
        Checkout checkout = new Checkout(20.0, true);
        employee.setCart(new Cart(2));
        employee.getCart().setCheckout(checkout);
        system.addEmployee(employee);
        system.initiateCheckout(employee.getEmployeeID(), item, 20.0);
        assertEquals(20.0, checkout.getTotalPrice(), 0.001);
    }

    //the following test is expected to fail
    @Test
    void testInitiateCheckoutWrongPrice() {
        StoreManagementSystemComp system = new StoreManagementSystemComp();
        Employee employee = new Employee("empId", "Test",
"test@example.com");
        Item item = new Item("item2", "Item2", 20.0, true);
        Checkout checkout = new Checkout(20.0, true);
    }

```

```

        employee.setCart(new Cart(2));
        employee.getCart().setCheckout(checkout);
        system.addEmployee(employee);
        system.initiateCheckout(employee.getEmployeeID(), item, 20.0);
        assertEquals(30.0, checkout.getTotalPrice(), 0.001); // This
assertion is expected to fail
    }
}

```

### ShippingInfoTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class ShippingInfoTest {
    @Test
    void testSetGetEmployee() {
        ShippingInfo shippingInfo = new ShippingInfo();
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        shippingInfo.setEmployee(employee);
        assertEquals(employee, shippingInfo.getEmployee());
    }
}

```

### PaymentSystemTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import java.util.Date;

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

public class PaymentSystemTest {
    @Test
    void testSetGetCheckout() {
        PaymentSystem paymentSystem = new PaymentSystem();
        Checkout checkout = new Checkout(100.0, true);
        paymentSystem.setCheckout(checkout);
        assertEquals(checkout, paymentSystem.getCheckout());
    }

    @Test
    void testSetGetBeenzCredit() {
        PaymentSystem paymentSystem = new PaymentSystem();
        BeenzCredit beenzCredit = new BeenzCredit();
        paymentSystem.setBeenzCredit(beenzCredit);
        assertEquals(beenzCredit, paymentSystem.getBeenzCredit());
    }

    @Test
    void testSetGetEmployee() {
        PaymentSystem paymentSystem = new PaymentSystem();
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
    }
}

```

```

        paymentSystem.setEmployee(employee);
        assertEquals(employee, paymentSystem.getEmployee());
    }

    @Test
    void testFinishAndPay() {
        PaymentSystem paymentSystem = new PaymentSystem();
        Checkout checkout = new Checkout(100.0, true);
        Cart cart = new Cart(1);
        Item item = new Item("item1", "Item1", 10.0, true);
        cart.addItem(item);
        checkout.setCart(cart);
        paymentSystem.setCheckout(checkout);
        paymentSystem.finishAndPay(item, "id1", new Date());
        assertTrue(checkout.isComplete());
        assertEquals(0,
            cart.getItems().stream().mapToInt(Item::getQuantity).sum()); // Check if
            cart is empty
    }
    //the following test is expected to fail
    @Test
    void testFinishAndPayFail() {
        PaymentSystem paymentSystem = new PaymentSystem();
        Checkout checkout = new Checkout(100.0, true);
        Cart cart = new Cart(1);
        Item item = new Item("item1", "Item1", 10.0, true);
        cart.addItem(item);
        checkout.setCart(cart);
        paymentSystem.setCheckout(checkout);
        paymentSystem.finishAndPay(item, "id1", new Date());
        assertFalse(checkout.isComplete());
        assertEquals(0,
            cart.getItems().stream().mapToInt(Item::getQuantity).sum()); // Check if
            cart is empty
    }
}

```

## ItemTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class ItemTest {
    @Test
    void testConstructorAndGetters() {
        Item item = new Item("id1", "Item1", 20.99, true);
        assertEquals("id1", item.getItemID());
        assertEquals("Item1", item.getName());
        assertEquals(20.99, item.getPrice(), 0.001);
        assertTrue(item.isAvailable());
    }

    @Test
    void testSetGetEmployee() {
        Employee employee = new Employee("id1", "Emanuele",
            "test@example.com");
        Item item = new Item("id1", "Item1", 20.99, true);
    }
}

```

```

        item.setEmployee(employee);
        assertEquals(employee, item.getEmployee());
    }

    @Test
    void testSetGetCart() {
        Cart cart = new Cart(5);
        Item item = new Item("id1", "Item1", 20.99, true);
        item.setCart(cart);
        assertEquals(cart, item.getCart());
    }
}

```

## EmployeeTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class EmployeeTest {
    //the following test is expected to fail
    @Test
    public void testBeenzBalanceFail() {
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        employee.setBeenzBalance(100.01);
        assertEquals(100, employee.getBeenzBalance(), 0.001); // Delta is
used for comparing doubles
    }
    @Test
    public void testBeenzBalance() {
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        employee.setBeenzBalance(100.00);
        assertEquals(100, employee.getBeenzBalance(), 0.001); // Delta is
used for comparing doubles
    }
    @Test
    void testNegativeBeenzBalance() {
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        employee.setBeenzBalance(100);
        assertTrue(employee.getBeenzBalance() >= 0);
    }
    @Test
    void testEmailSetterGetter() {
        Employee employee = new Employee("id2", "Emanuele",
"test@example.com");
        employee.setEmail("newemail@example.com");
        assertEquals("newemail@example.com", employee.getEmail());
    }
    @Test
    void testIdSetterGetter() {
        Employee employee = new Employee("id3", "Emanuele",
"test@example.com");
        employee.setEmployeeID("newId3");
        assertEquals("newId3", employee.getEmployeeID());
    }
}

```

```

//the following test is expected to fail
@Test
void testIdSetterGetterFail() {
    Employee employee = new Employee("id3", "Emanuele",
"test@example.com");
    employee.setEmployeeID("wrongID");
    assertEquals("newId3", employee.getEmployeeID());
}
@Test
void testNameSetterGetter() {
    Employee employee = new Employee("id4", "Emanuele",
"test@example.com");
    employee.setName("Nunzio");
    assertEquals("Nunzio", employee.getName());
}
@Test
void testCartSetterGetter() {
    Employee employee = new Employee("id5", "Emanuele",
"test@example.com");
    Cart cart = new Cart(2);
    employee.setCart(cart);
    assertEquals(cart, employee.getCart());
}
@Test
void testItemSetterGetter() {
    Employee employee = new Employee("id6", "Emanuele",
"test@example.com");
    Item item = new Item("item1", "Item1", 10.99, true);
    employee.setItem(item);
    assertEquals(item, employee.getItem());
}
@Test
void testShippingInfoSetterGetter() {
    Employee employee = new Employee("id7", "Emanuele",
"test@example.com");
    ShippingInfo shippingInfo = new ShippingInfo();
    employee.setShippingInfo(shippingInfo);
    assertEquals(shippingInfo, employee.getShippingInfo());
}
@Test
void testBeenzCreditSetterGetter() {
    Employee employee = new Employee("id8", "Emanuele",
"test@example.com");
    BeenzCredit beenzCredit = new BeenzCredit();
    employee.setBeenzCredit(beenzCredit);
    assertEquals(beenzCredit, employee.getBeenzCredit());
}
}

```

### CheckoutTest

```

package com.ase_assignment;

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

class CheckoutTest {

    @Test
    void testSetGetCart() {
        Checkout checkout = new Checkout(100.0, false);
    }
}

```

```

        Cart cart = new Cart(5);
        checkout.setCart(cart);
        assertEquals(cart, checkout.getCart());
    }

    @Test
    void testSetGetShippingInfo() {
        Checkout checkout = new Checkout(100.0, false);
        ShippingInfo shippingInfo = new ShippingInfo();
        checkout.setShippingInfo(shippingInfo);
        assertEquals(shippingInfo, checkout.getShippingInfo());
    }

    @Test
    void testSetGetBeenzCredit() {
        Checkout checkout = new Checkout(100.0, false);
        BeenzCredit beenzCredit = new BeenzCredit();
        checkout.setBeenzCredit(beenzCredit);
        assertEquals(beenzCredit, checkout.getBeenzCredit());
    }

    @Test
    void testSetGetPaymentSystem() {
        Checkout checkout = new Checkout(100.0, false);
        PaymentSystem paymentSystem = new PaymentSystem();
        checkout.setPaymentSystem(paymentSystem);
        assertEquals(paymentSystem, checkout.getPaymentSystem());
    }

    @Test
    void testSetGetEmployee() {
        Checkout checkout = new Checkout(100.0, false);
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        checkout.setEmployee(employee);
        assertEquals(employee, checkout.getEmployee());
    }

    @Test
    void testApplyBeenzCredits() {
        Checkout checkout = new Checkout(100.0, false);
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        checkout.setEmployee(employee);
        employee.setBeenzBalance(50);
        checkout.applyBeenzCredits(30);
        assertEquals(20, employee.getBeenzBalance(), 0.001);
        assertEquals(70, checkout.getTotalPrice(), 0.001);
    }
    //the following test is expected to fail
    @Test
    void testApplyBeenzCreditsFail() {
        Checkout checkout = new Checkout(100.0, false);
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        checkout.setEmployee(employee);
        employee.setBeenzBalance(50);
        checkout.applyBeenzCredits(30);
        assertEquals(20, employee.getBeenzBalance(), 0.001);
        assertEquals(70, checkout.getTotalPrice(), 0.001);
    }
}

```

```

@Test
void testConfirmOrder() {
    Checkout checkout = new Checkout(100.0, true);
    checkout.confirmOrder();
    assertTrue(checkout.isComplete());
}

@Test
void testConfirmOrderFail() {
    Checkout checkout = new Checkout(100.0, true);
    checkout.confirmOrder();
    assertFalse(checkout.isComplete());
}

@Test
void testCompleteCheckout() {
    Checkout checkout = new Checkout(100.0, true);
    checkout.setIsComplete(true);
    checkout.completeCheckout();
    assertTrue(checkout.isComplete());
}
}

```

## CartTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import java.util.List;

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

public class CartTest {
    @Test
    void testSetGetEmployee() {
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        Cart cart = new Cart(5);
        cart.setEmployee(employee);
        assertEquals(employee, cart.getEmployee());
    }

    @Test
    void testAddItem() {
        Cart cart = new Cart(5);
        Item item = new Item("item1", "Item1", 10.99, true);
        cart.addItem(item);
        assertTrue(cart.getItems().contains(item));
        assertEquals(cart, item.getCart()); // Verifying the back-reference
from item to cart
    }

    @Test
    void testAddItemFail() {
        Cart cart = new Cart(5);
        Item item = new Item("item1", "Item1", 10.99, true);
        cart.addItem(item);
        assertFalse(cart.getItems().contains(item));
        assertEquals(cart, item.getCart()); // Verifying the back-reference

```

```

from item to cart
    }

    @Test
    void testSetGetItems() {
        Cart cart = new Cart(5);
        Item item1 = new Item("item1", "Item1", 10.99, true);
        Item item2 = new Item("item2", "Item2", 20.99, true);
        List<Item> items = List.of(item1, item2);
        cart.setItems(items);
        assertEquals(items, cart.getItems());
    }

    @Test
    void testSetGetCheckout() {
        Cart cart = new Cart(8);
        Checkout checkout = new Checkout(100.99, true);
        cart.setCheckout(checkout);
        assertEquals(checkout, cart.getCheckout());
    }
}

```

### BeenzCreditTest

```

package com.ase_assignment;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class BeenzCreditTest {
    @Test
    void testSetGetEmployee() {
        Employee employee = new Employee("id1", "Emanuele",
"test@example.com");
        BeenzCredit beenzCredit = new BeenzCredit();
        beenzCredit.setEmployee(employee);
        assertEquals(employee, beenzCredit.getEmployee());
    }

    @Test
    void testSetGetCheckout() {
        Checkout checkout = new Checkout(100.99, true);
        BeenzCredit beenzCredit = new BeenzCredit();
        beenzCredit.setCheckout(checkout);
        assertEquals(checkout, beenzCredit.getCheckout());
    }

    @Test
    void testSetGetPaymentSystem() {
        PaymentSystem paymentSystem = new PaymentSystem();
        BeenzCredit beenzCredit = new BeenzCredit();
        beenzCredit.setPaymentSystem(paymentSystem);
        assertEquals(paymentSystem, beenzCredit.getPaymentSystem());
    }
}

```

### CommandInvokerTest

```

package com.ase_assignment.invokersTest;

import com.ase_assignment.Employee;
import com.ase_assignment.Item;

```

```

import com.ase_assignment.StoreManagementSystemComp;
import com.ase_assignment.commands.AddItemToCartCommand;
import com.ase_assignment.commands.Command;
import com.ase_assignment.invokers.CommandInvoker;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;

import static org.mockito.Mockito.*;

public class CommandInvokerTest {
    private CommandInvoker invoker;

    @Mock
    private Command mockCommand;

    @BeforeEach
    void setUp() {
        StoreManagementSystemComp system = new StoreManagementSystemComp();
        Employee employee = new Employee("id", "Test User",
"test@email.com");
        system.addEmployee(employee);
        Item item = new Item("item1", "Test Item", 10.0, true);
        Command addItemCommand = new AddItemToCartCommand(system,
employee.getEmployeeID(), item);
        CommandInvoker invoker = new CommandInvoker();
        invoker.setCommand(addItemCommand);
    }

    @Test
    void testExecuteCommand() {
        invoker.setCommand(mockCommand);
        invoker.executeCommand();
        verify(mockCommand, times(1)).execute();
    }

    @Test
    void testExecuteCommandWithNoCommandSet() {
        invoker.executeCommand();
        verify(mockCommand, never()).execute();
    }
}

```

### AddItemToCartCommandTest

```

package com.ase_assignment.commandsTest;

import com.ase_assignment.*;
import com.ase_assignment.commands.AddItemToCartCommand;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue;

class AddItemToCartCommandTest {
    private StoreManagementSystemComp system;
    private Employee employee;
    private Item item;
    private AddItemToCartCommand command;
}

```

```

@BeforeEach
void setUp() {
    system = new StoreManagementSystemComp();
    employee = new Employee("id1", "Emanuele", "test@example.com");
    item = new Item("item1", "Item1", 20.99, true);
    employee.setCart(new Cart(0));
    system.addEmployee(employee);
    command = new AddItemToCartCommand(system,
employee.getEmployeeID(), item);
}

@Test
void execute() {
    command.execute();
    assertTrue(employee.getCart().getItems().contains(item));
}
}

```

### FinishAndPayCommandTest

```

package com.ase_assignment.commandsTest;

import com.ase_assignment.*;
import com.ase_assignment.commands.FinishAndPayCommand;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Date;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

class FinishAndPayCommandTest {
    private PaymentSystem paymentSystem;
    private Employee employee;
    private Cart cart;
    private Item item;
    private FinishAndPayCommand command;
    private Date date;

    @BeforeEach
    void setUp() {
        paymentSystem = new PaymentSystem();
        employee = new Employee("id1", "John", "john@example.com");
        cart = new Cart(1);
        item = new Item("item1", "Item1", 20.0, true);
        date = new Date();
        employee.setCart(cart);
        cart.addItem(item);
        paymentSystem.setCheckout(new Checkout(20.0, true));
        paymentSystem.getCheckout().setCart(cart);
        command = new FinishAndPayCommand(paymentSystem, item,
employee.getEmployeeID(), date);
    }

    @Test
    void execute() {
        command.execute();
        assertTrue(paymentSystem.getCheckout().isComplete());
        assertEquals(1,

```

```
paymentSystem.getCheckout().getCart().getItems().size());  
    }  
}
```

### InitiateCheckoutCommandTest

```
package com.ase_assignment.commandsTest;  
  
import com.ase_assignment.*;  
import com.ase_assignment.commands.InitiateCheckoutCommand;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;  
  
class InitiateCheckoutCommandTest {  
    private StoreManagementSystemComp storeManagementSystem;  
    private Employee employee;  
    private Item item;  
    private InitiateCheckoutCommand command;  
    private Double totalPrice;  
  
    @BeforeEach  
    void setUp() {  
        storeManagementSystem = new StoreManagementSystemComp();  
        employee = new Employee("id1", "John", "john@example.com");  
        item = new Item("item1", "Item1", 20.0, true);  
        totalPrice = 20.0;  
        storeManagementSystem.addEmployee(employee);  
        employee.setCart(new Cart(1));  
        command = new InitiateCheckoutCommand(storeManagementSystem,  
employee.getEmployeeID(), item, totalPrice);  
    }  
  
    @Test  
    void execute() {  
        command.execute();  
        assertNotNull(employee.getCart().getItems());  
    }  
}
```