

The CoCoME Platform for Collaborative Empirical Research on Information System Evolution

Evolutionscenario in the second founding period of SPP 1593

Robert Heinrich, Niko Benkler, Tobias Haßberg, Ralf Reussner

June 3, 2018

Contents

1	Introduction	4
2	Evolution Scenarios	5
2.1	Evolution Scenarios of the Hybrid Cloud-based Variant	5
2.1.1	Setting up a Docker environment	5
2.1.2	Adding a Mobile App	5
2.2	Evolution Scenarios of the Microservice-based Variant	6
2.2.1	Defining different Microservices	6
3	Design Details for Evolution Scenarios	7
3.1	Adding a Mobile App Client	7
3.1.1	Use Cases of the Mobile App	7
3.1.2	Design of the Mobile App	11
3.2	Using a Docker Environment	13
3.3	Microservices Technology	14
3.3.1	Usecase implementation	14
3.3.2	Architectural overview	23
4	Implementation of Evolution Scenarios	24
4.1	Using a Docker Environment	24
4.2	Adding a Mobile App Client	26
4.3	Using Microservice Technology	28
5	Conclusion	29

List of Figures

3.1	Use Case Diagram CoCoME Mobile App	8
3.2	Component Diagram of the CoCoME Ecosystem After Adding the Mobile App Client	11
3.3	Sequence Diagram of Searching an Item in the Mobile App Client	12
3.4	Sequence Diagram of Processing a Sale	13
3.5	Extended technology stack CoCoME	14
3.6	Usecase 3 order products, part 1	15
3.7	Usecase 3 order products, part 2	16
3.8	Usecase 1 process sale, part 1	18
3.9	Usecase 1 process sale, part 2	19
3.10	Usecase 2 manage express mode	20
3.11	Usecase 7 change price	20
3.12	Usecase 8 product exchange	21
3.13	Usecase 5 show stock report	22
3.14	Usecase 6 show delivery report	23
4.2	Assignment of archive files to Servers	24
4.4	Assignment archive files to Servers	24
4.1	Deployment diagram CoCoME	25
4.3	Deployment diagram CoCoME Pickup Shop	26
4.5	Primary Classes of the App	27

1 Introduction

2 Evolution Scenarios

We implemented distinct evolution scenarios covering the categories adaptive and perfective evolution. Corrective evolution is not considered in the scenarios as this merely refers to fixing design or implementation issues.

2.1 Evolution Scenarios of the Hybrid Cloud-based Variant

This section introduces the two evolution scenarios of the hybrid cloud-based variant of CoCoME.

2.1.1 Setting up a Docker environment

The CoCoME company must reduce IT administration costs but frequent updates to the enterprise and store software are necessary to continuously improve the entire system. As a consequence, IT staff need to update the system components as soon as a new software version is released. An Operations Team member has to get access to the actual server in order to undeploy the old version and replace it with the new one. This is time consuming and expensive as the updates have to be done manually.

Therefore, a Docker version is elaborated to simplify the administration process. As soon as a new software version of CoCoME is ready for delivery, the Development Team wrap it into a Docker Image. This Image can be automatically deployed to the destination server according to the principle of Continuous Deployment (CD) [2].

2.1.2 Adding a Mobile App

After successfully adding a Pick-up Shop, the CoCoME company stays competitive with other online shop vendors (such as Amazon). In times of smartphones, customer do not only want to buy exclusively goods from their home computers. Purchasing goods 'on the way' comes more and more into fashion. This raises the idea to create a second sales channel next to the existing Pick-up Shop in the CoCoME system. As a consequence, more customers can be attracted to gain a larger share of the market.

The customer can order and pay by using the app. The delivery process is similar to the Pick-up Shop: The goods are delivered to a pick-up place (i.e. a store) of her/his choice, for example in the neighbourhood or the way to work. By introducing the Mobile App as a multi OS

application, the CoCoME system has to face various quality issues such as privacy, security and reliability. Also the performance of the whole application can be affected if many customers order via the app.

2.2 Evolution Scenarios of the Microservice-based Variant

This section introduces the evolution scenario of the Microservice-based variant of CoCoME.

2.2.1 Defining different Microservices

After a year of economical stagnation, the CoCoME company decides to restructure its infrastructure. Global players like Amazon or Netflix demonstrated that using a Microservice Architecture makes them more flexible regarding new functionality. When adding the Pick-up Shop, the CoCoME company realized that they have to break open the existing system. It was necessary to modify the *WebService::Inventory* and the *TradingSystem::Inventory* component [1]. Furthermore, adding a *MobileAppClient* demonstrated that the SOAP/WS*-based web services provided by CoCoME are not compatible with REST-based App development.

Inspired by the flexibility and reusability of Microservices, the CoCoME company decided to invest money into a restructuring process. The current system is divided into a collection of loosely coupled services. Each of them covers a specific part of the former CoCoME system. The aim is to preserve the functionality of the current system and solely change its architecture. This enables the company to develop new markets much easier and therefore secures the future competitiveness. For example, the CoCoME company wants to extend their product range by offering movie streaming. This requires a vastly different system. Nevertheless, Customer Management like login and means of payment are identical to the former CoCoME system. Those components already exists as a Microservice a therefore can be taken over. The management is certain that this will soon result in economical growth.

3 Design Details for Evolution Scenarios

In this chapter we provide the detailed design documentation for each of the evolution scenarios introduced in the prior section. Sec. 3.1 sketches the design decision for the Mobile App that provides a second sales channel next to the existing Pick-up Shop. Sec. 3.2 describes the adaptive changes of using a Docker environment to simplify the update process. They are both based on, or at least use the Hybrid Cloud-based Variant of CoCoME [1]. In contrast, Sec. 3.3 provides a detailed design documentation of a new architectural version of CoCoME. This perspective evolution scenario is realized based on the Microservice idea.

3.1 Adding a Mobile App Client

Developing the Mobile App Client as an extension of CoCoME requires additional use cases. They are described in Sec. 3.1.1. Sec. 3.1.2 describes extensions on design level. The content of this chapter mainly originates from [3].

3.1.1 Use Cases of the Mobile App

UC 14 - ProcessAppSale

Brief Description A Customer selects the product items s/he wants to buy and the payment by credit card is performed.

Involved Actors AppCustomer, Bank

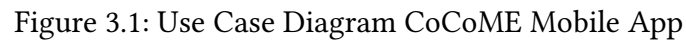
Precondition The App is ready to process a new sale and the Customer already has an account registered in the System.

Trigger The Customer opens the app and wants to buy product items.

Postcondition The Customer has paid and the sale is registered in the inventory.

Standard Process

1. The AppCustomer searches products provided by the App.
2. The AppCustomer can see details for each product on a separate site.
3. The AppCustomer adds the product items s/he wants to purchase to the Shopping Cart. Step 1-3 is repeated until all items are added to the cart.



- ### Alternative or Exceptional Processes

- In step 8: No Card available

1. In order to add a new credit card the Customer clicks the Add Card button.
 2. The Customer enters the card number of the new credit card and saves the card.
- In step 10: Card validation fails
1. The Customer tries again and again.
 2. Otherwise, the Customer can decide to cancel the purchase.

UC 15 - CreateAppCustomer

Brief Description The app offers a possibility to create a new Customer account.

Involved Actors AppCustomer

Precondition The Customer does not have a Customer account yet and the app is started.

Trigger A new AppCustomer wants to create an account.

Postcondition The User is authenticated.

Standard Process

1. The AppCustomer has to fill out forms, requesting all necessary information to create a new AppCustomer account.
 - (a) Form for name, email and password
 - (b) Form for address
 - (c) Summary of the information
2. The Customer fills out the forms, verifies and submits the information.
3. The app verifies the given information and creates a new Customer account in the Inventory.

Alternative or Exceptional Processes

- In step 3 : Provided information is incorrect or not valid. The Customer is notified of the problem and enters the information again until it passes the check.

UC 16 - AuthenticateAppUser

Brief Description The app provides the possibility to authenticate a User.

Involved Actors AppCustomer

Precondition The app is started.

Trigger An AppCustomer wants to authenticate his/herself.

Postcondition The AppCustomer is authenticated.

Standard Process

1. The AppCustomer gets displayed a login form. S/he is asked to enter email and password.
2. The App checks the provided credentials. If correct, the AppCustomer is logged in.

Alternative or Exceptional Processes

- In step 2: Wrong credentials
 1. An error message is displayed.
 2. The User may try again until the authentication succeeds.

3.1.2 Design of the Mobile App

Fig. 3.2 is the component of this evolution scenario. When adding the Mobile App client, the hybrid cloud-based variant of CoCoME did not have to be modified. Therefore, CoCoME is encapsulated in a single component. Simply the three web services *WebService::Inventory::LoginManager*, *WebService::Inventory::Store* and *WebService::Inventory::Enterprise* used by the App Client are emphasized. The entire component diagram for the hybrid cloud-based variant is available in the Technical Report [1].

Fig. 3.2 indicates that the AppShop requires an adapter to access the web services provided by CoCoME. This is because CoCoME uses SOAP/WS*-based web services which are not compatible with the technology used to implement the AppShop Client. A more detailed introduction about the technology used to implement the Mobile App Client can be found in [3].

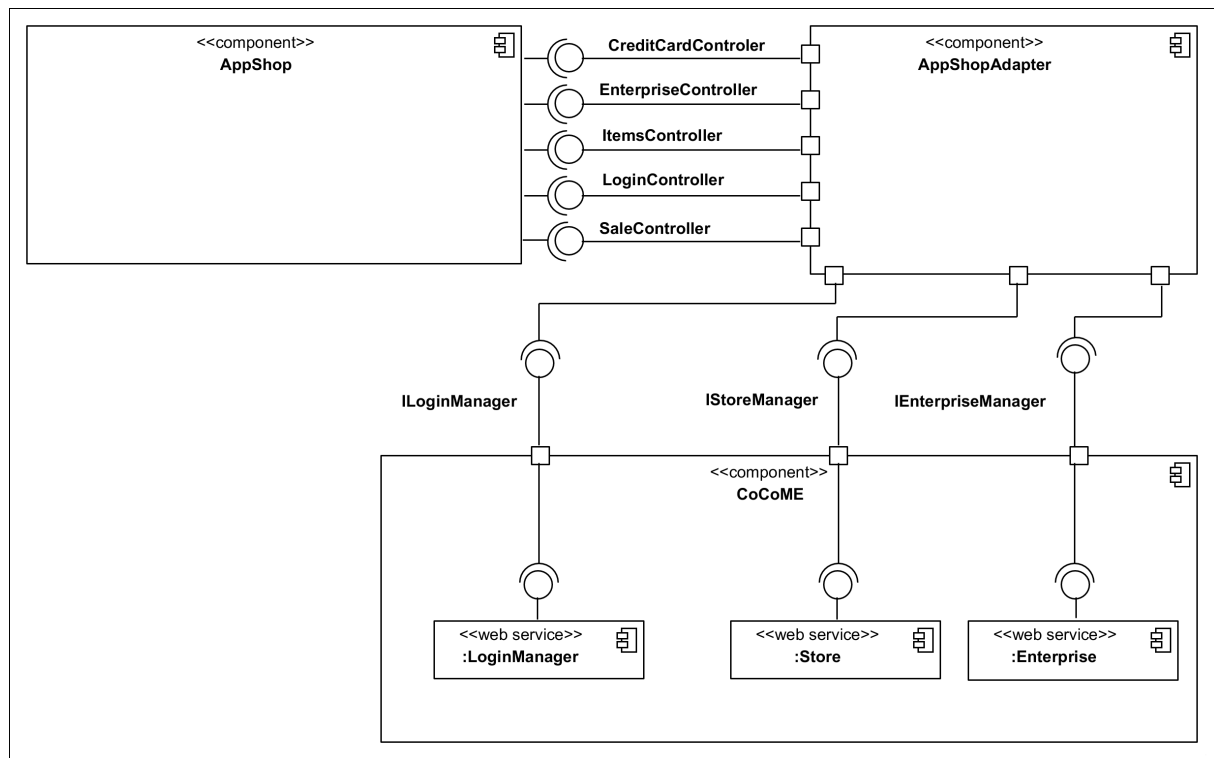


Figure 3.2: Component Diagram of the CoCoME Ecosystem After Adding the Mobile App Client

The *AppShopAdapter* consumes the three web services *WebService::Inventory::LoginManager*, *WebService::Inventory::Store* and *WebService::Inventory::Enterprise* and provides a Rest Api which is used by the actual *AppShop*. The Rest Api contains endpoints to retrieve and process Credit Card, Enterprise and StockItem information. To implement UC14-16, the Api also provides endpoints for user management and processing sales.

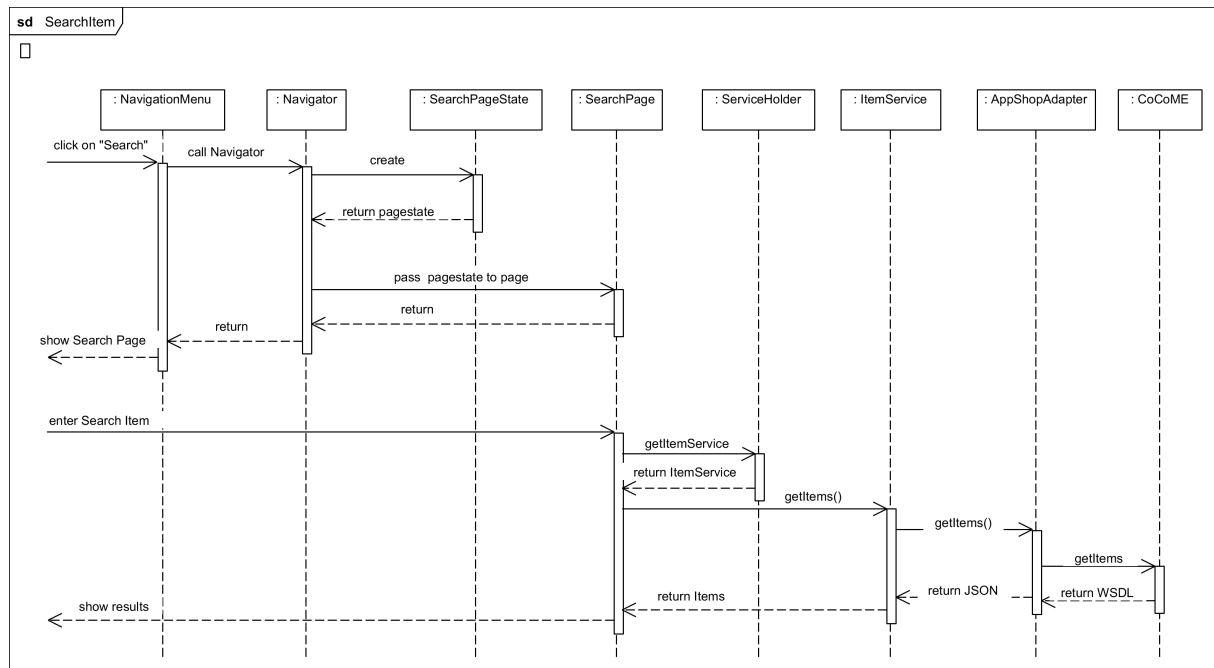


Figure 3.3: Sequence Diagram of Searching an Item in the Mobile App Client

Fig. 3.3 shows the process of opening a page to search for an Item. The customer opens the *WebShopClient* and triggers the "Search" function to search for an item. To open the page, the *NavigatorMenu* must call the *Navigator* which creates a pagestate object and passes the object to the page. This HTML page is now presented to the customer. To fill the page with information, i.e. when searching for a *ProductItem*, the page uses services provided by the *ServiceHolder*. In this case, the *ItemService* calls the responsible REST-Service of *AppShopAdapter* which in turn retrieves the necessary information from the WSDL services provided by *CoCoME*.

Fig. 3.4 demonstrates how the Mobile App Client processes sales. For the sake of clarity, the diagram is simplified and only contains the most important calls. First, the customer searches for items (according to Fig. 3.3). By clicking on the desired Item, the according *ItemPage* is shown. This page carries information about the Item. Here, the customer decides whether the Item should be added to the Shopping Cart or not. The last steps are repeated until the customer decides to proceed to the checkout. If not logged in, the customer gets forwarded to the *LoginPage*. When successfully logged in, the customer clicks the *BuyNow*-Button. The Sale process is finished as soon as the backend (*CoCoME*) has processed the sale.

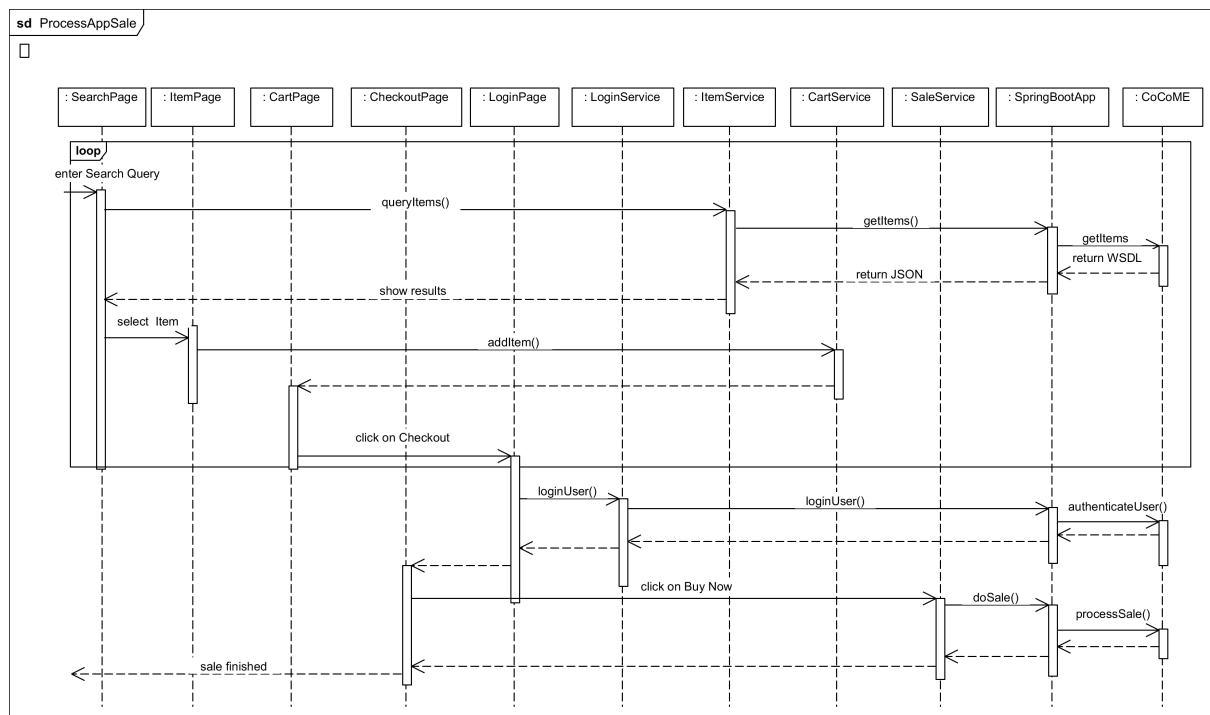


Figure 3.4: Sequence Diagram of Processing a Sale

3.2 Using a Docker Environment

As shown in Fig. 3.5, using a Docker Environment affects the technology stack by adding additional layers. More detailed, the given CoCoME Stack is moved into the Docker Deamon, which runs a Linux distribution. The original parts of the stack, like Glassfish and the Java Virtual Machine, are still a part of the stack.

The Dockerfile defines an environment based on the latest version of Ubuntu 16:04. Maven, Git and Java are also installed using the standard Ubuntu package manager.

Git has two purposes: On the one hand it is used to download the most recent version of CoCoME. On the other hand, it is used to download a prefabricated version of Glassfish that already includes domains and other adjustments required for CoCoME. Java is required by Glassfish and CoCoME as they need the Java Virtual Machine. Maven is needed to deploy the latest version of CoCoME onto the provided Glassfish servers.

During the development, it was decided to implement and provide two different versions. The first version always pulls the most recent CoCoME source code from GitHub, downloads the entire dependencies with Maven, compiles and builds the project and finally deploys CoCoME on the Glassfish servers. As a consequence, creating and starting a Docker Container takes about one hour.

In contrast, the second version only pulls a prefabricated version of CoCoME from GitHub. Therefore, pulling the source code up to building the project is skipped. Maven does not have

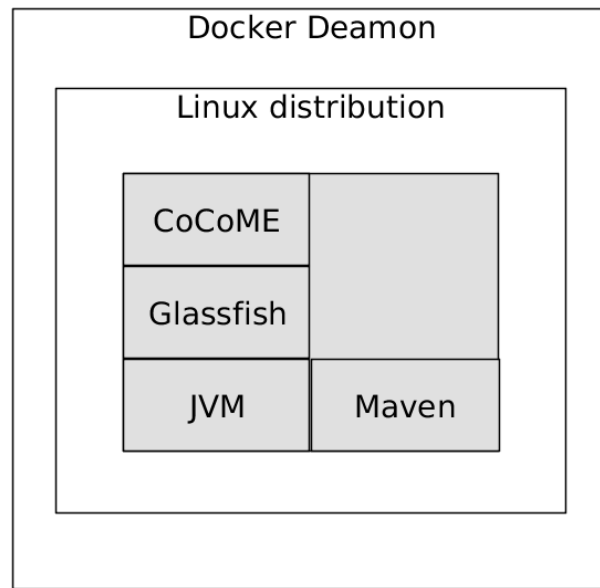


Figure 3.5: Extended technology stack CoCoME

to be included in the technology stack. Solely, deploying CoCoME on the Glassfish server is necessary.

This reduces the deployment time to a few minutes but has a disadvantage: The prefabricated version is updated manually. Therefore, it is sometime not the most recent version.

By providing both, a fast deploying version and a current version, the user can choose what's the best for its situation.

3.3 Microservices Technology

This section describes how the implementation of the usecases which are also implemented in the basic CoCoME version but also the general system design and the interaction of the microservices. In the following subsection [Usecase implementation](#) the realization details of different usecases is explained in detail. Further, the different usecases are allocated in the of the corresponding microservice. In the second subsection, named [Architectural overview](#) the general architectural overview of the microservices is described.

3.3.1 Usecase implementation

3.3.1.1 Orders

This section describes the design of the *Orders* microservice, which provides the main functionality for usecase 3 and usecase 4.

Behavioral View on UC 3 - Order Products

As shown in figure 3.6 and figure 3.7 this usecase is divided into two steps in the first step, user the frontend to create new OrderEntry elements. This elements are representing the different product entries at an order. It contains the informations about an the product to order and the ordering amount. They are saved in the Database until they are needed again in the second step again. In the second step, the ProductOrder element is created. This Object represents the order itself, in detail information about the ordering store, the date on which this order is submitted and a collection with the OrderEntry objects of the products which are ordered in this order. Also it set in the OrderEntry objects an reference to this ProductOrder object.

Behavioral View on UC 4 - Receive Ordered Products

The figure shows, the microservice first refreshes the store ProductOrder element and sets it delivery date to the in the call passed date. After this, it performs for each in the assigned a rest call to the store microservice in which the store itself is stored. With that rest call, the stockitems which are representing the corresponding product in the store are increasing their available amount of the product.

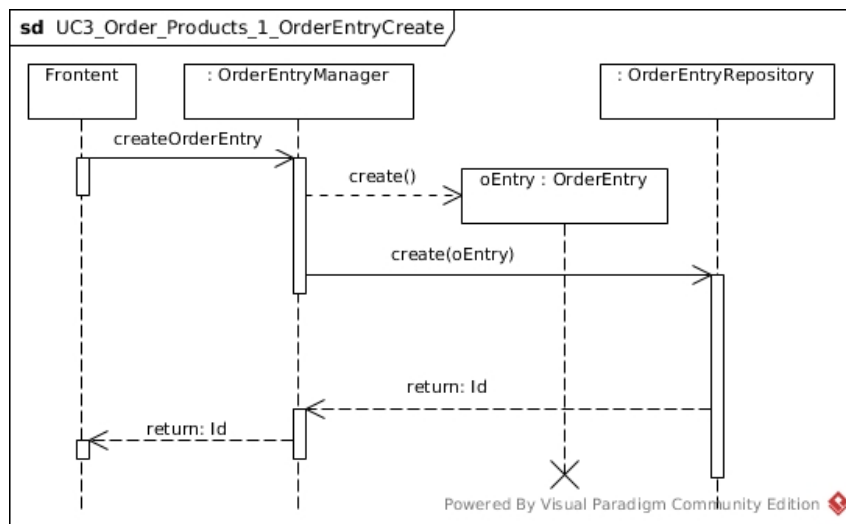


Figure 3.6: Usecase 3 order products, part 1

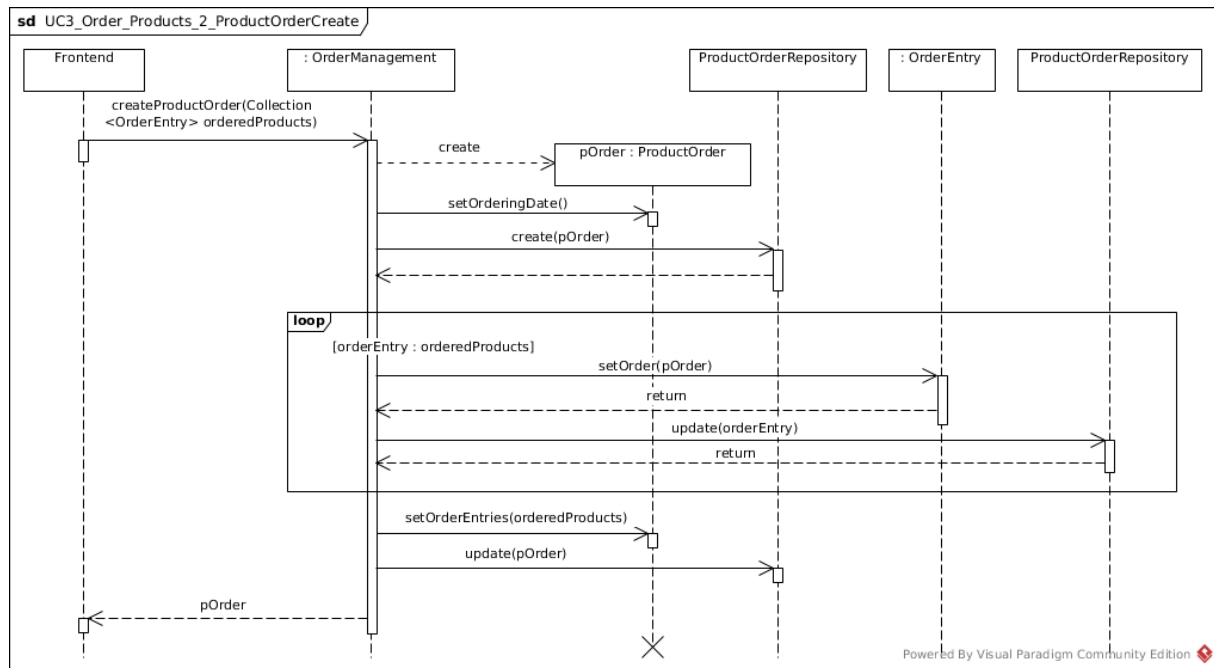


Figure 3.7: Usecase 3 order products, part 2

3.3.1.2 Stores

This section describes the design of the *Store* microservice, which provides the functionality of the usecases 1, 2, 7 and 8.

Behavioral View on UC 1 - Process Sale

The usecase 1 is pictured in figure 3.8 and 3.9. As the figures divide that usecase into 2 parts this explanation will do to explain the process properly. In the first step, shown in figure 3.8 the process of adding product to the sale process is shown. To add Products to a sale, at first the sale mode has to be activated in the cash desk, which is done after the startSale action is executed. This also resets the CashDesk from previous, probably not finished and canceled sale processes. For adding products to the sale process, either the barcode can be scanned and submitted as one or can entered digit by digit. This barcode entering is shown in the first if construct in the loop of figure 3.8. After scanning a barcode, it is checked if there is already an item was added to the sale and another one is available in stock, or if there is an item with that barcode in the stock available. In the first case, the the stock amount of that item is reduced and that item is added to the inventory and the total amount is increased. In the second case, mostly the same is done, but instead of increasing the product amount in the inventory, there is an new product added to that inventory. Otherwise, the attempt of adding an product with that barcode is quit. Subsequently, the display is updated and this Product is added to the printer output.

The second step of this usecase, shown in figure 3.9 handles the end of the sale process. The end

is indicated by calling the `finishSale` routine, which represents pressing the finish sale button at the cashbox. After pressing that button, the display updated and it can be chosen, between paying by card or paying by cash. In both cases, first the cash desk is set to the corresponding paying mode, second the needed informations about the credit card or the overhanded cash amount is passed through. After successfully ending the payment, the printer and display is updated and the cash desk ends the sale process.

Behavioral View on UC 2 - Manage Express Checkout

Figure 3.10 shows, how each cash desk performs on an external call the `updateExpressLight` routine. In this, the cash desk checks by itself if based on the recently proceeded the Express light should be activated or not. In consequence, the express light is set to the correct value.

Behavioral View on UC 7 - Change Price

As described in figure 3.11, in a first step, the database element which represents the store, managed by the store manager who wants to change the price is loaded. On this, the in the store available products are filtered to find the one with the correct product id. Having the correct `StockItem` found, the sale price of that object can be updated. Afterwards, the `StockItem` object is refreshed in the database.

Behavioral View on UC 8 - Product Exchange

The process on usecase 8 is shown in figure 3.12. It starts at the end of an sale process. The microservice checks by itself if the stock amount of the sold items have reached their minimal amount. If not, nothing will happen. Otherwise, this information is passed to the appropriate `EnterpriseManager`. This `EnterpriseManager` collects in an first step all of its stores, which are also selling this the specific product. If the `EnterpriseManager` has found at least one store which also provides that products, it figures out, which one of the possible exchange sources is the best. By reducing the stock amount of that product in this optimal store and increasing it in the store, which started the request, the product exchange is executed.

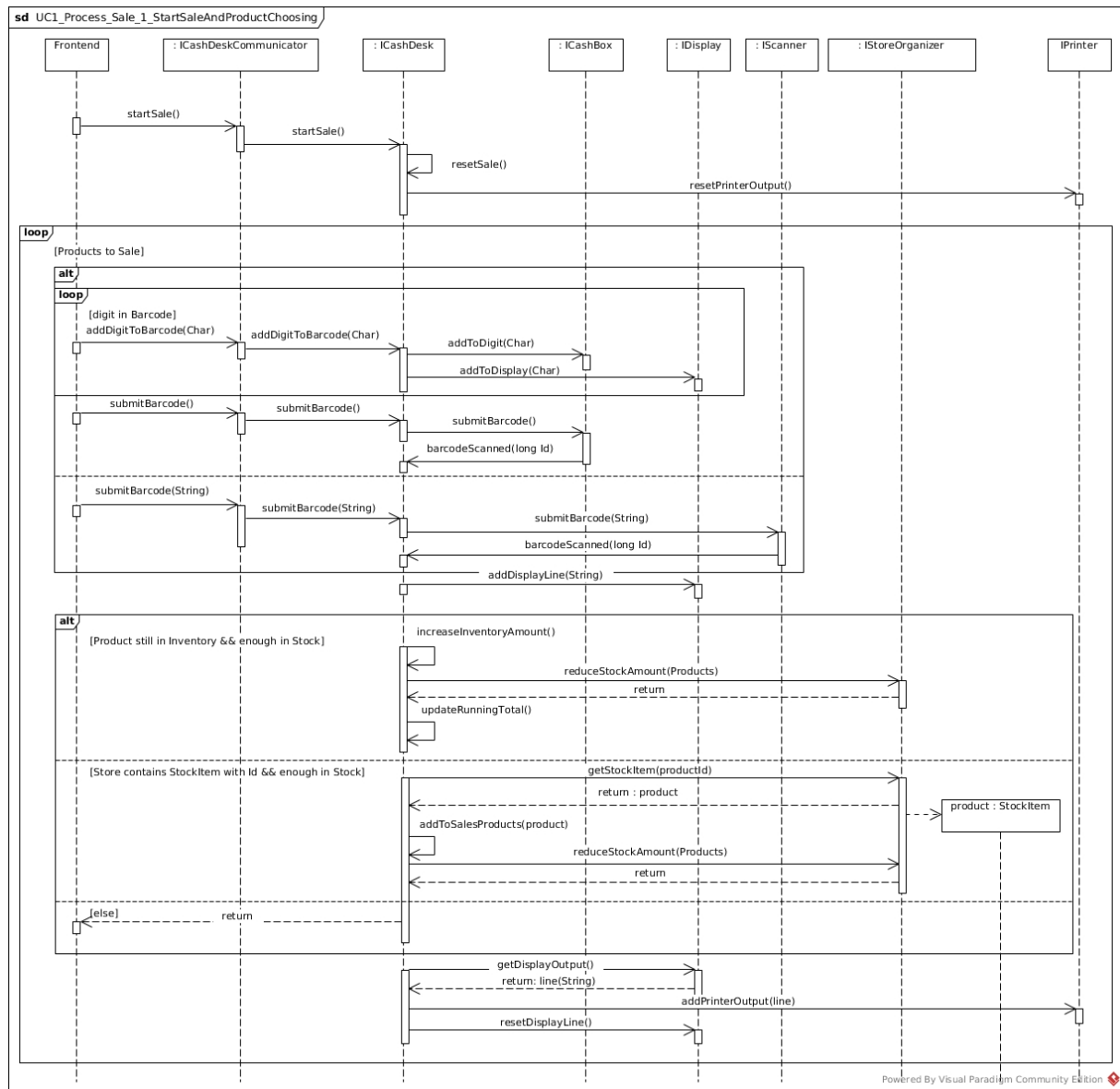


Figure 3.8: Usecase 1 process sale, part 1

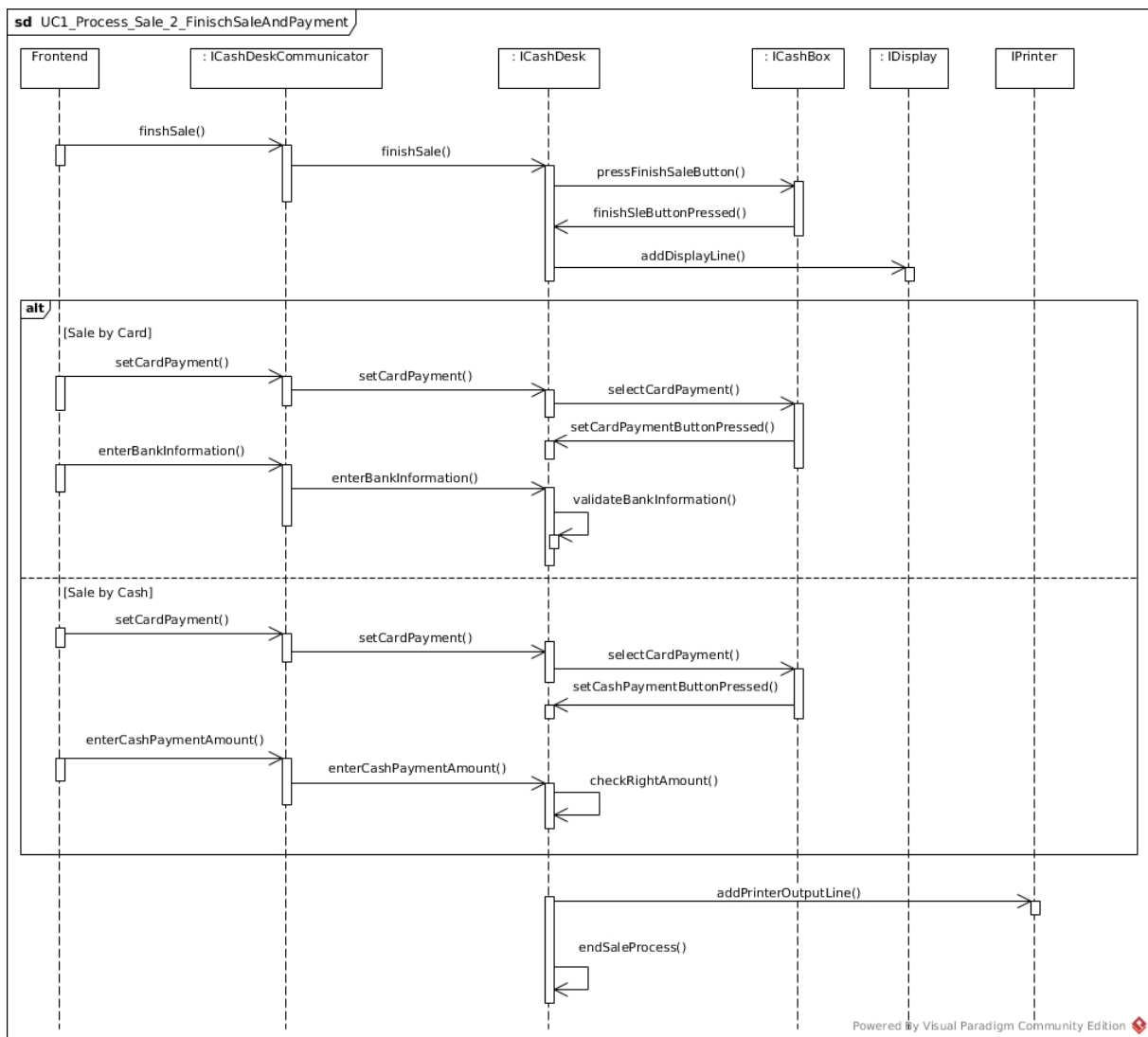


Figure 3.9: Usecase 1 process sale, part 2

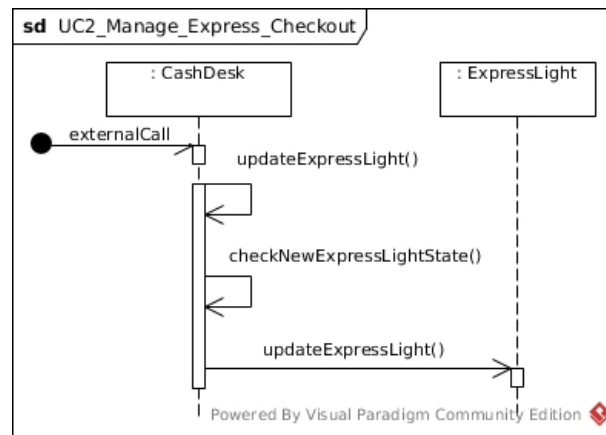


Figure 3.10: Usecase 2 manage express mode

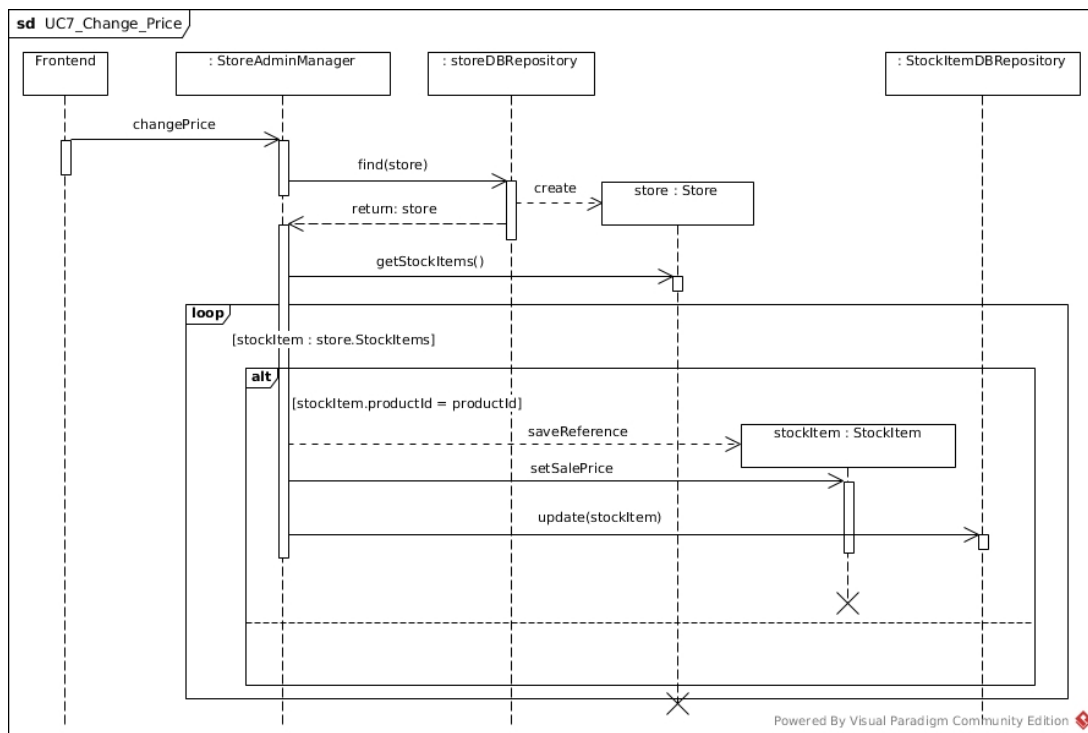


Figure 3.11: Usecase 7 change price

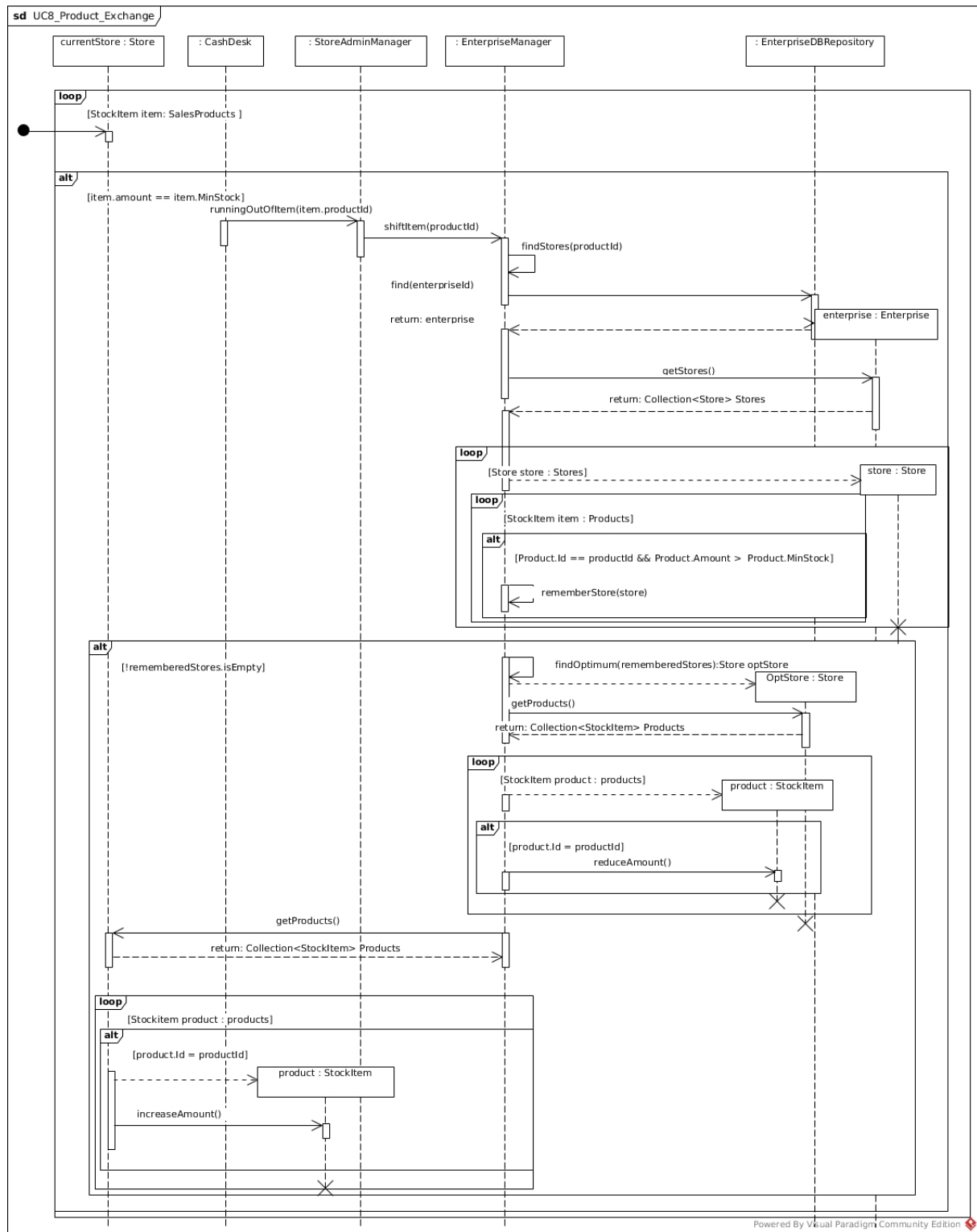


Figure 3.12: Usecase 8 product exchange

3.3.1.3 Reports

This section describes the design of the *Reports* microservice, which provides the functionality of the usecases 5 and 6

Behavioral View on UC 5 and Uc 6 - Show Stock Report and Show Delivery Report

In the following, the realisation of usecase 5 and 6 are described. Usecase 5 is described in figure 3.13 and usecase 6 in figure 3.14. In both cases, this microservice performs an rest call to receive informations from the the corresponding microservices which contains the requested informations. this informations are returned on that request and were modified to represent the information in an proper format. That format is passed back to the calling frontend.

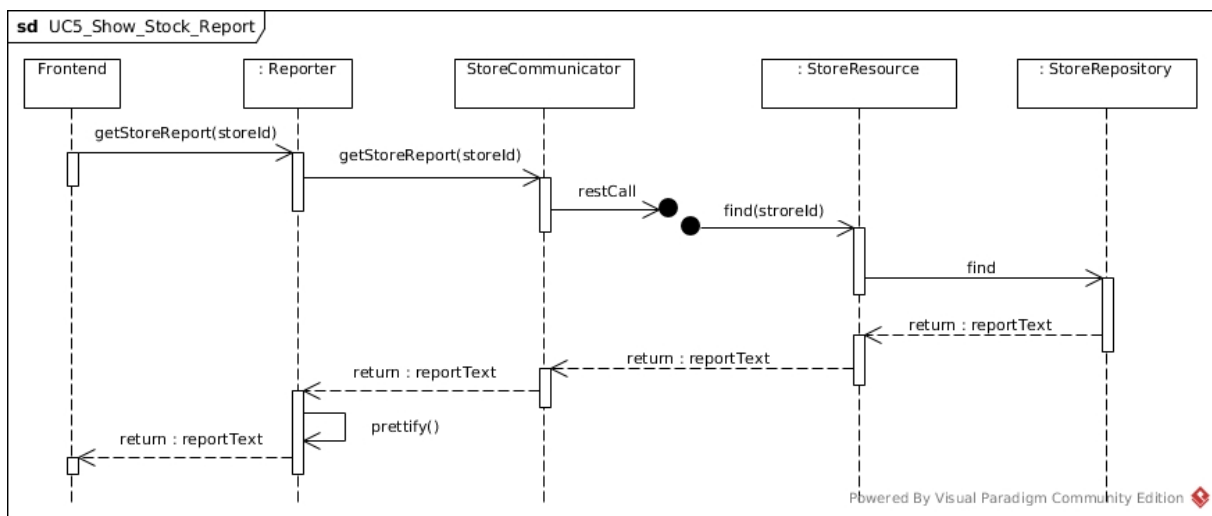


Figure 3.13: Usecase 5 show stock report

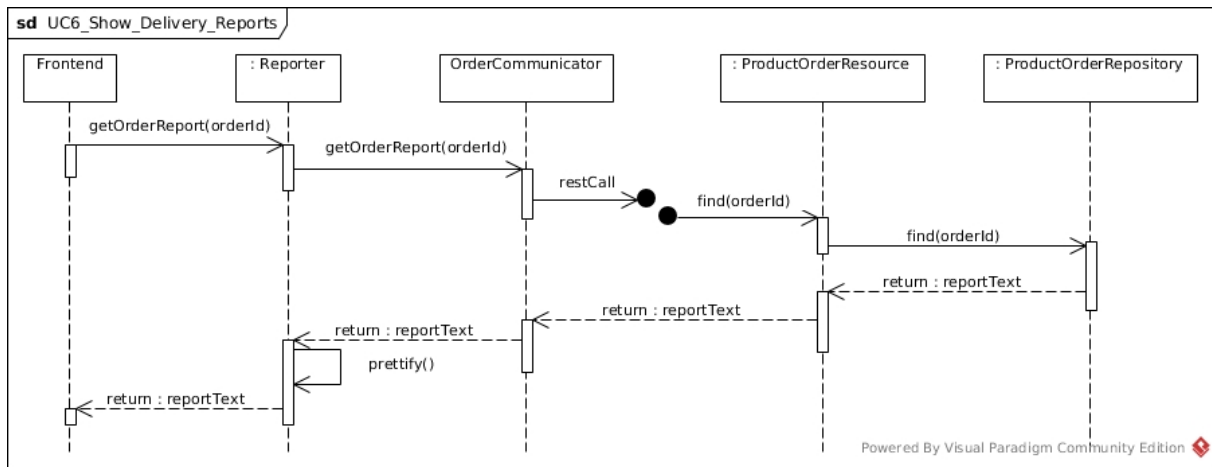


Figure 3.14: Usecase 6 show delivery report

3.3.2 Architectural overview

4 Implementation of Evolution Scenarios

This chapter describes implementation details for realizing the Docker environment 4.1, the Mobile App Client for the existing hybrid cloud-based variant of CoCoME (4.2) and the Microservice-based variant (4.3).

4.1 Using a Docker Environment

As shown in Fig. 4.1, the docker Container contains five different Glassfish servers. In particular they are called *WEB*, *ENTERPRISE*, *STORE*, *REGISTRY* and *ADAPTER*. By default, Glassfish provides a Derby DB that is connected to the Service Adapter using Java Database Conectivity (JDBS) interface.

The deployment assignment within the Docker environment is identical to the one specified in CoCoME deployment guide. This means the maven generated archive files *cloud-web-frontend*, *enterprise-logic-ear*, *store-logic-ear*, *cloud-registry-sevice* and *service-adapter-ear* are deployed on the servers by using the following assignment:

Server	Deployment file
WEB	cloud-web-frontend
ENTERPRISE	enterprise-logic-ear
STORE	store-logic-ear
REGISTRY	cloud-registry-service
ADAPTER	service-adapter-ear

Figure 4.2: Assignment of archive files to Servers

As mentioned earlier, there are two versions of this Docker project. The fast version can be extended by the Pick-Up shop¹. This Pick-Up shop runs inside a separate Docker container which is shown in figure 4.3. As shown in figure 4.3, this container provides only one Glassfish server.

Server	Deployment file
PICKUP_SHOP	cocome-pickup-war

Figure 4.4: Assignment archive files to Servers

¹<https://github.com/cocome-community-case-study/cocome-cloud-jee-web-shop>

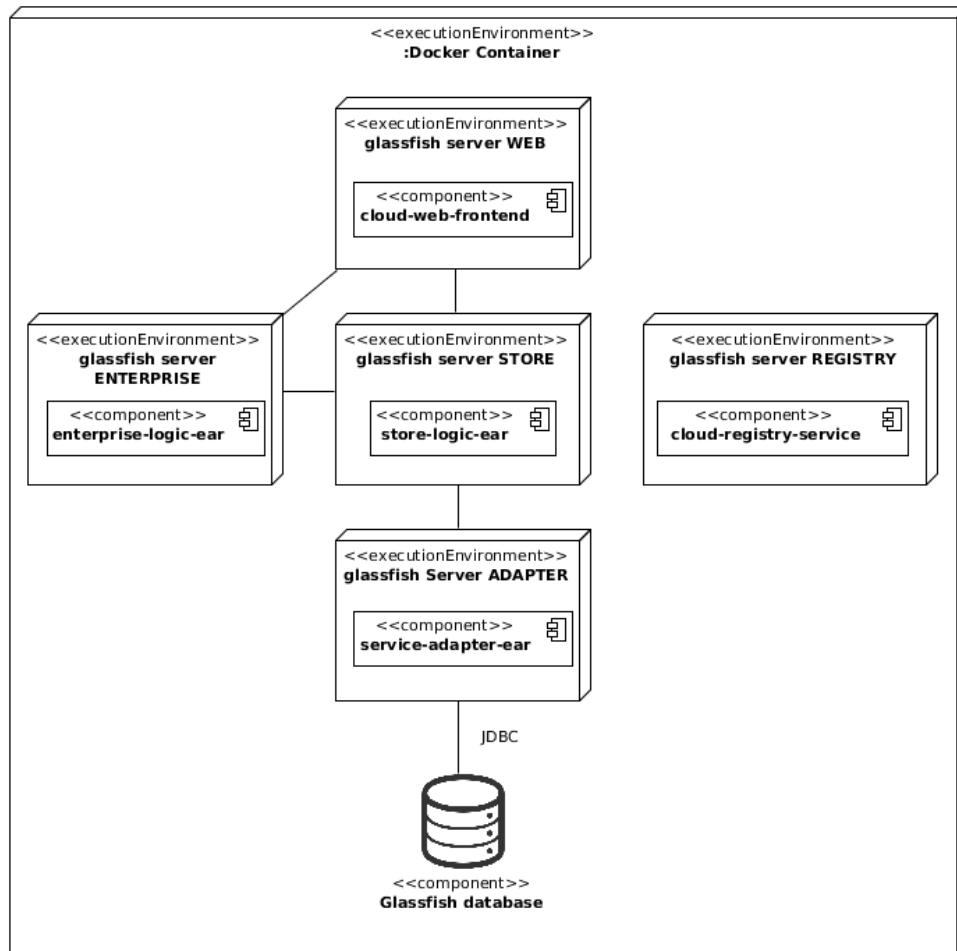


Figure 4.1: Deployment diagram CoCoME

To control the start of both containers, precisely the CoCoME and the Pick-Up Shop, another specific file is needed: the Docker Compose file. It ensures that the CoCoME Container is active, before the Pick-Up Shop container starts. This is necessary as the Pickup Shop requires a running instance of CoCoME to register itself.

Whereas CoCOME does not require the Pick-Up Shop, the inversion is not correct. Both containers need to communicate with each other. By default, docker prohibits any outgoing and ingoing communication from and in a container. This is solved by opening specific ports through which the communication is possible. Which ports the containers can use is specified in the Docker Compose file as well.

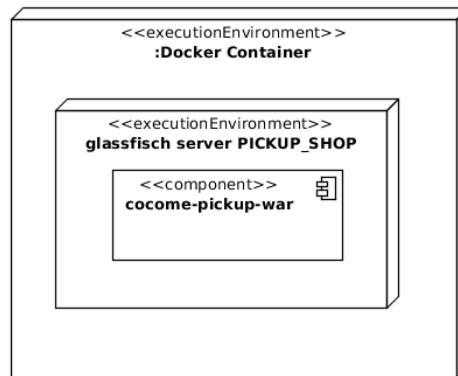


Figure 4.3: Deployment diagram CoCoME Pickup Shop

4.2 Adding a Mobile App Client

Adding a Mobile App Client did not require a modification within the hybrid cloud-based variant of CoCoME. The implementation was done using the Cordova framework and OnsenUI to provide a multi OS compatible backend and UI [3]. The App itself is written in Typescript/-Javascript. Fig. 4.5 shows the principal classes and their relationships.

The *Navigator* is the primary class that manages the pages. The pages consist of two components: The *Page* itself and its *PageState*. The *PageState* is used to store and transfer the current status of a page. There are currently six different pages available: *IndexPage*, *SearchPage*, *ItemPage*, *CheckoutPage*, *CartPage* and *LoginPage*. For the sake of clarity, they are subsumed under the generic terms *ConcretePage* and *ConcretePageState*.

Pages use components. Such components are i.e. the *Navbar* or the *Searchbar*. These components are abstract descriptions of UI elements that are connected to the actual *HTML-elements* via Knockout.js. By using Knockout.js, changing values of a component results in an immediate change of the UI. Besides, the App Client retrieves information of the CoCoME system. As mentioned in 3.1.2, the Client is not able to access the CoCoME system directly. Therefore, the pages use *Services* provided by a *ServiceHolder* to call the *AppController*' Rest-API. The *AppController* is written in Java using the SpringBoot framework and converts the Rest-requests of the App Client to SOAP-Requests in order to match the CoCoME-API.

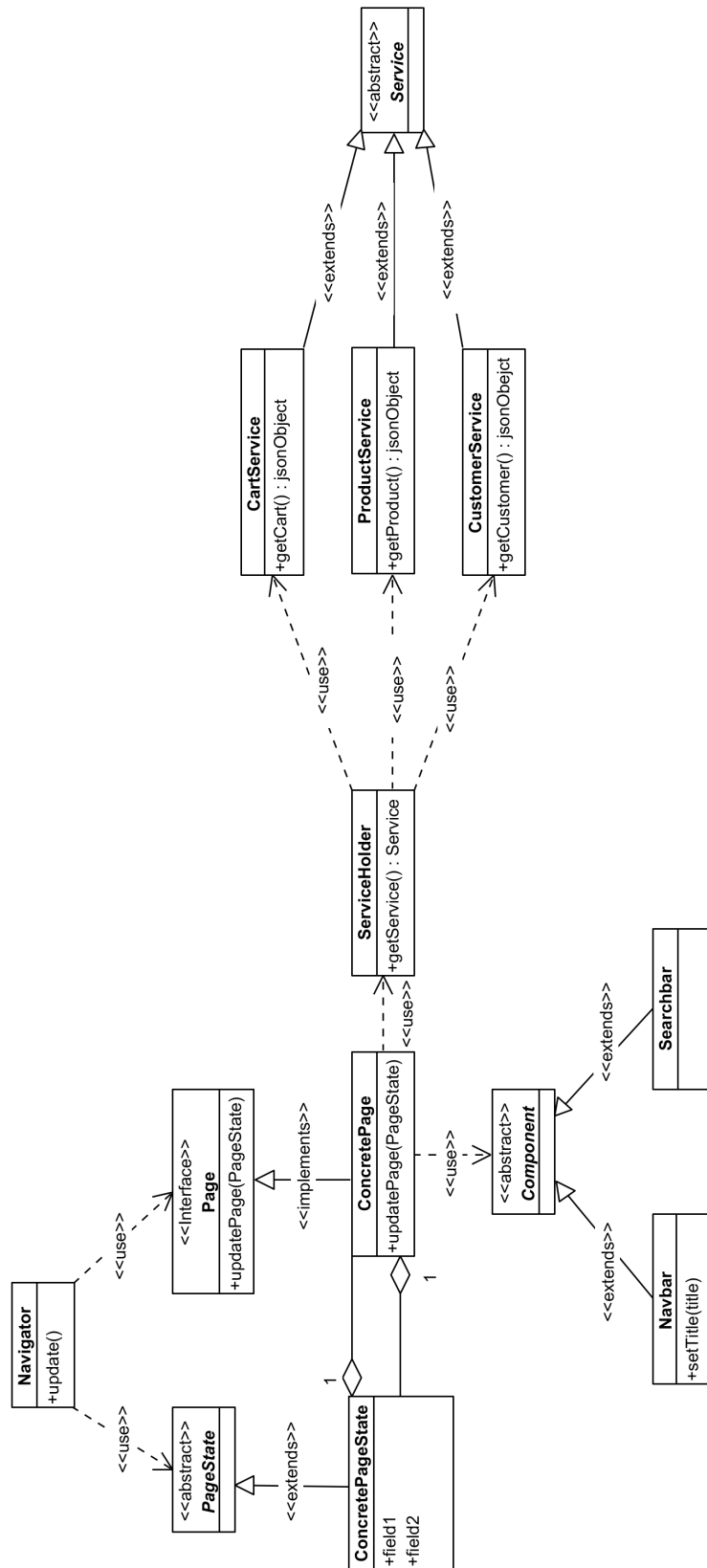


Figure 4.5: Primary Classes of the App

4.3 Using Microservice Technology

5 Conclusion

Bibliography

- [1] R. Heinrich, K. Rostami, and R. Reussner. The cocome platform for collaborative empirical research on information system evolution. Technical Report 2, 2016.
- [2] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the "stairway to heaven"—a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 392–399. IEEE, 2012.
- [3] J. Schnabel. Mobile application client for a cloud based software system. https://github.com/cocome-community-case-study/cocome-cloud-jee-app-shop/blob/master/doc/SQEE1617_MobileApp_Schnabel.pdf.
- [4] N. Sommer. Erweiterung und wartung einer cloud-basierten jee-architektur. <https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest/blob/master/doc/report.pdf>.