Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2016,2
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

The CoCoME Platform for Collaborative Empirical Research on Information System Evolution

Robert Heinrich, Kiana Rostami, Ralf Reussner *

2016

*Institute for Program Structures and Data Organization

Contents

1 Introduction **7**

2 CoCoME Platform Overview **8**

3 Evolution Subject **9**
3.1 Plain Java Variant 11
3.2 Service-oriented Variant 11
3.3 Hybrid Cloud-based Variant 13

4 Evolution Scenarios **14**
4.1 Evolution Scenarios of the Plain Java Variant 14
4.2 Evolution Scenarios of the Hybrid Cloud-based Variant 14
4.2.1 Platform Migration 14
4.2.2 Adding a Pick-up Shop 15
4.2.3 Database Migration 15
4.2.4 Adding a Service-Adapter 16

5 Design Details for Evolution Scenarios **17**
5.1 Replacing the Database 17
5.2 Platform Migration 17
5.3 Design Decision for the Hybrid Cloud-based Variant 26
5.3.1 Communication Model 26
5.3.2 Basic Components 29
5.3.3 Event Bus and Scopes 30
5.3.4 Web Service Adapters 30
5.4 Adding a Pick-up Shop 31
5.4.1 Use Cases of the Pick-up Shop 31
5.4.2 Design of the Pick-up Shop 34
5.5 Adding a Service-Adapter 40

6 Life-Cycle **48**

7 Implementation of Evolution Scenarios **49**
7.1 Platform Migration 49
7.2 Adding a Pick-up Shop 50

7.3 Adding a Service-Adapter 51
7.3.1 Service-Adapter API 51
7.3.2 Libraries and Frameworks 51

8 Conclusion 55

List of Figures

2.1 Overview of the Three Parts of the CoCoME Platform 8

3.1 Overview of the CoCoME Structure 10
3.2 Use Cases of CoCoME 10
3.3 Architecture Overview of the Plain Java Variant of CoCoME 12
3.4 Architecture Overview of the Service-oriented Variant of CoCoME 12

5.1 CoCoME Component Structure Before and After Change (Plain Java Variant). 18
5.2 First Part of the Sequence Diagram of the Process Sale Use Case. 19
5.3 Second Part of the Sequence Diagram of the Process Sale Use Case. 21
5.4 Sequence Diagram of the Process of Paying by Cash. 22
5.5 Sequence Diagram of Entering the Amount as Digits Through the Cash Box Number Pad (Cashier). 23
5.6 Sequence Diagram of Paying by Credit Card. 24
5.7 Sequence Diagram of Outputting Information About the Sale and Persisting the New State (Last Steps of a Sale) 25
5.8 Coarse Structure of the Hybrid Cloud-based Variant of CoCoME 26
5.9 Architecture Overview of the Hybrid Cloud-based Variant of CoCoME 28
5.10 Changed Use Cases due to the Introduction of the Pick-up Shop 31
5.11 Hybrid Cloud-based Variant Component Structure After Adding the Pick-up Shop (Added Components are Highlighted). 35
5.12 Sequence Diagram of Selecting a Store in the Pick-up Shop. 37
5.13 Sequence Diagram of the Process Online Sale Use Case of the Pick-up Shop. . 38
5.14 Sequence Diagram of Authenticating a User in the Pick-up Shop. 39
5.15 Sequence Diagram of Adding a new Credit Card of a Customer in the Pick-up Shop. 39
5.16 Adding a Service-Adapter to CoCoME 42
5.17 The Service-Provider Framework 43
5.18 Generic Navigation in the Service-Provider Framework 44
5.19 Response ServiceAdapter 45
5.20 Response ServiceProviderDatabase 46
5.21 Response ServiceProviderBookSale 47
5.22 Get-Request on ServiceProviderDatabase 47

6.1 Overview of the CoCoMEP Life-Cycle 48

7.1 Overview of the project structure in the cloud-logic-service project of the hybrid-cloud variant. 50
7.2 Overview of Format Parsing for Client Communication 52
7.3 The Lexer-Parser-Framework 53
7.4 The Table-Framework 54

Acknowledgement

We would like to thank our student assistants Tobias Pöppke and Giovanni Alessandro Giusa for contributing to this Technical Report.

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

1 Introduction

In industrial practice, many information systems are operated over decades. During operation they face various modifications, e.g. due to emerging requirements, bug fixes, and environmental changes, such as legal constraint or technology stack updates. In consequence, the systems change continually which is referred to as software evolution [13]. Supporting software evolution is a competitive advantage in software engineering. A variety of methods aim at supporting different aspects of software evolution. However, it is hard to assess their effectiveness and to compare them due to divergent characteristics. Empirical research in terms of case studies and controlled experiments is useful to validate these methods. However, empirical studies on software evolution are rarely comprehensive. They only cover few of the many aspects needed to study evolution such as (i) long time-frames of observation are required to analyze changes, (ii) large amount of artifacts and (iii) various types of artifacts are affected by evolution, (iv) artifacts repeatedly change, (v) changes partly build upon each other, (vi) various stakeholders are involved, (vii) access to relevant project data, (viii) relevant project data must be documented over long time spans, (ix) relevant context knowledge must be documented beyond the code base and issue trackers.

To study evolution comprehensively we believe it is important to collaborate by joint research in order to increase coverage of the aspects. Joint research supports sharing of knowledge and resources [16]. In particular, this allows replicating studies which in general is important to confirm and to strengthen results of empirical research [11] and thus enhance evidence. Our goal is to support joint research by collaboration and replication in empirical studies based on common evolution scenarios and artifacts. Currently, empirical studies on software evolution are seldom comparable as they vary in analyzed subjects and execution process. Furthermore, these studies are seldom reusable as important artifacts (e.g., requirements, design decisions, or context knowledge) are often not provided to the community. To the best of our knowledge, there is neither a community-accepted case study for software evolution nor a common benchmark available. Consequently, a common basis for study collaboration and replication is missing.

In this technical report, we propose CoCoMEP[1] – a platform for collaborative empirical research on information system evolution. Under a “platform" we understand a comprehensive knowledge base for the evaluation process that can be exploited and extended by other researchers with different backgrounds and research interests. It provides assistance on diverse characteristics important for software evolution, e.g. the life-cycle of the system, artifacts in different revisions, and comprehensive evolution scenarios. CoCoMEP builds upon the established CoCoME case study [10] which is further evolved in the course of this report.

[1]The term is a combination of Common Component Modeling Example “CoCoME " [10] and “Platform"

2 CoCoME Platform Overview

In [8] we analyzed related work on empirical research and conducted a literature review. We analyzed related work with regard to collaboration. In particular, we focused on replicability and comparability that are both indispensable to enable research collaboration. The aim was to learn from experiences in empirical research and derive requirements as basis for the design of CoCoMEP. In the literature review we analyzed how well existing studies on software evolution support the requirements identified in related work. The shortcomings identified in the literature review clarified the need for improvement in case study research on information system evolution as further described in [8].

We developed the research platform CoCoMEP depicted in Fig. 2.1 to address these shortcomings. The platform consists of three parts – evolution subject, evolution scenario and evolution life-cycle – which are described in detail in Sec. 3 through Sec. 6.

An evolution subject is the amount of artifacts in different revisions (e.g., requirements, design documents, source code, or monitoring data) that represent an information system. On this account, the established CoCoME case study serves as a study subject (Sec. 3).

An evolution scenario describes changes to a certain evolution subject. We developed examples of change scenarios in information system evolution and describe them in Sec. 4.

An evolution life-cycle integrates activities and their relationships required to implement one or more evolution scenarios. We constructed sample activities in system development and operation, and arranged them in life-cycle form (Sec. 6).

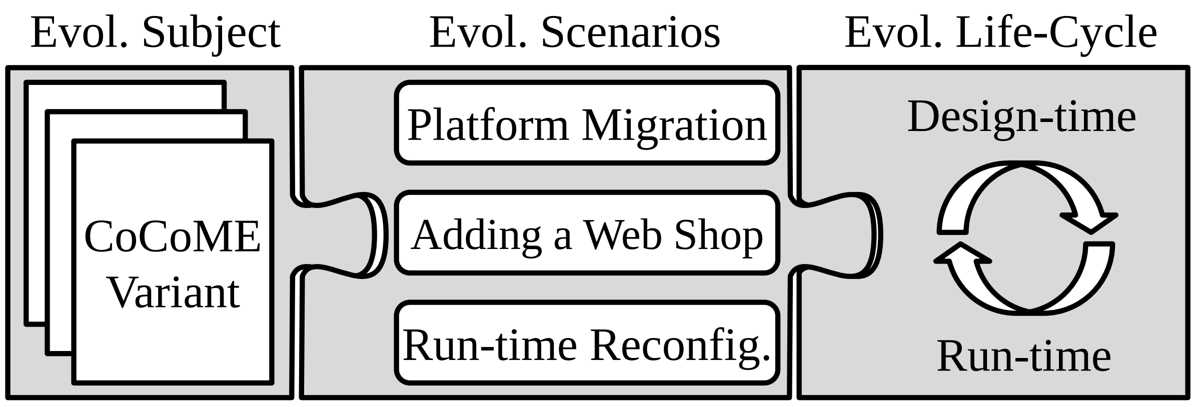


Figure 2.1: Overview of the Three Parts of the CoCoME Platform [8]

3 Evolution Subject

We use CoCoME as evolution subject in our platform. CoCoME has been set up initially in a GI Dagstuhl research seminar as a common case study on which several methods in the context of component-based software engineering have been applied. Since more and more people do research on software evolution, CoCoME has been applied in new areas as a demonstrator for software evolution methods.

CoCoME represents a trading system as it can be observed in a supermarket chain handling sales. This includes processing sales at a single store of the chain, e.g scanning products or paying, as well as enterprise-wide administrative tasks, e.g. inventory management or reporting. An overview of the structure of CoCoME is given in Fig. 3.1. Each store of the CoCoME chain contains several cash desks whereas the set of cash desks is called cash desk line. The cash desk is the place where the cashier scans the goods a customer wants to buy. The central unit of each cash desk is the cash desk PC. The cash desk line is connected to a store server. A set of stores is organized in the CoCoME enterprise where an enterprise server exists to which all stores are connected. Use cases supported by CoCoME are depicted in Fig. 3.2. A detailed description of the initial requirements, architecture, and system behavior in form of sequence diagrams is given in [10]. In the course of this report CoCoME faces changes by various evolution scenarios while we present changes to requirements, architecture, and system behavior. The CoCoME movie[1] gives a quick introduction to CoCoME in the context of the DFG Priority Programme 1593.

Since CoCoME has been applied and evolved successfully in various research projects, e.g. SLA@SOI[2], Q-Impress[3] or the DFG Priority Programme 1593[4], several variants exist that span different platforms and technologies, such as Plain Java code, service-oriented frameworks or hybrid cloud architectures. Furthermore, various development artifacts are available, such as requirements specification or design documentation, that changed over time. CoCoME is well suited to serve as a study subject because the supermarket context is commonly comprehensible and the complexity of the system is appropriate. As CoCoME is a distributed system, several quality properties are affected by evolution.

[1] http://www.dfg-spp1593.de/cocome
[2] http://sla-at-soi.eu
[3] www.q-impress.eu
[4] http://www.dfg-spp1593.de

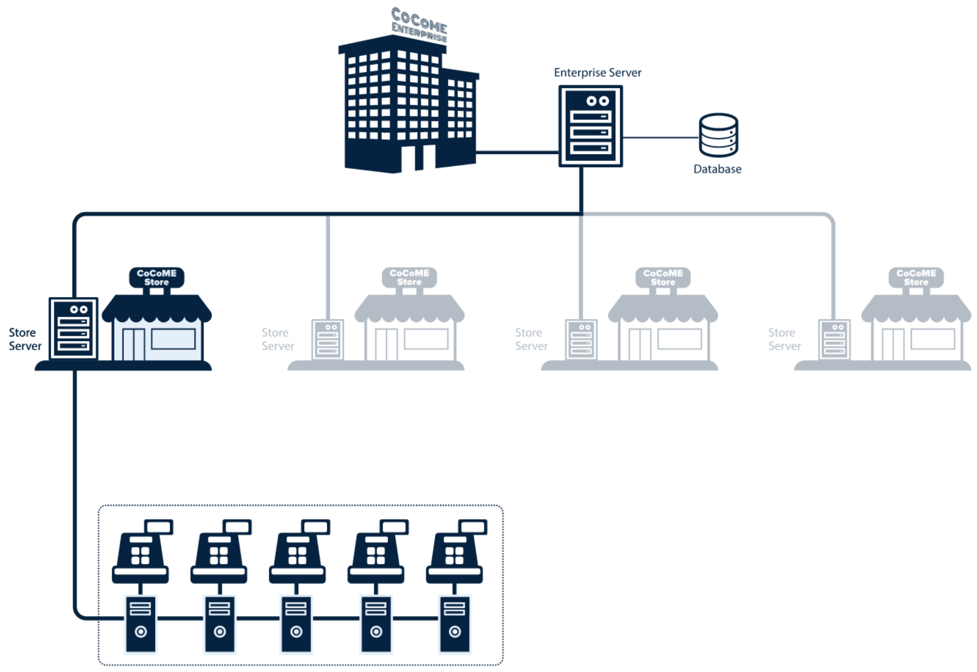


Figure 3.1: Overview of the CoCoME Structure

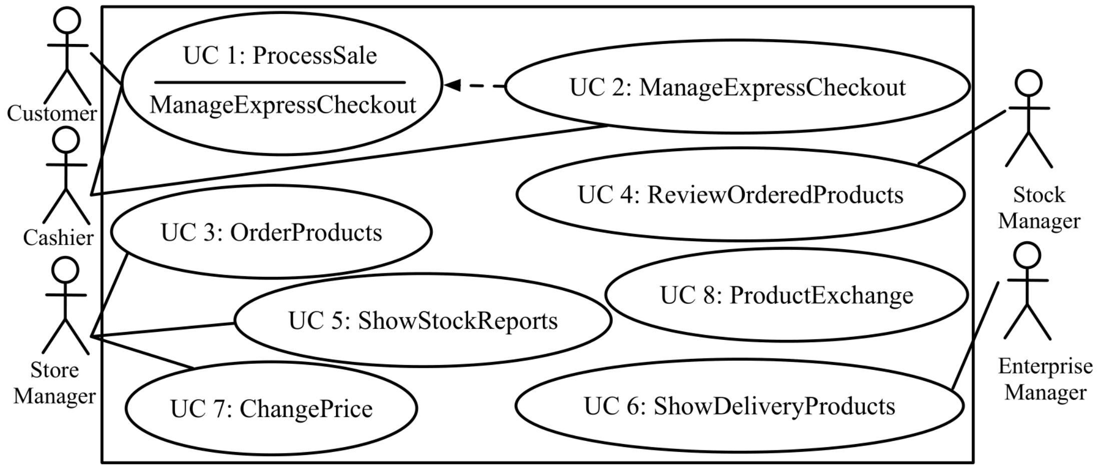


Figure 3.2: Use Cases of CoCoME [10]

3.1 Plain Java Variant

The Plain Java variant of CoCoME was the outcome of the GI Dagstuhl seminar on component-based software engineering. The architecture of the Plain Java variant is depicted in Fig. 3.3. The Plain Java variant of CoCoME uses Java SE in combination with Java Database Connectivity (JDBC), the Java Persistence API (JPA) and the Java Message Service (JMS). JMS is used to provide a way for communication between the components. The main component is the TradingSystem component. It consists of the TradingSystem::CashDeskLine component and the TradingSystem::Inventory component. The TradingSystem::CashDeskLine in turn consists of several CashDesk components representing the physical cash desks in a store with their corresponding components. There is one Coordinator component per store which receives sales events from the cash desks and changes the express mode state if needed. The TradingSystem::Inventory consists of the Console component which provides a user interface for store related operations through its Store component. The Console::Reporting component provides the user interface to retrieve enterprise or store reports. The central component of the TradingSystem::Inventory is the Application component. It provides the cash desk and the store user interface the operations to retrieve data and to book sales. The data is transferred in the form of Transfer Objects to provide an abstraction layer between the database and the other components. To retrieve the reporting information for the presentation layer, the Application::Reporting component provides the needed interface. There is also a ProductDispatcher component available to dispatch needed stocks from one store to another if necessary. A connection to the underlying database is realized by the Data component which relies on JDBC and JPA to persist and retrieve data. It is divided into three sub-components, Store, Enterprise and Persistence. The Store and Enterprise components are only used to query store or enterprise data, whereas the Persistence component writes objects to the database. See [10] for inner, more detailed component structure.

3.2 Service-oriented Variant

The service-oriented variant of CoCoME has been developed in the SLA@SOI research project. In the project CoCoME served as an open reference demonstrator. For constructing the service-oriented variant, CoCoME was extended with additional Web Service layer on top of the original CoCoME components. An architecture overview of the service-oriented variant is depicted in Fig. 3.4. A diverse set of deployment options was achieved by complete separation of the application logic from the data model and the service composition layer. The application was transformed into a manageable and SLA-aware application ready to be deployed with the SLA@SOI framework.

See project report *Deliverable D.B2b Reference Demonstrator*[5] for design documentation of the service-oriented variant of CoCoME.

[5] `http://sla-at-soi.eu/wp-content/uploads/2011/08/D.B2b-M38-Reference_Demonstrator.pdf`

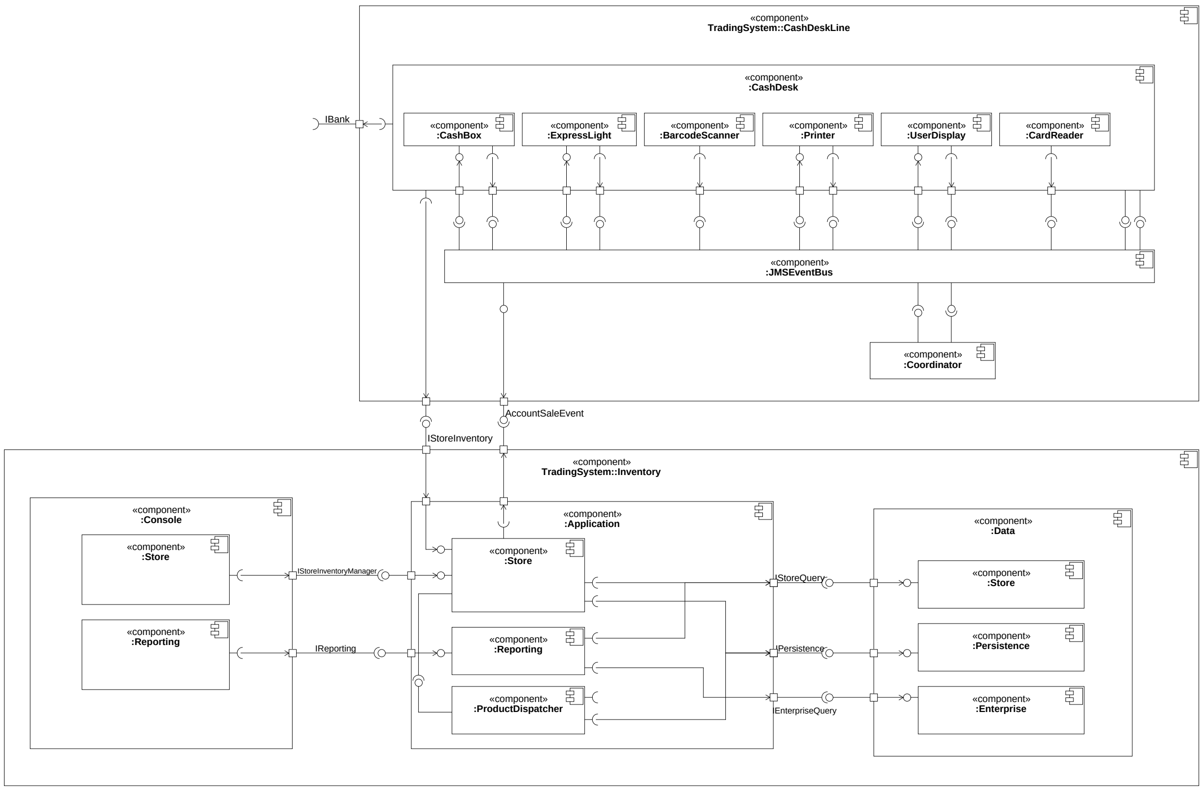


Figure 3.3: Architecture Overview of the Plain Java Variant of CoCoME [10]

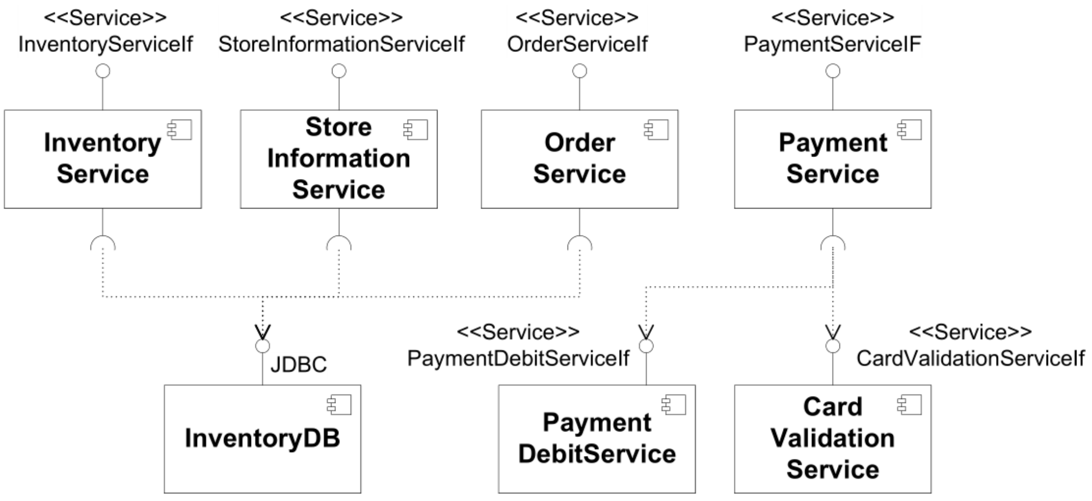


Figure 3.4: Architecture Overview of the Service-oriented Variant of CoCoME

3.3 Hybrid Cloud-based Variant

The hybrid cloud-based variant of CoCoME was developed in the DFG Priority Programme *Design For Future - Managed Software Evolution* (SPP 1593) [4]. This variant of CoCoME is using Java EE technologies for both the frontend and the backend. The hybrid cloud-based variant of CoCoME evolved from the Plain Java variant by implementing the evolution scenarios described in Sec. 4.2.

A coarse-grained component diagram of the hybrid cloud-based variant of CoCoME is shown in Fig. 5.8. The frontend uses Java Server Faces (JSF) to implement the user interface. In the WebFrontend::UseCases component the presentation logic is implemented which uses the components in the TradingSystem component to store the data retrieved from the ServiceAdapter. The ServiceAdapter component defines and implements an interface for database access and internally uses JDBC and JPA to access the underlying database. To query the database, the ServiceAdapter provides a Representational State Transfer (REST) style interface over Hypertext Transfer Protocol (HTTP) further described in Sec. 5.5.

Additional abstraction layers are introduced for the communication between the presentation layer and the business logic. These layers are located in the WebService component. The inner structure of the TradingSystem was nearly left unchanged as shown in the fine-grained component diagram in Fig. 5.9. One exception is the event bus. Instead of the JMS event bus the Context and Dependency Injection (CDI) event bus is used. Another change is that the components in Data now use the ServiceAdapter instead of the database directly. This allows for more flexibility in the cloud context. The newly introduced WebService::CashDesk component provides the frontend with a way to access the cash desk components. It is designed as a wrapper around the business logic so the method of accessing the business logic can be exchanged just by exchanging the wrapper classes. This is also the purpose of the WebService::Inventory component. The WebService::Inventory contains the Enterprise component to enable the frontend to access enterprise related information. This is necessary to enable several tasks needed for database administration like the listing of all stores in a specific enterprise. Design details are given in Sec. 5.

4 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. The scenario “Replacing the Database" reflects an adaptive evolution of the Plain Java variant of CoCoME (Sec. 4.1). An adaptive evolution of the hybrid cloud-based variant is reflected in the scenario “Platform Migration" due to evolving technology (Sec. 4.2.1). A perfective evolution is represented in the scenario “Adding a Pick-up Shop" by emerging user requirements (Sec. 4.2.2). Furthermore, in order to accommodate the self-adaptiveness of modern software architectures, reconfiguration during system operation is addressed in the scenario “Database Migration" (Sec. 4.2.3). "Adding a Service-Adapter" is another perfective evolution scenario to facilitate the further extension of the hybrid cloud-based variant (Sec. 4.2.4).

4.1 Evolution Scenarios of the Plain Java Variant

The evolution scenario **Replacing the Database** refers to the Plain Java variant of CoCoME [9]. In the scenario, CoCoME faces performance issues. In order to avoid them the company which operates CoCoME decides to replace the existing database. They shift away from a relational database (e.g., MySQL) to a non-relational database (e.g., CouchDB). Artifacts affected by this evolution scenario are described in [12].

4.2 Evolution Scenarios of the Hybrid Cloud-based Variant

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

4.2.1 Platform Migration

The CoCoME company must reduce operating costs of the resources and, therefore, migrates some resources to the cloud. The enterprise server and its connected database are now running in the cloud.

The introduction of the cloud enables flexible adaptation and reconfiguration of the system, however, causes new challenges regarding aforementioned quality properties which must be considered in development and operation. For example, a look back in the recent past shows that privacy is one of the most important quality properties for cloud systems.

4.2.2 Adding a Pick-up Shop

where the customers can order online and pick-up the goods at a chosen store. This design-time modification includes adding new use cases and modifying existing design models.

The CoCoME company is in competition with online shop vendors (such as Amazon). In order to increase its market share, the CoCoME company management decides to offer a pick-up service for goods to address emerging customer requirements. The customers can order and pay online. The goods are delivered to a pick-up place (i.e. a store) of her/his choice, for example in the neighborhood or the way to work. If the order has not been paid online, the goods have to be paid at the pick-up place (either per credit card or cash). Modifications regarding the use cases of the pick-up shop are depicted in Fig. 5.10. By introducing the pick-up shop as web application, the CoCoME system is transforms from a closed system (only employees can access and access depends on the location, e.g. a store) to an open system (customers can accessed via internet). This raises certain consequences such that the number of users is not restricted any longer. Hence, various quality properties are affected, e.g. privacy, security, performance, and reliability.

4.2.3 Database Migration

After a while, the CoCoME company starts a big advertise campaign. Advertisements lead to an increased amount of sales. Thus, the performance of the system may suffer due to limited capacities of the cloud provider currently hosting the enterprise database. Migrating the database from one cloud provider to another may solve the scalability issues.

Especially in the cloud, the application usage, performance, pricing and privacy are closely interrelated. The application usage impacts on the application's performance and pricing. Continuously appraised elasticity rules trigger the migration and replication of cloud application's software components among geographically distributed data centers. Both, migration and replication, may lead to the violation of privacy policies that prescribe certain geo-locations. Furthermore, a cloud application may also face performance/availability trade-offs as replication is often done for improving the system's overall availability, not just performance, which again might violate privacy policies [6].

This scenario represents a reconfiguration at run-time. Migrating the database may cause a privacy issue due to violations of privacy constraints [5]. According to a privacy constraint[1] of the European Union (EU) sensitive data must not leave the EU. Since the CoCoME enterprise is located within the EU its databases containing customer data must be hosted on data centers within the EU. This scenario is about dynamic analysis of cloud applications at run-time to identify upcoming quality flaws. It includes model-based observation and prediction techniques in flexible environments [6].

[1] http://eur-lex.europa.eu

4.2.4 Adding a Service-Adapter

CoCoME grew in size and complexity due to prior evolution scenarios. It is hard to extend the system because a large code-base must be understood by developers beforehand. Therefore, a Service-Adapter is added to the system which facilitates the further extension of CoCoME. It provides an additional layer to extend CoCoME by REST-Services. The Service-Adapter is composed of a set of services each of which can be browsed via a catalog and provides a certain functionality. Using the Service-Adapter the set of available functionality can extended easily by adding new services.

5 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. 5.1 sketches modifications to the design of the Plain Java variant by the evolution scenario Replacing the Database. Sec. 5.2 describes the evolution of the Plain Java variant to the hybrid cloud-based variant by the scenario Platform Migration. Detailed design decisions regarding this evolution are described in Sec. 5.3. The hybrid cloud-based variant is further modified in Sec. 5.4 by the evolution scenario Adding a Pick-up Shop and in Sec. 5.5 by the scenario Adding a Service-Adapter.

5.1 Replacing the Database

Replacing a relational database by a non-relational database raises certain consequences. Since JDBC has just been developed to provide a connection to relational databases, the interface has to be replaced, too. This is visualized in Fig. 5.1 by comparing the component structure before and after change. Moreover, in the given scenario, the `Data` component is affected by changing the `Database` component as depicted in the figure.

5.2 Platform Migration

The evolution scenario Platform Migration transfers the Plain Java variant of CoCoME to the hybrid cloud-based variant. As mentioned before, for the design of the hybrid cloud-based variant additional abstraction layers are introduced for the communication between the presentation layer and the business logic. These layers are located in the WebService::CashDesk and WebService::Inventory components depicted in Fig. 5.9. The WebService::CashDesk component provides the frontend with a way to access the cash desk components. The WebService::Inventory component enables the frontend to access enterprise related information. Wrappers are designed around the business logic so the method of accessing the business logic can be exchanged just by exchanging the wrapper classes.

An important task of the wrappers in WebService::CashDesk is to activate and deactivate the correct scope for the cash desk components to run in. Because the cash desk components are stateful, it is necessary that a method is called on the correct component. The CashDeskSessionScoped scope stores the states of the cash desk components of every cash desk and the correct scope has to be active when invoking a method on a cash desk. To activate the correct scope, the wrapper uses the name of the cash desk and the identifier of its containing store.

The deployment of the hybrid cloud-based variant is similar to the plain Java variant. In

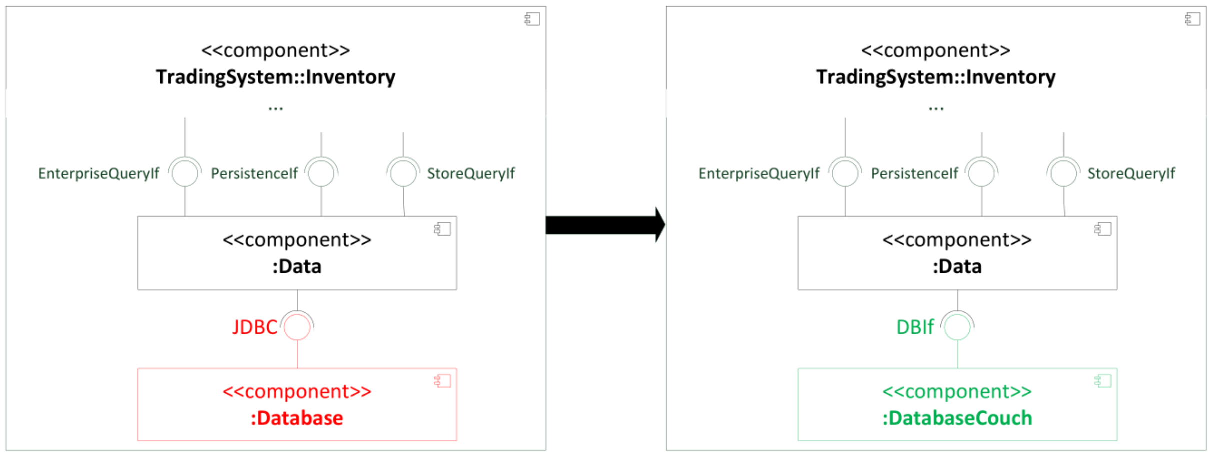


Figure 5.1: CoCoME Component Structure Before and After Change (Plain Java Variant).

Fig. 5.9 the deployment of a component is annotated in brackets. If there is no deployment annotated to a component, it is the same as for the containing component. The store server includes the components WebService::CashDesk, WebService::Inventory::Store, TradingSystem::CashDeskLine, TradingSystem::Inventory::Application::Store, TradingSystem::Inventory::Data::Store as well as the External::Bank component. The enterprise server includes all remaining components of the WebService and TradingSystem components and both servers access the data they need through the Service-Adapter. In addition to those components, the WebFrontend::UseCases component is deployed on a separate server. The Service-Adapter may be deployed on the enterprise server or on a separate database server.

As a behavioral overview of the evolved cloud-based variant of CoCoME, it suffices to look at the Process Sale use case introduced in Herold et al. [10]. This use case gives a good overview of the interactions between all system components during a sale. Because the business logic was merged from the monolithic variant, the behavior did not change for the other use cases except for the calls to the wrapper classes. Therefore, see the behavioral view of the original monolithic variant for further details. Note, interface names are used in the following to denote that an implementation of the interface is injected into the calling component via a dependency injection mechanism.

Figures 5.2 and 5.3 show the sequence diagram of the Process Sale use case in the evolved cloud-based variant. To start the sale, the cashier presses the "Start New Sale" button at an instance of *NewSaleProcess* in the *WebFrontend*. This initiates a call to the *pressControlKey* method in the *ICashBox* wrapper which in turn activates the correct *CashDeskSessionScoped* instance for the cash desk on which this operation should be executed. The scope is addressed by the store identifier and the cash desk name.

The wrapper then calls the method on the *ICashBoxModel*, which then proceeds to send a *SaleStartedEvent* over the event bus to notify other components of the new sale. Those components can then reset their state accordingly. When all components have been notified

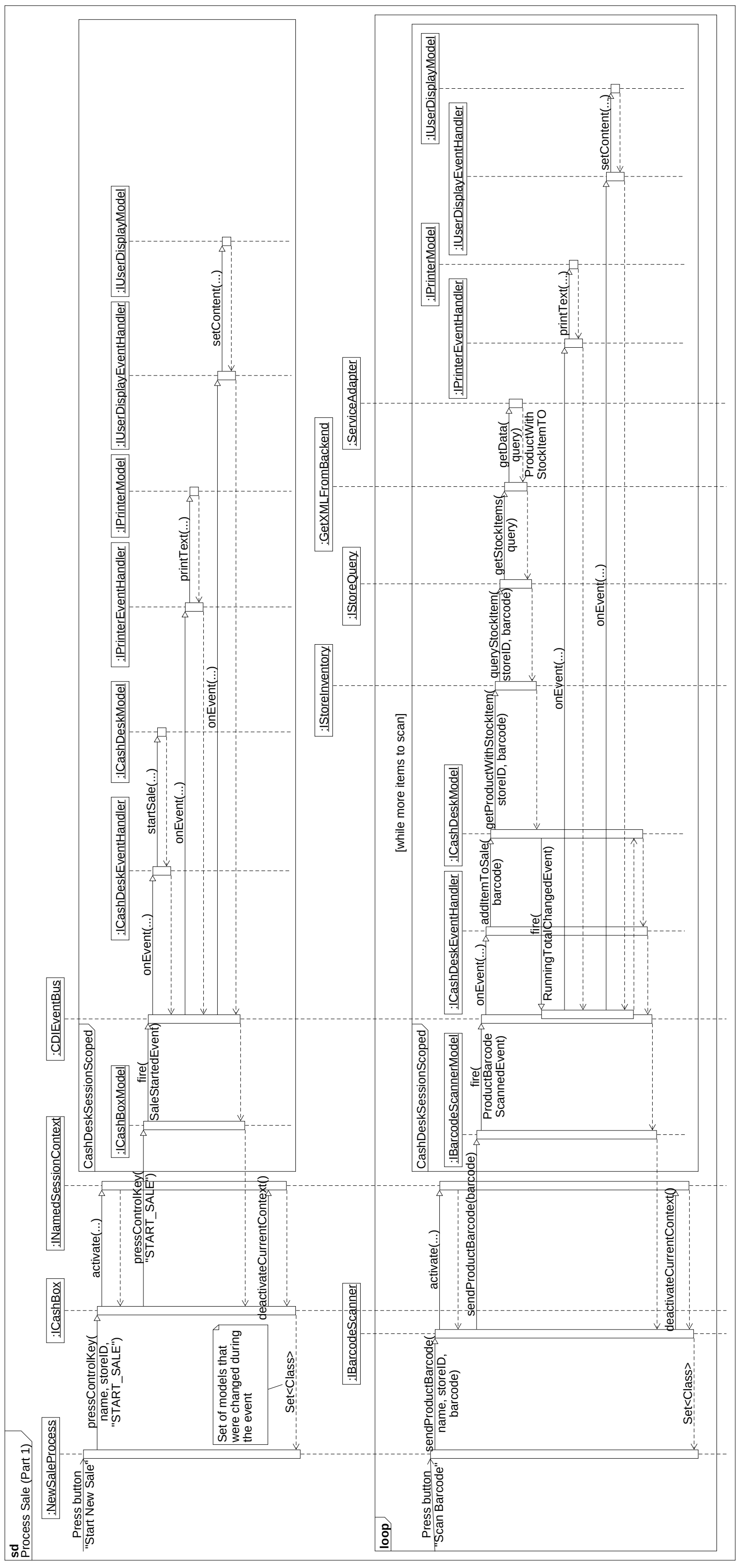


Figure 5.2: First Part of the Sequence Diagram of the Process Sale Use Case.

and the operation on the *ICashBoxModel* returned, the wrapper deactivates the scope. There also exists a listener that listens to *ContentChangedEvents*. Those events are fired when the state of a component changes and the names of the components that had their state changed since the last check can be retrieved from the listener. All wrappers return a set of names of changed components to the caller, so that the caller may poll their new state subsequently. Now the cash desk is ready to scan items and add them to the sale.

New items are added when the cashier scans the barcode of a product or inputs the barcode manually. Because of the *WebFrontend* of this variant of CoCoME it is only possible to input the barcode manually and then press the button "Scan Barcode". This again triggers the activation of the correct scope and forwards the event to the *IBarcodeScannerModel* component. A *ProductBarcodeScannedEvent* is fired and the *ICashDeskModel* queries the product information for the scanned item from the database by calling *getProductWithStockItem* on its *IStoreInventory* component. This in turn queries the database through the ServiceAdapter to get the needed information. This process is repeated as long as there are items to add to the sale.

If no more items are left to be scanned, the customer may choose to pay by cash or by credit card. The process of paying by cash is depicted in the sequence diagram in Fig. 5.4 and the process of paying by credit card in Fig. 5.6. The cashier may start the payment process by either clicking on the "Pay by Cash" or "Pay by Credit Card" button. Clicking either one of them will result in a call to *ICashBoxModel* to finish the sale and fire a *SaleFinishedEvent*.

If the customer chose to pay by cash, the next step is to fire a *PaymentModeSelectedEvent* from *ICashBoxModel*. The cashier can then either directly enter the received cash amount or type the amount on the number pad of the cash box. Because there is only a *WebFrontend* in the cloud-based variant, the cashier has to enter the amount directly. The process of typing in the digits at the cash box is shown in Fig. 5.5. In both cases when the amount was entered a *CashAmountEnteredEvent* is fired to update the other components with the received amount. Directly after that, the *ICashDeskModel* calculates the change amount and sends it to the listening components via a *ChangeAmountCalculatedEvent*. This causes the cash box to open, so the cashier can collect the cash amount and return the change to the customer. When closing the cash box, the sale is getting booked to the database.

If the customer wants to pay by credit card, *ICashBoxModel* fires the corresponding *PaymentModeSelectedEvent*. The customer then enters her/his credit card info and her/his credit card PIN until the *IBank* returns a valid transaction and a positive *DebitResult* or the customer chooses to pay by cash.

To account for a sale, paid either by cash or by card, and to persist the new stock item amounts, the finish sale sequence is called. Fig. 5.7 shows the steps in this procedure. First, an *AccountSaleEvent* is fired which signals the *IStoreEventHandler* to account the sale and update the database. After that, the cash desk components get informed that the sale was successful by a *SaleSuccessEvent* and finally the *ICoordinator* is called through a *SaleRegisteredEvent*. The coordinator then checks if the cash desk should switch to express mode or not. If express mode is required, it fires an *ExpressModeEnabledEvent*. In contrast to the original Plain Java variant of CoCoME, it is not yet possible to queue changes while the ServiceAdapter is not reachable.

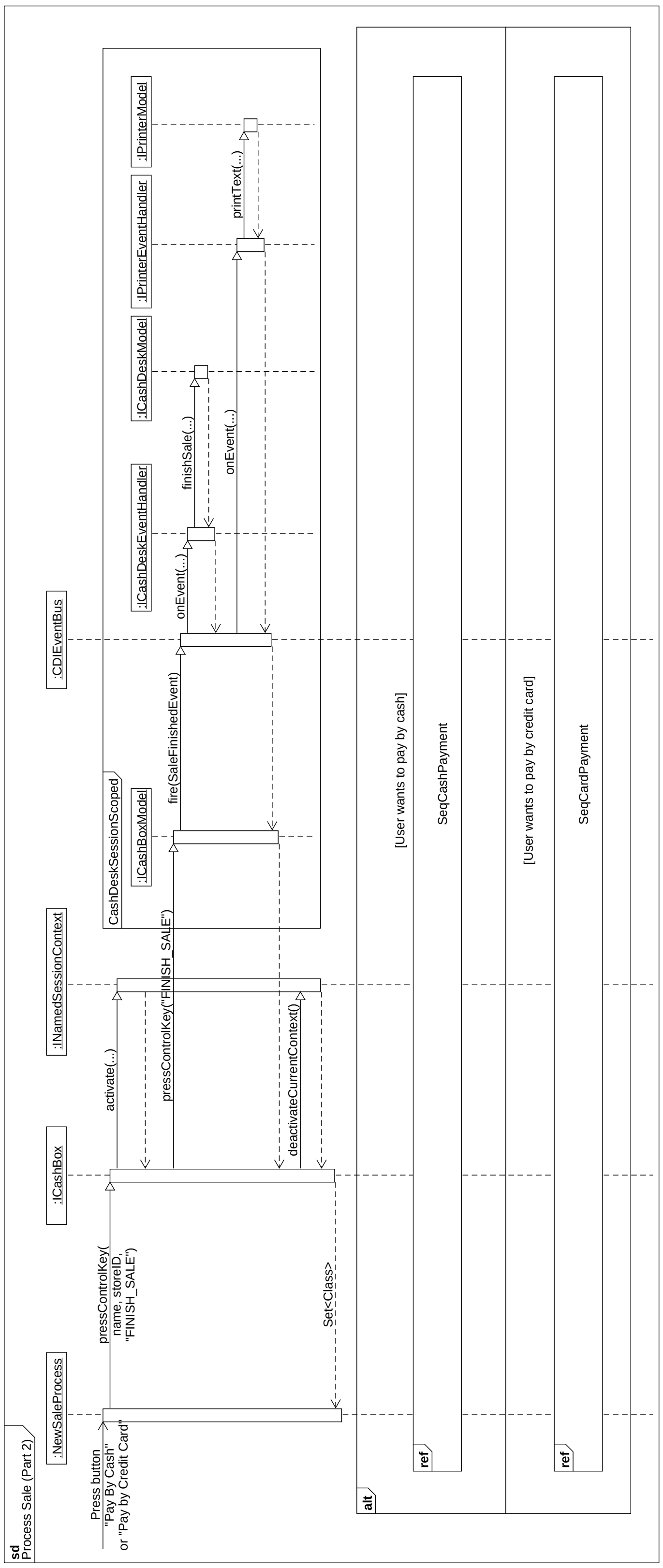


Figure 5.3: Second Part of the Sequence Diagram of the Process Sale Use Case.

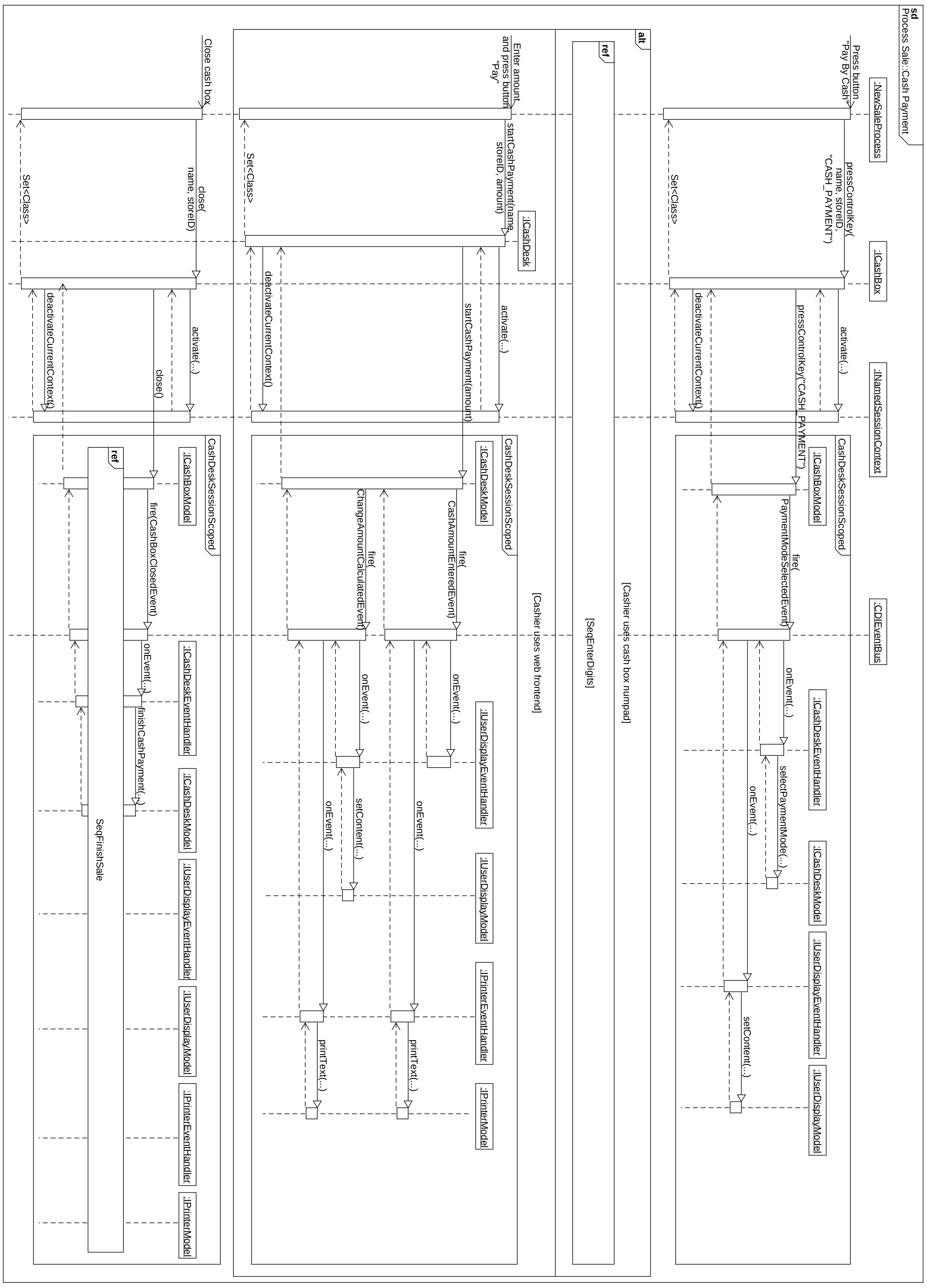


Figure 5.4: Sequence Diagram of the Process of Paying by Cash.

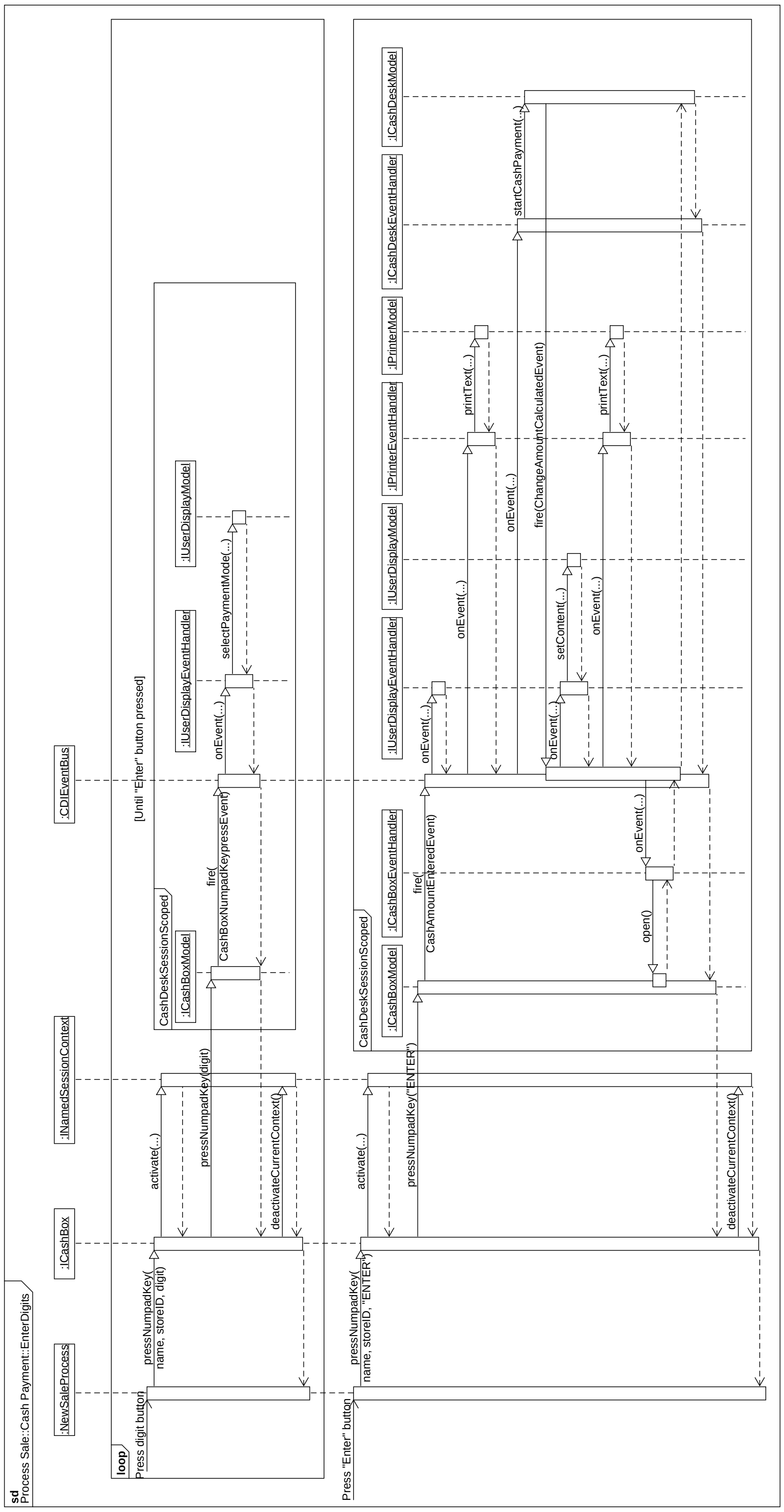


Figure 5.5: Sequence Diagram of Entering the Amount as Digits Through the Cash Box Number Pad (Cashier).

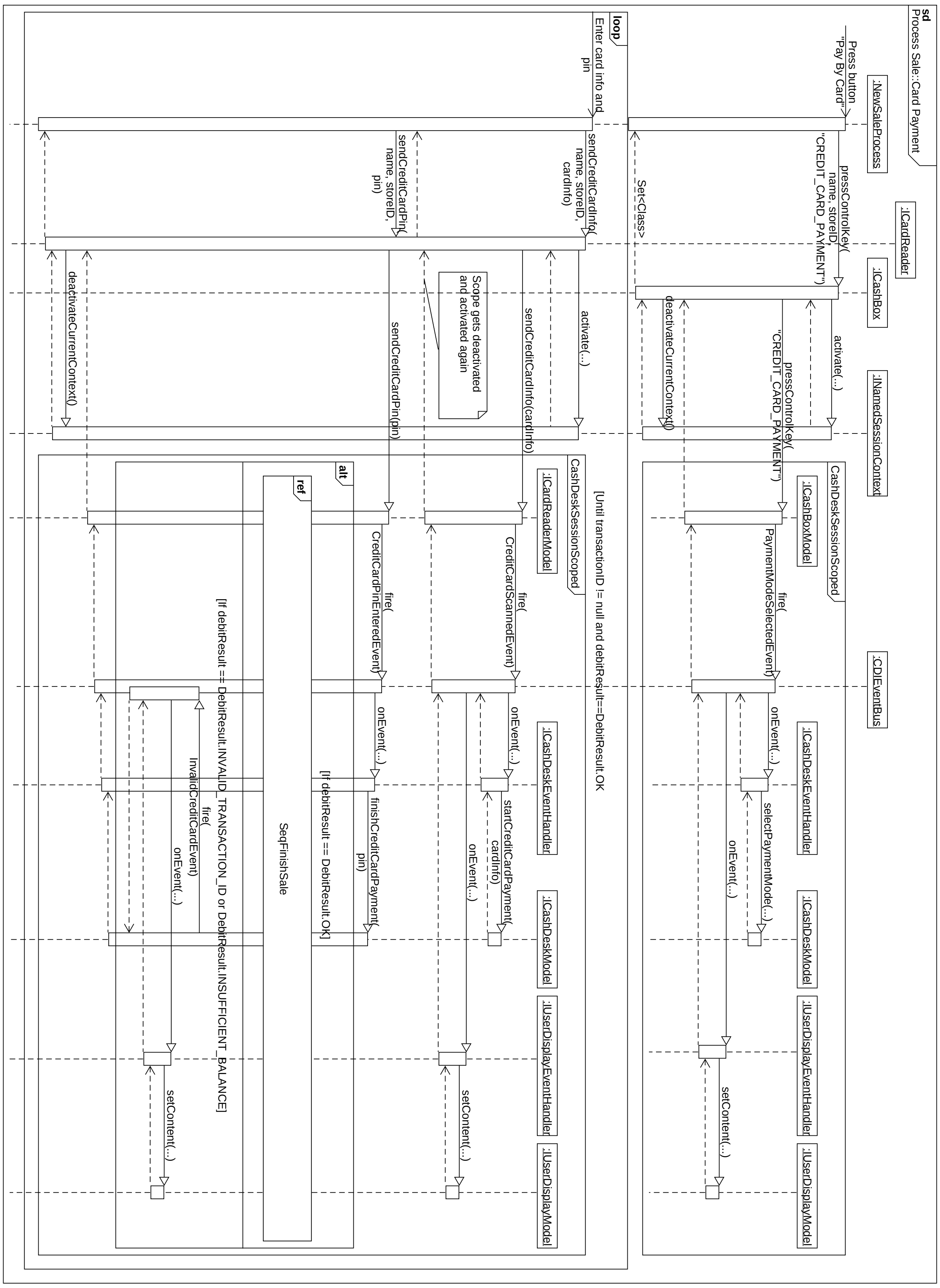

sd Process Sale::Card Payment
:NewSaleProcess
:CardReader
:CashBox
:INamedSessionContext
:CDIEventBus
Press button "Pay By Card"
pressControlKey(name, storeID, "CREDIT_CARD_PAYMENT")
Set<Class>
activate(...)
deactivateCurrentContext()
pressControlKey("CREDIT_CARD_PAYMENT")
CashDeskSessionScoped
:CashBoxModel
fire(PaymentModeSelectedEvent)
onEvent(...)
:CashDeskEventHandler
selectPaymentMode(...)
:CashDeskModel
onEvent(...)
:IUserDisplayEventHandler
setContent(...)
:IUserDisplayModel
loop
Enter card info and pin
sendCreditCardInfo(name, storeID, cardInfo)
sendCreditCardPin(name, storeID, pin)
Scope gets deactivated and activated again
sendCreditCardInfo(cardInfo)
sendCreditCardPin(pin)
[Until transactionID != null and debitResult==DebitResult.OK
:CardReaderModel
fire(CreditCardScannedEvent)
fire(CreditCardPinEnteredEvent)
startCreditCardPayment(cardInfo)
finishCreditCardPayment(pin)
alt
ref
SeqFinishSale
[If debitResult == DebitResult.OK]
[If debitResult == DebitResult.INVALID_TRANSACTION_ID or DebitResult.INSUFFICIENT_BALANCE]
fire(InvalidCreditCardEvent)

Figure 5.6: Sequence Diagram of Paying by Credit Card.

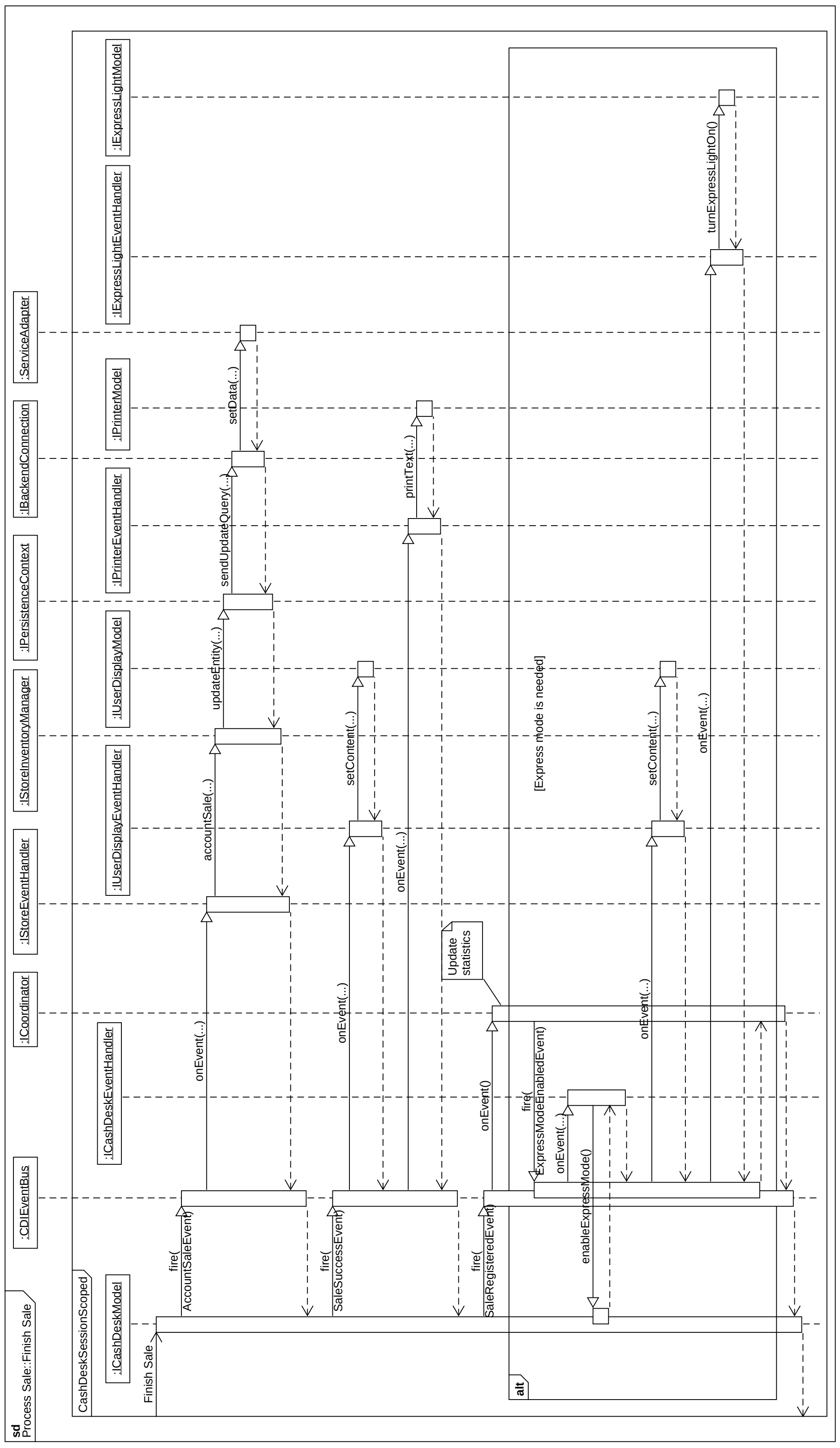


Figure 5.7: Sequence Diagram of Outputting Information About the Sale and Persisting the New State (Last Steps of a Sale)

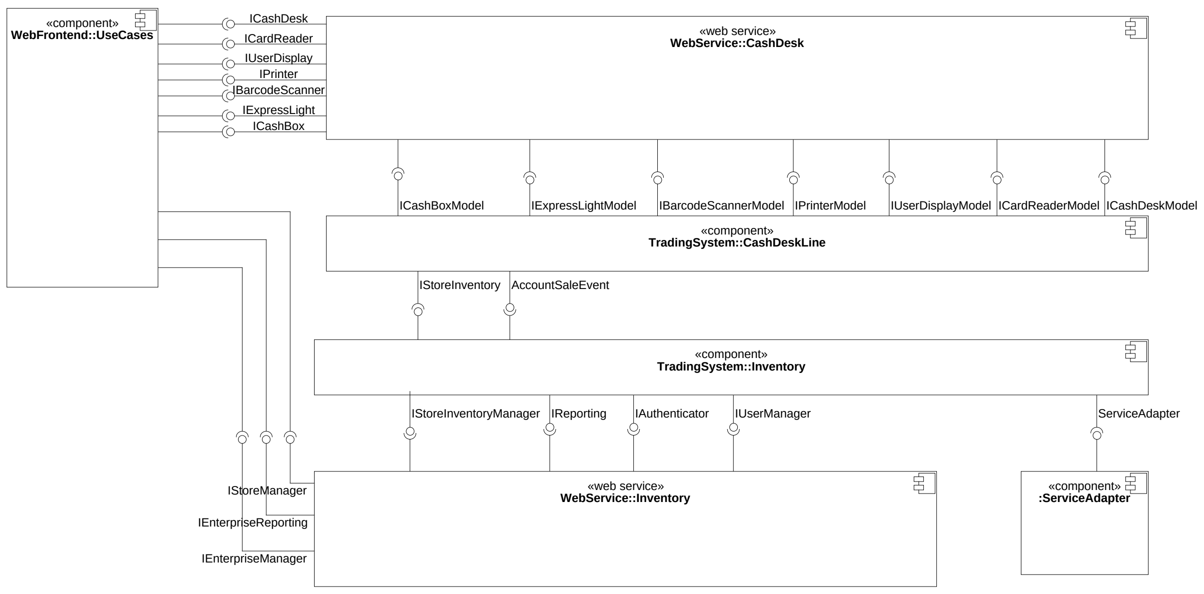


Figure 5.8: Coarse Structure of the Hybrid Cloud-based Variant of CoCoME

5.3 Design Decision for the Hybrid Cloud-based Variant

To achieve the functionality and the ability to deploy the evolved variant in the cloud, several design decisions had to be made. These decisions concern the basic components of which the components will be composed, the event bus to be used and the scopes for each component. Also the decision to implement the web service via an additional wrapper layer will be discussed, as well as the communication model for this web service.

5.3.1 Communication Model

To achieve the separation of presentation layer and the business logic layer in a way that enables the system to be deployed on different servers in a cloud, it is necessary to decide in which way the layers communicate with each other. The most popular way to solve this, is the use of web service interfaces, because they provide a method to integrate different layers in a Remote Procedure Call (RPC) style. The decision which communication model to use for communication between the presentation layer and the business logic layer is explained here.

Table 5.1: Comparison between the supported concepts of REST and WS-* web services.

Supported Concept	RESTful	WS-*
Contract Design		
Contract-first		✓

Continued on next page

Table 5.1: Continued

Supported Concept	RESTful	WS-*
Contract-last		✓
Contract-less	✓	
Data Modeling		
XML Schema		✓
Do-it-yourself	✓	
Payload Format		
XML (SOAP)	✓	✓
XML (POX)	✓	
JSON	✓	
YAML	✓	
MIME	✓	

There are two main styles of web services currently in wider use. These are the RESTful style [3] and the SOAP/WS-* based web services [1]. There exist a wide variety of other methods to integrate applications via RPC like the Java Remote Method Invocation (RMI). The problem with these methods and technologies is that they are not interoperable and therefore put limitations on the implementation of the clients.

Pautasso et al. [15] give an overview of the architectural decisions and possibilities for each of the two web service styles. Based on their comparison of both web service styles, the decision was made to implement a WS-* style web service as the interface of the business logic. In Tab. 5.1 a part of the comparison between RESTful and WS-* web services is shown. This comparison is based on the above mentioned work by Pautasso et al. and shows some important aspects that lead to the conclusion to use a WS-* style web service.

The first decision is based on the supported contract design. WS-* style web services support a contract-first or a contract-last approach. That means, it is possible to first define a contract and the implement the web service according to that contract or to derive the contract from an existing implementation. A contract for a WS-* web service is defined by a WSDL file and describe the methods, arguments and return types of the web service. RESTful web services only support a design without contracts. Instead, the caller has to obtain the needed information like method arguments by other means.

Because the existing monolithic variant of CoCoME already defines interfaces for the presentation layer to access the business logic the contract-last or the contract-less approach are possible choices.

The next choice is what kind of data modeling to use. Because the WS-* web services are based on XML and SOAP messages, the WS-* web services support strongly typed data types through XML schema files. RESTful web services rely on the caller to correctly interpret the data coming from the web service. Because of the additional tool support for data types and

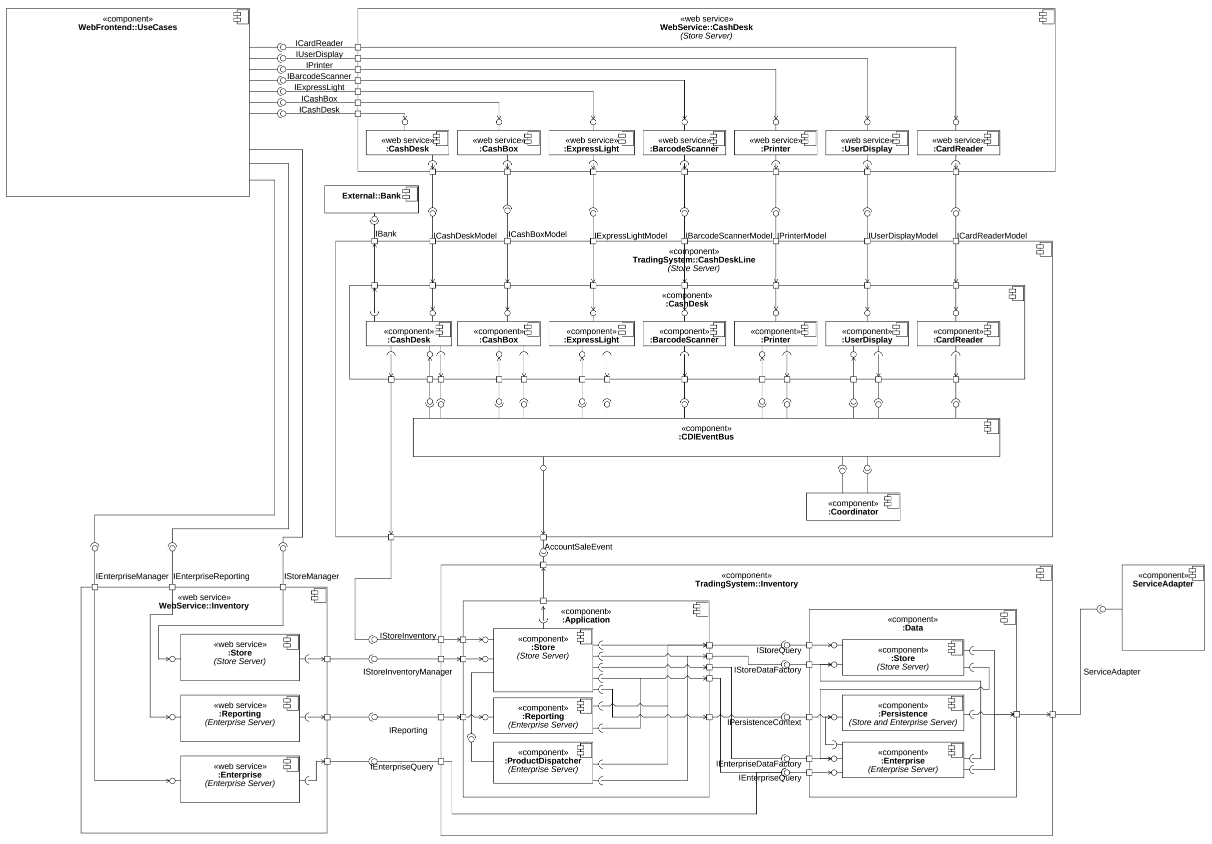


Figure 5.9: Architecture Overview of the Hybrid Cloud-based Variant of CoCoME

the automatic generation of stubs in the client code, the WS-* web service is better suited for communication between the presentation layer and the business logic layer in the cloud-based CoCoME variant.

WS-* web services can only be used in conjunction with XML and SOAP and therefore limit the possible payload of the messages. In contrast, RESTful web services are very flexible with regards to the payload and can also use Plain Old XML (POX), JSON, YAML and MIME as payload. JSON is the most frequent choice because of the compact payloads it allows. Using WS-* web services results in a significant communication overhead because of the verbosity of the XML/SOAP combination in contrast to using RESTful web services with JSON.

Both web service styles support HTTPS for securing the communication between the client and server. If more advanced security features are needed however, only WS-* web services provide a standardized way to enable message level security features through the WS-Security standard. Because CoCoME runs in an enterprise environment it may be necessary to implement such message level security features.

The tool support for WS-* web services is more developed because the WS-* stack is standardized and therefore easier to develop generic tools for it than for the highly flexible and unstandardized RESTful web service style. With the existing tools for WS-* web services it is possible to generate code stubs for most popular languages. That makes it easier to implement additional clients to the business logic like mobile devices.

The WS-* web services therefore provide a more convenient and easy way for clients to access the web services and considering the standardization as well as the security features offered, the decision was made to use WS-* web services for communication between the presentation layer and the business logic layer.

5.3.2 Basic Components

The basic components used to implement the original monolithic variant of CoCoME are Plain Old Java Objects (POJOs). The initial cloud-based variant uses Java EE technologies like JSF, Enterprise Java Beans (EJBs) and Context and Dependency Injection (CDI) beans in conjunction with POJOs. This necessitates an application server but enables the support of transactions, sessions and the possibility to define different security requirements on EJBs. Every EJB is also a CDI bean and therefore inherits their capability of dependency injection and context dependency. Context dependency enables a CDI bean to be active, and for example receive events, only while a specific context is active.

Because the initial cloud-based variant is already using Java EE technologies and the advantages like contexts and dependency injection provided by them are the reasons to also use these technologies for the implementation of the business logic. Furthermore, Java EE enables the generation of WS-* web services by simply annotating classes and methods. The application server then generates the contract WSDL file automatically and the code stubs for the client can also be generated with tool support.

This leads to the decision to leverage the abilities of EJB and CDI beans as the basic components of the business logic layer.

5.3.3 Event Bus and Scopes

CDI provides an event bus implementation to decouple event producers and consumers. All events fired are synchronously distributed to all registered consumers in the currently active scope. There are five predefined scopes defined in CDI: application scope, request scope, session scope, dependent scope and conversation scope.

The application scope is active while the application server is running and the application is deployed on the server. The request scope is active during a single call from the client to the CDI bean whereas the session scope is active during all calls from one client to the bean during a session. All CDI beans are dependent scoped by default. This scope is active depending on the life cycle of the one client to which the dependent scoped object is bound. The conversation scope is active during the interaction of a single user with a Java Server Faces application. This may also span multiple invocations of the application.

It would be beneficial if any cash desk component could fire events that are delivered only to other cash desk components of the same cash desk. This would allow to simplify these components, not needing to include the cash desk's name in each event or creating different JMS topics as it is the case in the original variant. Furthermore, it is necessary for a cash desk to keep its state, for example if the express mode was activated, during the complete lifetime of the cash desk. None of the above predefined scopes matches these requirements so it was necessary to implement a custom scope, the *CashDeskSessionScoped* scope.

This scope is activated manually and is addressed by the cash desk's name and the store identifier of the cash desk. The scope may be deactivated after the completion of an operation on the cash desk components. After deactivation, all components remain in their current state. It is possible to delete a specific cash desk's scope, but it is not performed automatically, that means all cash desk scopes that are being created stay in memory. This reflects the nature of a supermarket, where the number of cash desks normally does not change over a long time period.

Because of the benefits of a specific scope for the cash desk components, the *CashDeskSessionScoped* scope was implemented.

5.3.4 Web Service Adapters

The communication between the presentation layer and the business logic layer is done via the web service wrappers as stated above. It is possible to annotate the business logic components themselves to provide the required interface. However, this would make it more difficult to change the integration style and the business logic components would have the additional responsibility of exposing their functionality as a web service.

This can be avoided by wrapping the business logic with the web service wrappers. These wrappers are only responsible for exposing the web service to be used by the clients and the business logic components can be maintained separately. Another benefit of this additional abstraction layer is the possibility to exchange the web service wrappers without affecting the business logic.

Also, because the cash desk components rely on the *CashDeskSessionScoped* scope to be active, the web service wrappers can be used to activate the correct scope for each invocation of their methods. For these reasons the decision was made to implement the web service as a wrapper around the business logic.

5.4 Adding a Pick-up Shop

Building upon the hybrid cloud-based variant of CoCoME this section discusses design details of the Pick-up shop. Sec. 5.4.1 describes modifications and extensions to use cases. Sec. 5.4.2 describes modifications and extensions on design level.

5.4.1 Use Cases of the Pick-up Shop

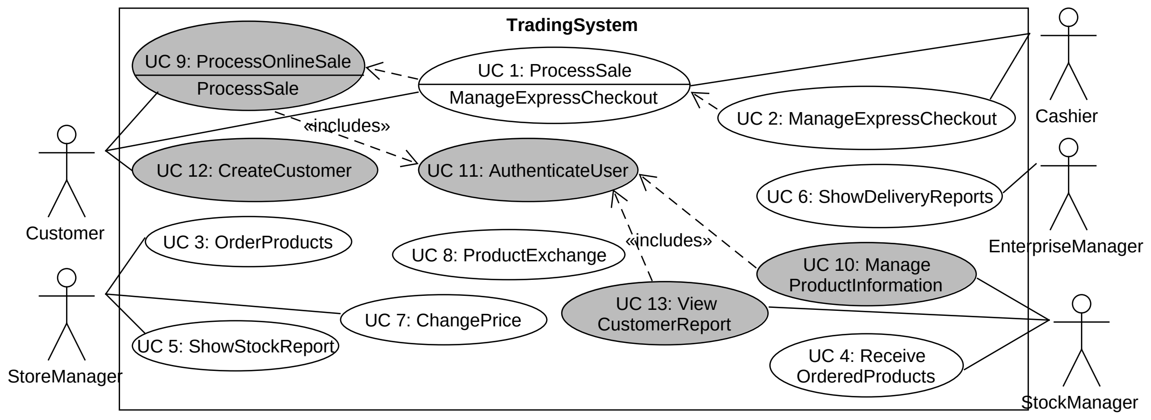


Figure 5.10: Changed Use Cases due to the Introduction of the Pick-up Shop

UC 9 - Process Online Sale

Brief Description A Customer selects the product items s/he wants to buy and the payment by credit card is performed.
Involved Actors Customer, Bank
Precondition The Pick-up Shop is ready to process a new sale and the Customer already has an account registered in the System.
Trigger The Customer requests the Pick-up Shop website and wants to buy product items.
Postcondition The Customer has paid and the sale is registered in the Inventory.
Standard Process

1. The Customer selects the Store where s/he wants to pick up his purchased product items.

2. The Customer adds the product items s/he wants to purchase to the Shopping Cart by clicking the *Add to Cart* button.
   Step 2 is repeated until all items are added to the cart.
3. The Customer clicks the button to display the Shopping Cart.
4. The System presents the Customer with the item names, prices and the running total.
5. Denoting the end of adding items, the Customer presses the *Proceed to checkout* button.
6. The Customer is presented with a login form and is required to complete the *Authenticate User* use case.
7. In order to initiate card payment, the Customer selects a credit card to use for the purchase.
8. The Customer enters her/his PIN in the designated field presented by the System.
9. The System presents the Customer with an overview of the purchase, the Customer confirms the purchase and waits for validation.
   Step 9 is repeated until a successful validation or the Customer decides to cancel the purchase.
10. Completed sales are logged by the System and sale information are sent to the Inventory in order to update the stock.
11. The Customer is presented with a success message and the product items are being prepared to be picked up by the customer.
12. The Customer leaves the website.

Alternative or Exceptional Processes

- *In step 7: 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 9: Card validation fails*
  1. The Customer tries again and again.
  2. Otherwise, the Customer can decide to cancel the purchase.

UC 10 - Manage Product Information

Brief Description The System provides the opportunity to change the stored information of a product.

Involved Actors Stock Manager

Precondition The store management website is available and the Stock Manager is authenticated.

Trigger The Stock Manager wants to change the stored information of a product.

Postcondition The stored information for the considered product has been changed and will now be shown when requested by the Pick-up Shop.

Standard Process

1. The System presents an overview of all available products in the enterprise.
2. The Stock Manager selects a product item and changes its stored information to the desired information.
3. The Stock Manager commits the change by pressing the corresponding button.

UC 11 - Authenticate User

Brief Description The System provides the opportunity to authenticate a User.
Involved Actors User
Precondition The System shows the Pick-up Shop website.
Trigger A User wants to authenticate himself.
Postcondition The User is authenticated.
Standard Process

1. The User is presented with a login form and enters his username and password.
2. The System checks the provided credentials and logs the User in.

Alternative or Exceptional Processes

- *In step 2: Wrong credentials*
  1. The User is presented with an error message.
  2. The User may try again until the authentication succeeds.

UC 12 - Create Customer

Brief Description The System provides the opportunity to create a new Customer account.
Involved Actors Customer
Precondition The Customer does not yet have a Customer account and the System is started.
Trigger A new Customer wants to create an account.
Postcondition A new Customer account is created and stored in the Inventory.
Standard Process

1. The System presents the Customer with a form to fill out, requesting all necessary information to create a new Customer account.
2. The Customer fills out the form and submits the information.
3. The System checks the provided 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 13 - View Customer Report

Brief Description The System provides an opportunity to generate reports about the purchases of a Customer.
Involved Actors Stock Manager
Precondition The store management website is available and the Stock Manager is authenticated.
Trigger The Stock Manager wants to see the report about a Customer.
Postcondition The report about the Customer has been generated and is displayed on the store management website.

Standard Process

1. The Stock Manager enters the Customer identifier and submits the request to create the report.
2. A report including all purchases made by this Customer is displayed.

5.4.2 Design of the Pick-up Shop

For adding the Pick-up Shop, the hybrid cloud-based variant had to be modified and extended to fit the needs arising from an online shop. In Fig. 5.11 the component diagram of the modified hybrid cloud-based variant is shown. The first addition is the implementation of a system for customers to register and log in. This functionality requires the *ServiceAdapter* to store the login information and additional data like credit card data and the customer's preferred store in the data store. Because the login functionality can also be used by the web frontend, the data stored for this purpose is divided up into user data used for authentication and customer data. The general user data contains information on the username, used credentials and roles of a user. The customer data contains the data only needed for customers and is linked to the corresponding user record.

The second addition is to include the mechanisms for the creation, modification and authentification of customers into the business logic tier. To this end the *Inventory* component is extended by a new *UserManager* component. This component implements the communication with the *ServiceAdapter* to retrieve, modify or create the user and customer data. The data is processed as needed by the *UserManager* component inside the *Application* component. Access to this functionality is provided via web service interfaces which extend the web service interfaces of the existing hybrid cloud-based variant. The two added web service interfaces include one interface for the authentication functionality in the *WebService::Inventory::LoginManager* component and one interface for other operations on users and customers in the *WebService::Inventory::UserManager* component.

These interfaces are used by the *PickupShop* component, which implements the frontend with which the customers interact. The *ShoppingCart* component keeps track of all items the customer wants to buy and is responsible for calculating the total price of these items. When the customer is done adding items to the *ShoppingCart* and proceeds, the sale is persisted by a wizard implemented in the *CheckOut* component. This component uses the external bank interface to debit the total price from the customers credit card. Only if this operation succeeds, the sale gets sent to the business logic and then persisted into the data store, as it is shown in Fig. 5.13. The *Inventory* component queries the available stock items of stores as well as general information about available stores and enterprises for the customer to pick up the ordered goods. The information queried is cached locally to minimize the number of queries to the business logic and the data store. This component is also responsible to propagate a sale event to the business logic.

An important aspect of the design is to make the system scale well due to the possibly high number of requests the Pick-Up Shop could be faced with. This is being addressed by only making calls from the Pick-Up Shop frontend to the business logic and the data store when

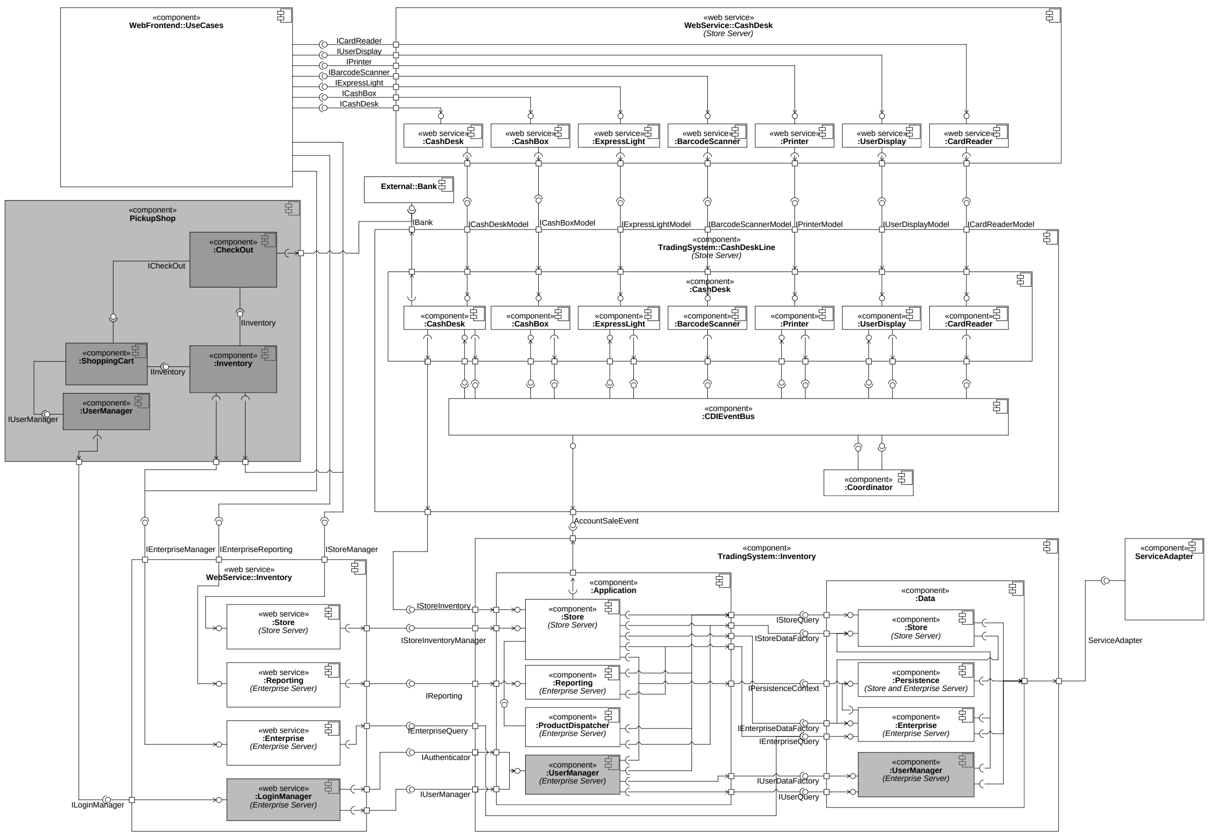


Figure 5.11: Hybrid Cloud-based Variant Component Structure After Adding the Pick-up Shop (Added Components are Highlighted).

absolutely necessary, therefore minimizing the additional load on these layers. Moreover, the login functionality, user data manipulation, store and item queries as well as the persisting of sales do not require the Pick-Up Shop nor any other component involved to rely on any kind of state other than the persisted state of the system in the data store. Therefore the Pick-Up Shop can be deployed on multiple machines simultaneously, enabling the use of load balancing mechanisms and therefore the possibility of horizontal scalability.

To give a behavioral overview of the added Pick-up Shop, the behavior during the Process Online Sale use case will be explained below. The customer interacts with the Pick-up Shop through a web frontend which passes the customers' commands to the correct underlying component and method in the component. The interactions with the web frontend is omitted in the following sequence diagrams and the corresponding actions are invoked directly on the components for simplification. As in Sec. 5.2, interface names are used to show that a concrete implementation of the interface is injected into the calling component via a dependency injection mechanism.

Fig. 5.13 shows the process of a customer choosing and buying products at the Pick-up Shop. The customer arrives at the Pick-up Shop and is required to select the store where s/he wants to pick up the products after the purchase. The selection of the store is initiated by the customer who is first asked to choose the enterprise and then to choose the specific store in that enterprise. To retrieve the enterprises and the stores, the called *IEnterpriseInformation* and *IStoreInformation* interfaces use *IEnterpriseQuery*. *IEnterpriseQuery* first looks for the requested information inside its internal cache and if the information is missing, queries the business logic through the *IEnterpriseManager* or *IStoreManager* web service interfaces. The customers' selections are then stored inside the SessionScoped implementations of the *IEnterpriseInformation* and *IStoreInformation* interfaces. Once the store was selected, the navigation element for selecting a store is deleted by a call to the *removeElement* method in the *NavigationMenu* component as shown in Fig. 5.12.

The customer is then presented with a list of all available stock items and may choose to add the ones s/he would like to purchase to his shopping cart. The list of stock items is retrieved through the *IInventory* interface and also use an internal cache. If the stock items are not yet stored in the cache they will be retrieved through the *IStoreQuery* interface which in turn also calls the *IStoreManager* web service interface in the business logic. To add a stock item to the cart, the stock item is passed to the SessionScoped implementation of the *IShoppingCart* interface which adds it to its internal data structures and computes the new price for the cart.

When the customer then proceeds to the checkout with his cart, the ConversationScoped *CheckOutWizard* starts a new checkout conversation and redirects the customer to a specific checkout site. If the customer is not already logged in, the customer has to log in before the checkout can be completed. The log in process is shown in Fig. 5.14. The customer enters his username and password in a log in form. These values are stored in the SessionScoped *UserLogin* component which is responsible for handling the login process. The login process itself is handled via a login module which plugs into GlassFish's built-in login functions. To log in, the *UserLogin* component calls GlassFish's log in function which in turn calls the *authenticateUser* method in the *LogicServiceLoginModule*. This method calls the *ILoginManager* web service

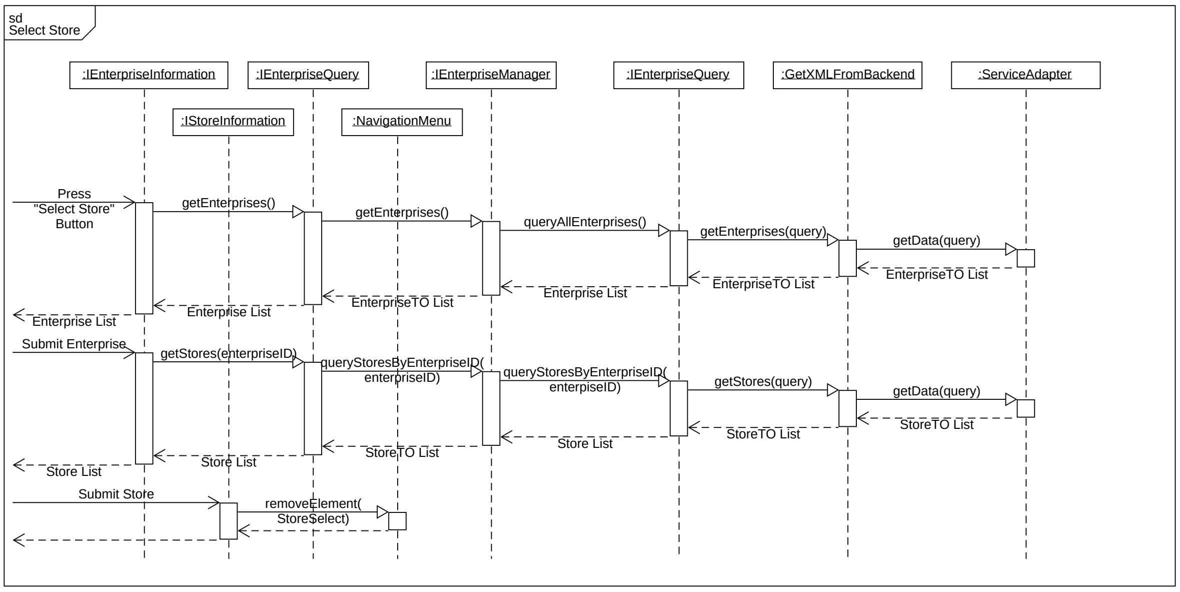


Figure 5.12: Sequence Diagram of Selecting a Store in the Pick-up Shop.

interface in the business logic, which uses the *IAuthenticator* interface to perform the check. To this end, the *IUserQuery* queries the data store for the user record and checks if the provided credentials match the stored credentials. If the credentials match, the authentication was successful.

Once logged in, the customer has to select the credit card which should be charged with the purchase. If the customer decides to add a new card to his account, s/he is presented with a form where s/he enters his card information. This is shown in Fig. 5.15. Once the information is entered it is stored in the *AddCreditCardWizard* component and the customer account is updated by first adding the new card to the currently loaded *Customer* component which contains all information on the customer. This updated *Customer* gets updated in the data store as well by calling the *IInventory* interface which hands the updated information to the *ICustomerQuery* which in turn transforms the data and sends it to the business logic through the *ILoginManager* web service interface. The web service sends the data on to the data store through the *IUserManager* interface.

Once the credit card is selected, the selected card is stored in the *CheckOutDetails* and the customer has to enter her/his PIN for the credit card which is also stored. When the customer has reviewed her/his order and proceeds, the card data and the PIN are retrieved from the *CheckOutDetails* and are used to obtain a transaction id from the *IBank* interface. If the transaction id is not null the card is debited by another call to *IBank*. In case there was an error, the customer is presented with an error message and may try again.

If the transaction succeeds the *IInventory* interface is called and calls the *IStoreQuery* to loop over all the items in the shopping cart. For each item a *SaleTO* is generated and passed to the

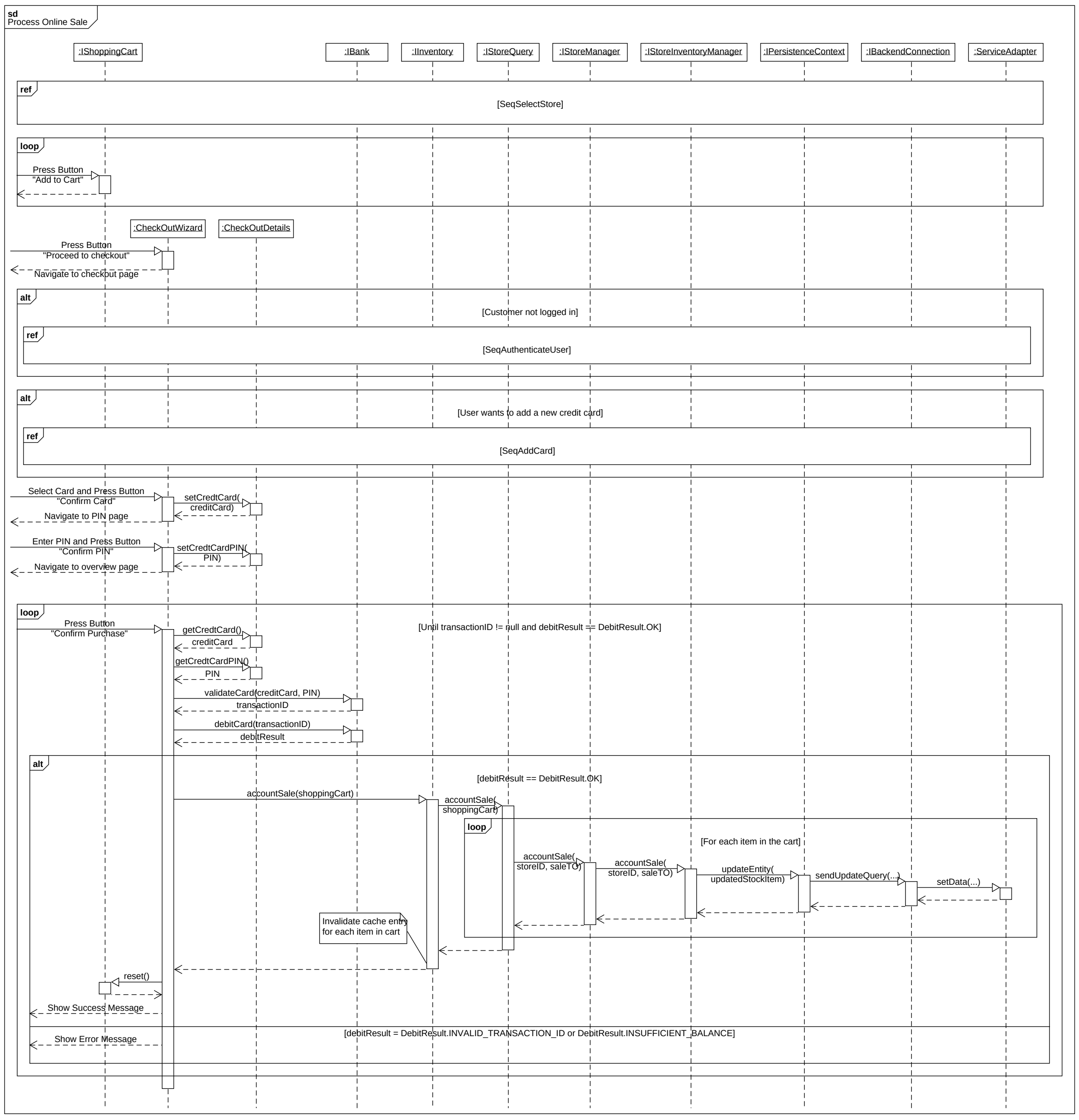

sd
Process Online Sale
:IShoppingCart
:IBank
:IInventory
:IStoreQuery
:IStoreManager
:IStoreInventoryManager
:IPersistenceContext
:IBackendConnection
:ServiceAdapter
ref
[SeqSelectStore]
loop
Press Button "Add to Cart"
:CheckOutWizard
:CheckOutDetails
Press Button "Proceed to checkout"
Navigate to checkout page
alt
[Customer not logged in]
ref
[SeqAuthenticateUser]
alt
[User wants to add a new credit card]
ref
[SeqAddCard]
Select Card and Press Button "Confirm Card"
setCredtCard(creditCard)
Navigate to PIN page
Enter PIN and Press Button "Confirm PIN"
setCredtCardPIN(PIN)
Navigate to overview page
loop
Press Button "Confirm Purchase"
[Until transactionID != null and debitResult == DebitResult.OK]
getCredtCard()
creditCard
getCredtCardPIN()
PIN
validateCard(creditCard, PIN)
transactionID
debitCard(transactionID)
debitResult
alt
[debitResult == DebitResult.OK]
accountSale(shoppingCart)
accountSale(shoppingCart)
loop
[For each item in the cart]
accountSale(storeID, saleTO)
accountSale(storeID, saleTO)
updateEntity(updatedStockItem)
sendUpdateQuery(...)
setData(...)
Invalidate cache entry for each item in cart
reset()
Show Success Message
[debitResult = DebitResult.INVALID_TRANSACTION_ID or DebitResult.INSUFFICIENT_BALANCE]
Show Error Message

Figure 5.13: Sequence Diagram of the Process Online Sale Use Case of the Pick-up Shop.

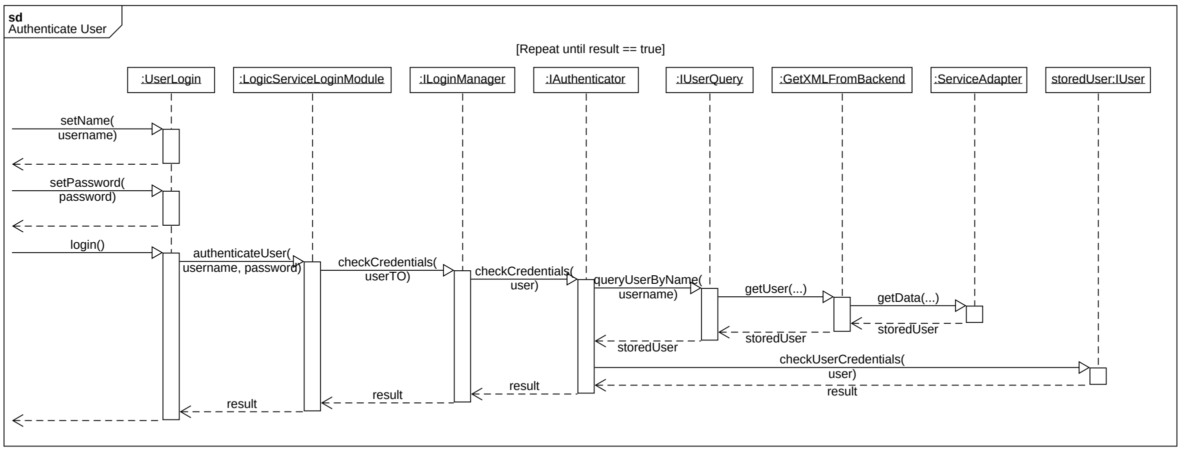


Figure 5.14: Sequence Diagram of Authenticating a User in the Pick-up Shop.

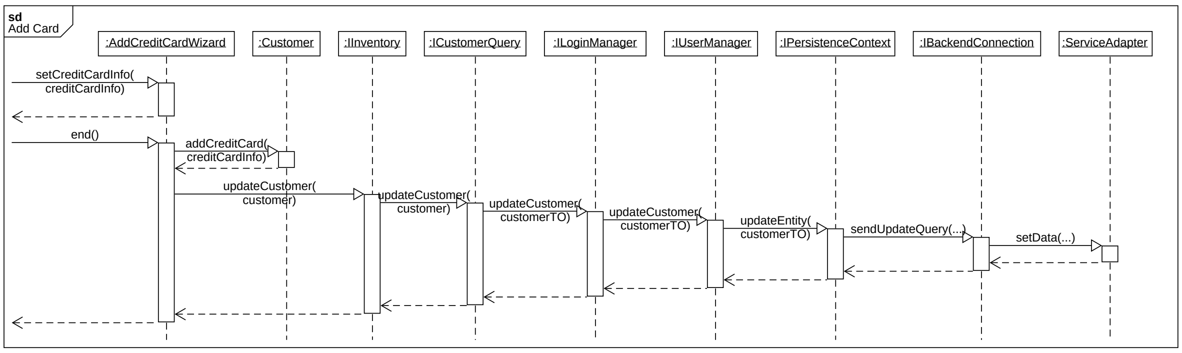

sd
Add Card
:AddCreditCardWizard
:Customer
:IInventory
:ICustomerQuery
:ILoginManager
:IUserManager
:IPersistenceContext
:IBackendConnection
:ServiceAdapter
setCreditCardInfo(creditCardInfo)
end()
addCreditCard(creditCardInfo)
updateCustomer(customer)
updateCustomer(customer)
updateCustomer(customerTO)
updateCustomer(customerTO)
updateEntity(customerTO)
sendUpdateQuery(...)
setData(...)

Figure 5.15: Sequence Diagram of Adding a new Credit Card of a Customer in the Pick-up Shop.

business logic via the *IStoreManager* web service interface to update the inventory and account the sale. After the sale is completed, the internal cache entry in the *IInventory* for each item that was sold is invalidated and the shopping cart is reset. The customer is presented with a success message at the end.

5.5 Adding a Service-Adapter

The Service-Adapter framework is introduced in the evolution scenario Adding a Service-Adapter to facilitate the extension of CoCoME. The framework provides an additional layer to extend CoCoME by REST services. Before, extension was only possible by introducing new services at code-base level. Consequently, new contributers had to understand most of the code-base in order to extend CoCoME in an appropriated way. Since this requires a lot of upfront effort due to the complexity of the code-base, a new extension approach became necessary. Two technologies are available to design the Service-Adapter.

- *Remote-Object interfaces*; are EJB interfaces which act like a facades to external clients. They provide functionallity on CoCoME and can be injected by different other projects using JNDI technology.
- *REST services*; completely encapsulate CoCoME to clients and provide well-defined service interfaces.

On the one hand remote objects are easy to implement. JEE provides a very good framework using annotations. Clients using this interfaces implement against a Java interface which is very handy. On the other hand this approach forces clients to inject the interfaces. Clients are constrained on technology (in our case JEE).

A REST service layer allows for coupling extensions loosely to CoCoME. Clients can be developed using different kind of technologies. Further, clients only have to understand the service rather than the whole code-base of CoCoME. Using this approach, however, a REST service framework must be developed and coupled to CoCoME.

The taken solution is a mix of both. To be addressable by some kind of REST service, CoCoME exposes internals by using EJB interfaces. These interfaces provide a facade to an adapter which itself will provide the infrastructure for the REST service framework. So called Service-Provider will itself provide access to different kind of services implemented on different platforms and nodes. Since the adapter providing the REST infrastructure (Service-Adapter) has its own code-base, it can be deployed on a separate node. The architecture of the Service-Adapter and already implemented services are explained in the following.

Extension Concept Overview An overview of the Service-Adapter extension is given in Fig. 5.16. The ServiceAdapter is located between the RESTful services and the actual CoCoME implementation. The ServiceAdapter is composed of a set of ServiceProvider which encapsulate a domain within the system. Each ServiceProvider provides a set of Service, the actual functionality. The set of ServiceProvider can be browsed by the ServiceProviderCatalog. Fig. 5.17 shows the different entities of the extension concept by starting from the ServiceProviderCatalog up to the actual Service. Each kind of Service is implemented within a specific ServiceProvider. A ServiceProvider can already be available or has to be implemented for a given Service. The ServiceProvider acts like an extension point from where the services are reachable and where additional services can be added, if they fit in the domain of the ServiceProvider.

In the current implementation, ServiceProvider and Service are implemented as Java Servlets. They use remote Enterprise Java Beans in the CoCoME backend to provide access to the desired functionality. In case of ServiceProviderDatabase, the DatabaseAccess bean was designed to provide access to the database used by the CoCoME implementation.

For the client tier the actual used technology becomes irrelevant. The only important information is the protocol of communication used by the desired Service. The actual communication, however, is done via HTTP methods like GET, POST, PUT and DELETE.

Facades/Extension Points The remote objects in the CoCoME backend implementation are used as extension points. They are designed to meet the needs of a specific Service or even a set of Service in the ServiceProvider.

Navigation The Service-Provider framework is composed as depict in Fig. 5.17. This framework is used to realize the navigation. Each ServiceProviderCatalog, ServiceProvider and Service has its own URL where further navigation or the actual HTTP request URL can be used.

An overview of the navigation in the Service-Provider framework is given in Fig. 5.18. The figure shows how a request to the Service-Adapter is scheduled within the Service-Provider framework.

The response of the Service-Adapter is depicted in Fig. 5.19 and discussed in the following. The figure shows the two Service-Providers – BookSale and Database.

Service-Provider Database In order to access the database of CoCoME, the Service-Provider cocome.cloud.sa.serviceprovider.impl.ServiceProviderDatabase offers three services as depicted in Fig. 5.20.

- GetData
  Get data from the database. Query-Select
- SetData
  Update or insert data into the database. Query-Update and Query-Insert
- Schemas
  Provide schemas in order to use GetData and SetData.

GetData Service In order to get data from the database, select queries are used as URL parameters. The following setup is needed to make a request:

- HTTP-GET request
- Content-Type is either application/xml or application/csv
- query.select as url parameters

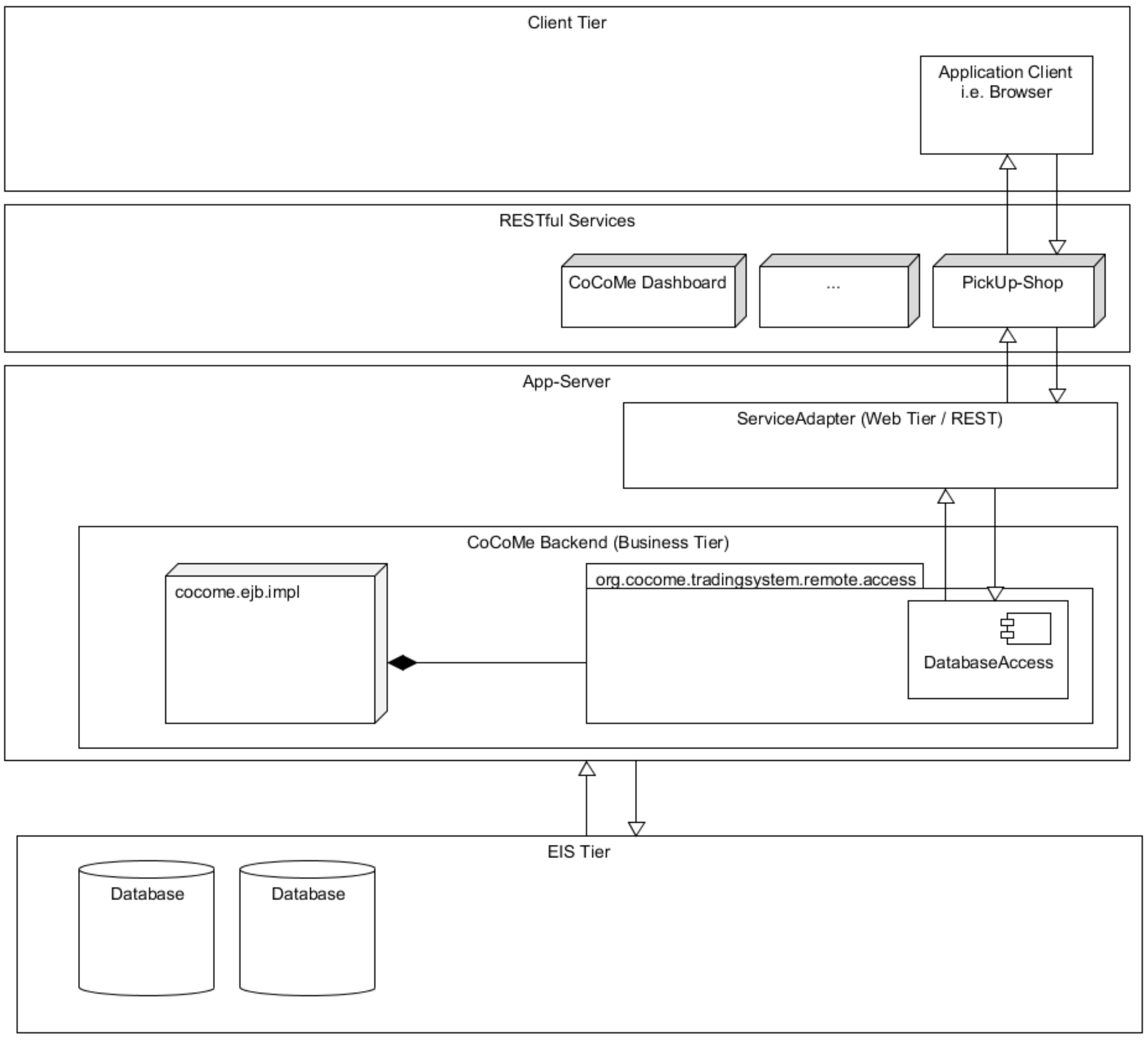

Client Tier
Application Client i.e. Browser
RESTful Services
CoCoMe Dashboard
...
PickUp-Shop
App-Server
ServiceAdapter (Web Tier / REST)
CoCoMe Backend (Business Tier)
cocome.ejb.impl
org.cocome.tradingsystem.remote.access
DatabaseAccess
EIS Tier
Database
Database

Figure 5.16: Adding a Service-Adapter to CoCoME

Fig. 5.22 depicts how the request is handled. A Get request containing the query string is sent to the ServiceProviderDatabaseServlet. The query is forwarded to the DatabaseAccess bean which provides the requested information from the database or responds an error. The Servlet provides the requested information in XML or CSV format or responds an error message.

Query Language Queries are passed by the URL parameters. The listings show the syntax and an application example of the query.

Syntax:

```
query.select=entity.type=[Entity];[ConstrainA];[ConstrainsB];...
```

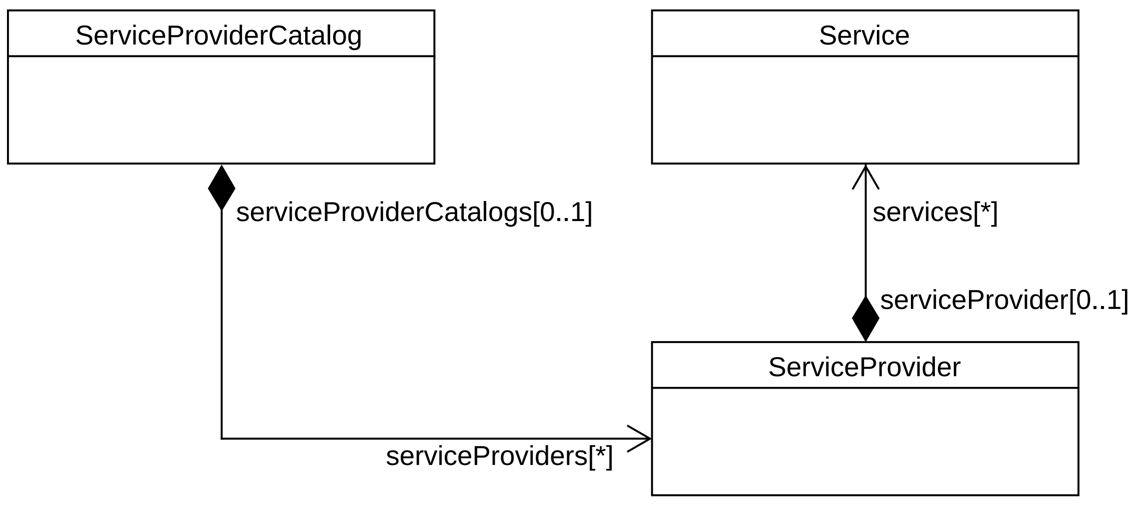


Figure 5.17: The Service-Provider Framework

Example:

```
query.select=entity.type=Product;product.purchasePrice=<100;product.name=LIKE
'T*'
```

The example shows how to apply the query to get all products which price is less than or equal to 100 and the name of the product starts with the letter T. For building constrains, the Java Persistence Query Language (JPQL) is used. All constrains are logically composed with AND. If OR is required, two different requests have to be applied. The constrains available depend on the entity. They are accessed with the dot-operator.

Joins of tables are done automatically if the corresponding entity has a field of the required table. For instance, the entity Product has a field referring to the entity ProductSupplier. A constrain which choses all ProductSupplier having a name starting with A looks like:

```
product.supplier.name=LIKE 'A*'
```

In the listing the tables Product and ProductSupplier are joined automatically using the JPQL. Therefore constrains can be passed like using JPQL.

SetData The SetData service supports two kind of requests:

- query.insert
  HTTP-POST request
- query.update
  HTTP-PUT request

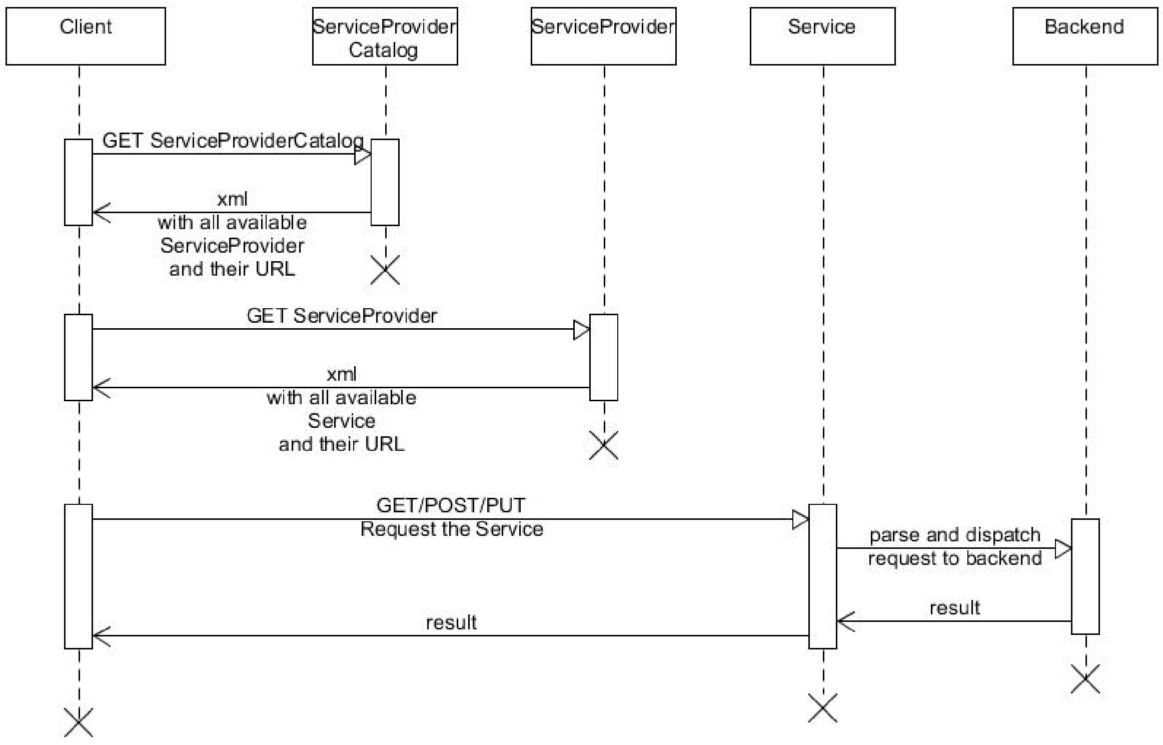

Client
ServiceProvider Catalog
ServiceProvider
Service
Backend
GET ServiceProviderCatalog
xml with all available ServiceProvider and their URL
GET ServiceProvider
xml with all available Service and their URL
GET/POST/PUT Request the Service
parse and dispatch request to backend
result
result

Figure 5.18: Generic Navigation in the Service-Provider Framework

There are two formats to send the data. One is the XML format using the table schema. The other is the CSV format. The content type must be defined either application/xml or application/csv. The default is application/xml. The syntax is specified in the listings.

```
query.insert=[Entity]
```

for inserting and

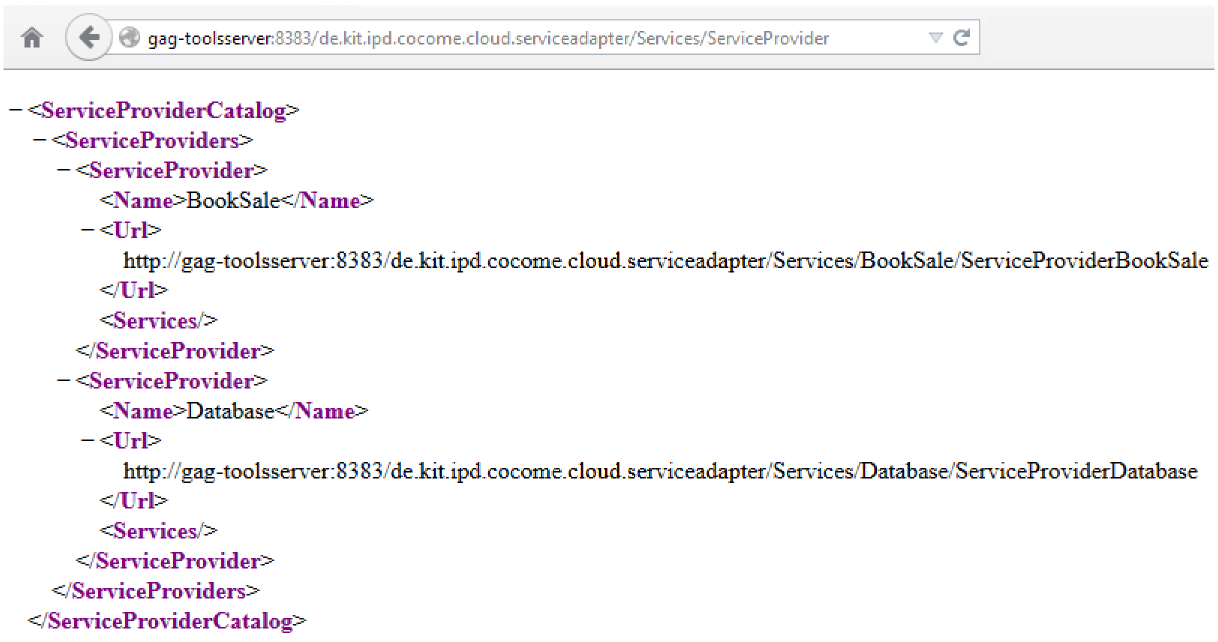
```
query.update=[Entity]
```

for updating. Data is passed by the data-stream.

After the POST/PUT, the Service-Adapter will answer and provide a notification whether the action was successfully or not. This message is provided in a XML file. The schema is the Message.xsd.

Service-Provider BookSale The BookSale Service-Provider offers two services as depicted in Fig. 5.21.

- GetAllProducts
  Get all products from the database


gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/ServiceProvider

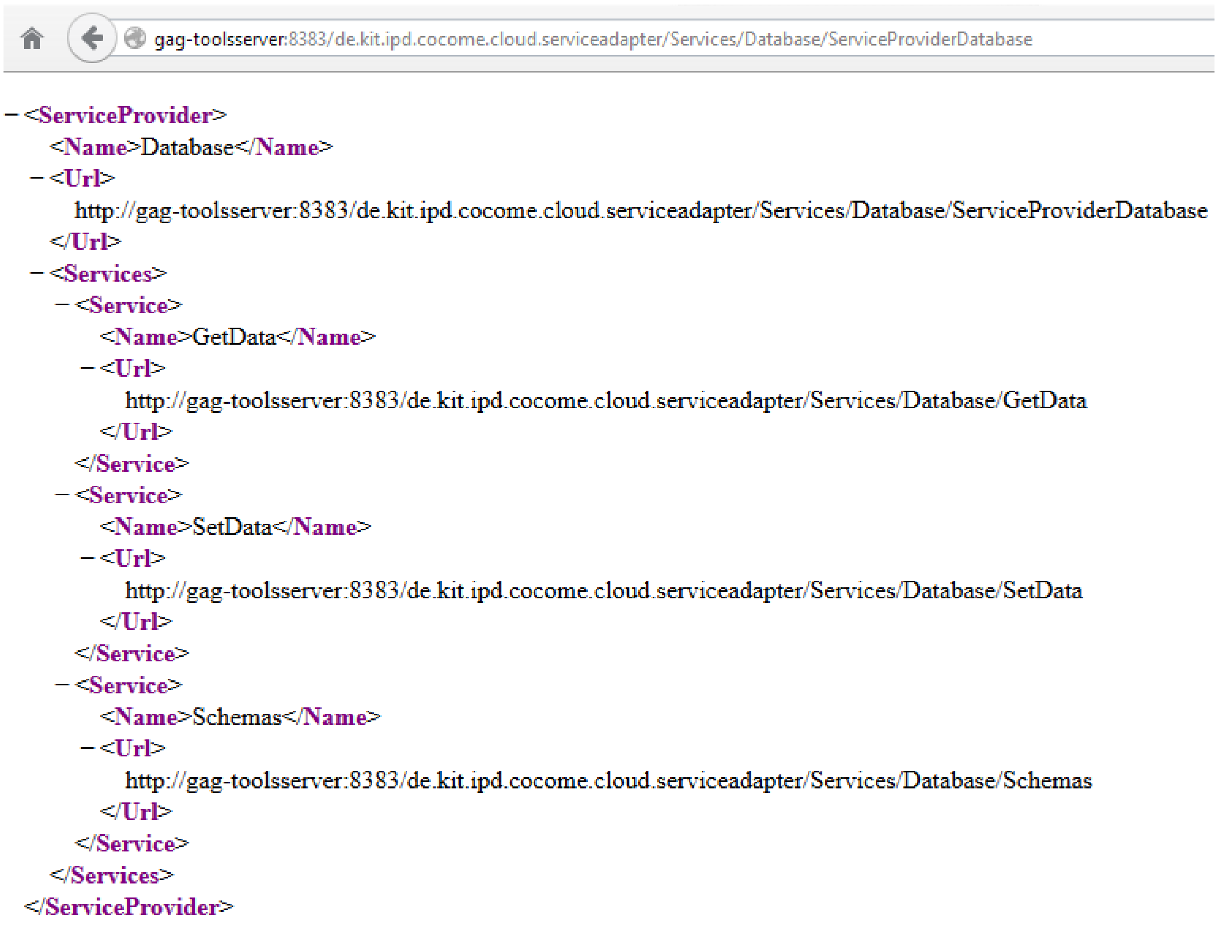
```
– <ServiceProviderCatalog>
  – <ServiceProviders>
    – <ServiceProvider>
        <Name>BookSale</Name>
      – <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/BookSale/ServiceProviderBookSale
        </Url>
        <Services/>
      </ServiceProvider>
    – <ServiceProvider>
        <Name>Database</Name>
      – <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/ServiceProviderDatabase
        </Url>
        <Services/>
      </ServiceProvider>
    </ServiceProviders>
  </ServiceProviderCatalog>
```

Figure 5.19: Response ServiceAdapter

- OrderProducts
  Order the given products

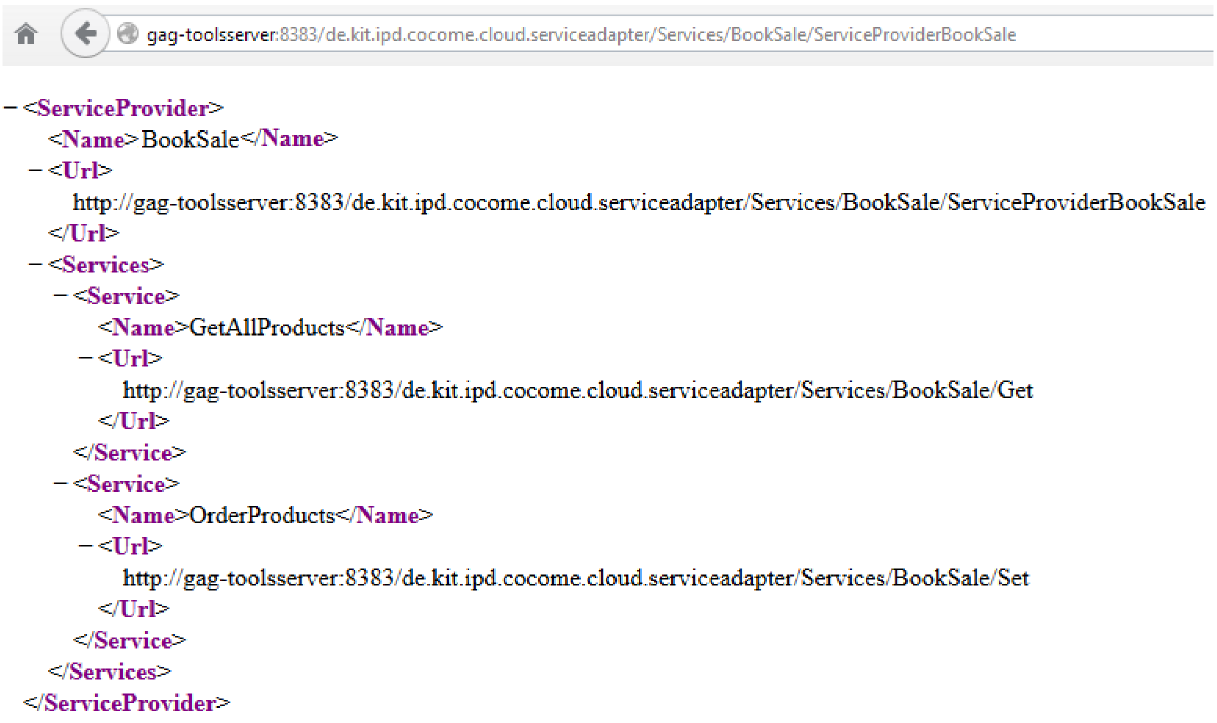
The Service-Provider uses the GetData service and the SetData service from the Service-ProviderDatabase. It demonstrates that the framework can be used to compose new services based on already available services.

The OrderProducts service requires input data from the input stream. This data should contain the following information: ProductOrderId, StoreId, ProductBarcode, OrderDeliveryDate, OrderOrderingDate, OrderAmount. The service saves the information into the database.


gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/ServiceProviderDatabase

```
− <ServiceProvider>
    <Name>Database</Name>
  − <Url>
      http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/ServiceProviderDatabase
    </Url>
  − <Services>
    − <Service>
        <Name>GetData</Name>
      − <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/GetData
        </Url>
      </Service>
    − <Service>
        <Name>SetData</Name>
      − <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/SetData
        </Url>
      </Service>
    − <Service>
        <Name>Schemas</Name>
      − <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/Schemas
        </Url>
      </Service>
    </Services>
  </ServiceProvider>
```

Figure 5.20: Response ServiceProviderDatabase


gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/BookSale/ServiceProviderBookSale

```
– <ServiceProvider>
    <Name>BookSale</Name>
  – <Url>
      http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/BookSale/ServiceProviderBookSale
    </Url>
  – <Services>
    – <Service>
        <Name>GetAllProducts</Name>
      – <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/BookSale/Get
        </Url>
      </Service>
    – <Service>
        <Name>OrderProducts</Name>
      – <Url>
          http://gag-toolsserver:8383/de.kit.ipd.cocome.cloud.serviceadapter/Services/BookSale/Set
        </Url>
      </Service>
    </Services>
  </ServiceProvider>
```

Figure 5.21: Response ServiceProviderBookSale

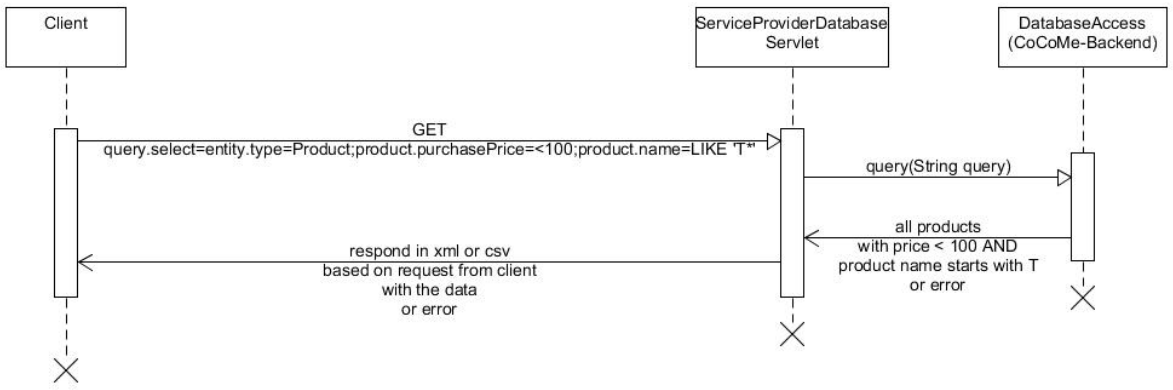

Client
ServiceProviderDatabase Servlet
DatabaseAccess (CoCoMe-Backend)
GET query.select=entity.type=Product;product.purchasePrice=<100;product.name=LIKE 'T*'
query(String query)
all products with price < 100 AND product name starts with T or error
respond in xml or csv based on request from client with the data or error

Figure 5.22: Get-Request on ServiceProviderDatabase

6 Life-Cycle

We developed a set of sample activities typical in information system evolution and arranged them in life-cycle form (cf. Fig. 6.1) to cope with aforementioned evolution scenarios.

An iteration in the life-cycle starts with a change request. This is either an external request or caused by upcoming issues at run-time. Examples of change requests are the need for migrating the system to the cloud (Sec. 4.2.1) or for a pick-up shop (Sec. 4.2.2). Design decisions for system modification are made to cope with the request. The decisions are documented and a static quality analysis is conducted to identify quality impacts at design-time. The system design is adapted (or initially specified) and implemented in source code. After deploying the (modified) application, a dynamic quality analysis is conducted for the running system. If an upcoming quality issue has been identified, two ways of modification are distinguished – automated adaptation and manual evolution. Automated adaptation refers to the capability of the system to adapt itself. An example of automated adaptation is migrating the database from one cloud provider to another (Sec. 4.2.3). Manual evolution triggers another iteration in the life-cycle.

Various alternatives and variations to the proposed life-cycle are possible. However, the proposed life-cycle is one representative of a research basis suitable to conduct empirical studies on software evolution as it covers to the evolution characteristics of Demeyer et al. [2]. It comprises artifacts that correspond to all phases in the system's life-cycle (Life-cycle characteristic). It covers iterations and increments in the development process (evolution characteristic). It provides a concrete setting to qualify the application domain (i.e. e-commerce), problem domain (i.e. web-based system) and solution domain (e.g., architecture, source code) of the case (domain characteristic). It supports evaluating the kinds of tools necessary to replicate the case, such as implementation/design languages, operating system, or development/CASE environments (tool characteristic).

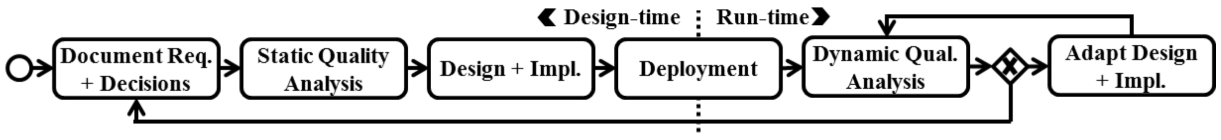


Figure 6.1: Overview of the CoCoMEP Life-Cycle [7]

7 Implementation of Evolution Scenarios

This chapter describes implementation and configuration details for realizing the hybrid cloud-based variante of CoCoME and the corresponding evolution scenarios. Sec. 7.1 provides insights on the implementation of the Platform Migration scenario. The implementation of the scenario Adding a Pick-up Shop is described in Sec. 7.2. Sec. 7.3 explains the API of the service adapter.

7.1 Platform Migration

The Platform Migration evolution scenario transfers the Plain Java variant to the hybrid cloud-based variant by introducing web service technologies. The implementation of the hybrid cloud-based variant was done using Eclipse for Java EE and GlassFish 3.1 was used as application server to deploy the application. Java EE version 6 was used because GlassFish 3.1 is only compatible with this specific version. To automate the build process and the deployment of the business logic and the presentation layer, Maven was used.

To deploy the *WebFrontend* and the business logic with its web service wrappers, the Maven install procedure uses a plug-in to directly deploy the application to the server. For this purpose, the presentation layer is bundled into a .war archive from the cloud-web-frontend project. The business logic is located in the cloud-logic-service project and consists of eight sub-projects, java-utils, cloud-logic-core-api, cloud-logic-core-impl, cloud-logic-core-services, cloud-registry-service, cloud-registry-client, cloud-enterprise-logic and cloud-store-logic. An overview of the project structure and dependencies in the cloud-logic-service project is shown in a package diagram in Fig. 7.1. The *import* relationship between packages is defined as a relationship in which the importing namespace adds the names of the members of the imported package to its own namespace. In contrast, the *merge* relationship indicates that the contents of both packages are to be combined, similar to Generalization [14].

Classes needed by the enterprise and the store server like transfer objects and interfaces of components are defined in the cloud-logic-core-api project and implementations are provided in the cloud-logic-core-impl project which depends on the java-utils project. In the java-utils project some classes are implemented to ease the communication with the service adapter. The cloud-logic-core-services project provides interface definitions for the core web services as well as proxy classes to ease the use of these services for the clients. Any client can access the services by including the cloud-logic-core-services project and using the proxy classes. The registry where the addresses of available store and enterprise servers are located is implemented in the cloud-registry-service project. To look up an endpoint registered in the registry, a client is implemented in the cloud-registry-client project.

The cloud-enterprise-logic and cloud-store-logic projects depend on the above projects and

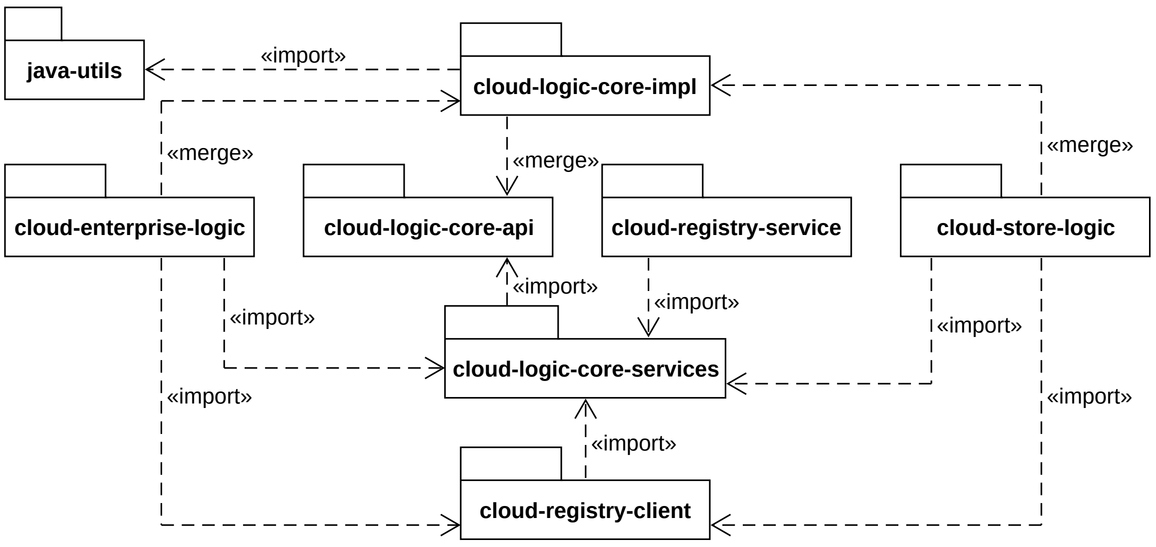


Figure 7.1: Overview of the project structure in the cloud-logic-service project of the hybrid-cloud variant.

implement the business logic for the enterprise and store server. The cloud-enterprise-logic project consists of three subprojects, enterprise-logic-ear, enterprise-logic-ejb and enterprise-logic-webservice. The ejb project contains the enterprise beans that contain the business logic and the webservice project contains the web service wrappers. These projects are bundled together into a deployable .ear file by the ear project.

The cloud-store-logic has a similar structure with the subprojects store-logic-ear, store-logic-ejb, store-logic-webservice and store-logic-integrationtest. Integration tests for the store server are provided inside the store-logic-integrationtest project and will be executed when using the install command of Maven on the cloud-store-logic project.

Because of this structure it is possible to deploy every layer on a different application server or cluster of application servers. Also, all layers use web service technologies to communicate with each other. This enables the application to be deployed in a cloud environment. Moreover, it is possible to choose whether to use several store servers and an enterprise server or to use a single server because the implementation of the business logic is capable of handling both deployment options.

7.2 Adding a Pick-up Shop

Adding the Pick-Up Shop required to extend the hybrid-cloud variant, therefore the implementation was also done using Eclipse and was updated to work with Java EE version 7 in conjunction with GlassFish 4.1. The hybrid-cloud variant was refactored so that it now includes the cloud-logic-webservicestubs project. This project provides the generated Java stubs which provide access to the web service interfaces and is included in the Pick-Up Shop and the *WebFrontend*.

The Pick-Up Shop itself consists of the cloud-pickup-shop project which implements the Pick-Up Shop's frontend and is bundled into a .war file by the Maven install procedure before it is being deployed . The cloud-auth-provider project is an OSGi bundle and provides the functionality for authenticating, logging in and logging out users and is used by the Pick-Up Shop.

7.3 Adding a Service-Adapter

This section explains implementation details of the Service-Adapter. First we give an overview of the Service-Adapter API in Sec. 7.3.1 before we describe used libraries and frameworks in Sec. 7.3.2.

7.3.1 Service-Adapter API

An overview of the Service-Adapter API is given in Fig. 7.2. The Service-Adapter offers two different data formats – CSV and XML. For the CSV format (comma-separated) a semicolon is used as delimiter. First line is the column names. Each line is divided by a break line. This format can be used for inserting update data. When it comes to selecting data, the server will always respond in XML as top-level format. Single nodes, especially the node providing the result of the query can be defined in the request. It can be chosen as XML or CSV. The XML format of the result is explained in Sec. 7.3.2. It is based on a custom library which is serialized and unserialized using the JAXB framework of Java. For all formats libraries are provided.

XML schemas are provided as service of ServiceProviderDatabase by the given url:
`http://host:port/de.kit.ipd.cocome.cloud.serviceadapter/Services/Database/Schemas`

Internally the Table-Framework is used as first/last step in representation of data provided to the client or from internal logic. We developed specific libraries for parsing which are available in the project de.kit.ipd.java.utils.

7.3.2 Libraries and Frameworks

In the following, simple class names are used for simplify the text. Class names are capitalized. In the Fig. 7.3 and 7.4 the full qualified names can be extracted. In case of unclear reference, the full qualified name is used.

The **Lexer-Parser-Framework** depicted in Fig. 7.3 provides a structure to implement simple lexers and parsers. The lexer scans raw text and provides events, which can be used by the parser to build a model of the raw text. In case of the CSVParser the model is represented by

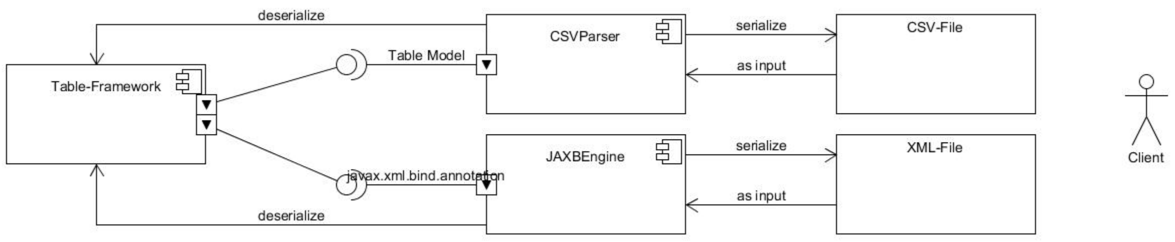


Figure 7.2: Overview of Format Parsing for Client Communication

the Table class, since a CSV file is basically a table. The lexer has a StateMachine which should be able to detected different patterns in the text and switch from one State to another. Based on the state machine there are different end states which can be hooked up using the LexerVisitor. In case of the CSVParser, the CSVParser implements the LexerVisitor interface. Obviously it depends on the implementation of the StateMachine how the Parser has to understand different end states. The process of designing a new Parser comprises the following four steps:

- Designing a state machine and optimize it for minimal states.
- Implement the different States using the State interface.
- Implementing the StateMachine using different States. Therefore, the AbstractStateMachine can be used as base class to extend from. The CharStreamStateMachine is an actual implementation of the StateMachine to parse text in the first place. Using this CharStreamStateMachine, only the States have to be implemented.
- Implement the Parser using an own designed model. The Parser itself would provide the model to the client.

The **CSVParser** uses the Lexer-Parser-Framework to realize a parser for CSV files. The following snippets show how to use the parser to build and parse a CSV file.

Build a CSV file:

```
CSVParser csvparser = new CSVParser();
Table<String> table = new Table<>();
... fill table
csvparser.setModel(table);
String csvFileContent = csvparser.toString();
```

Parse CSV file:

```
(Get String or InputStream. Here named as content)
CSVParser csvparser = new CSVParser();
csvparser.parse(content);
Table<String> table = csvparser.getModel();
```

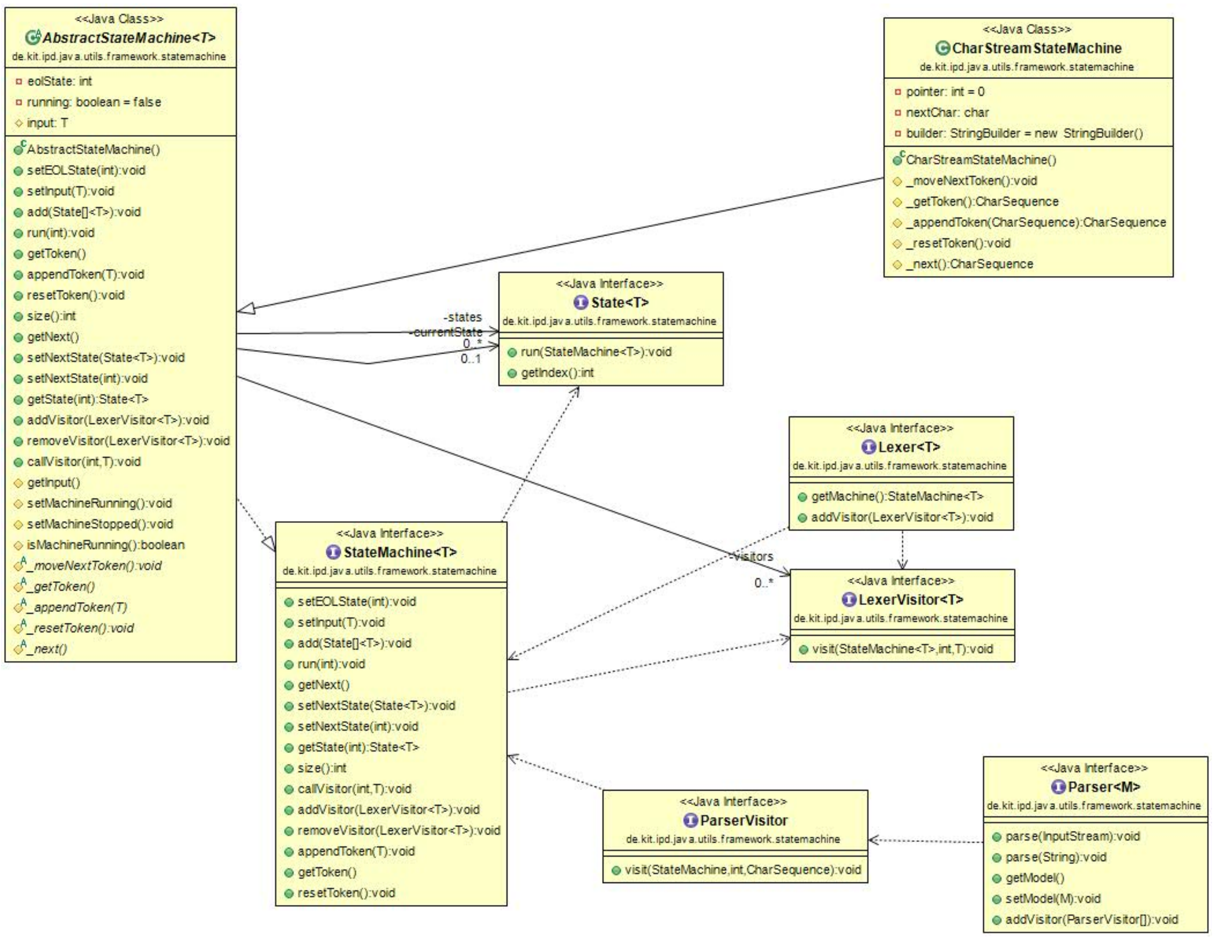
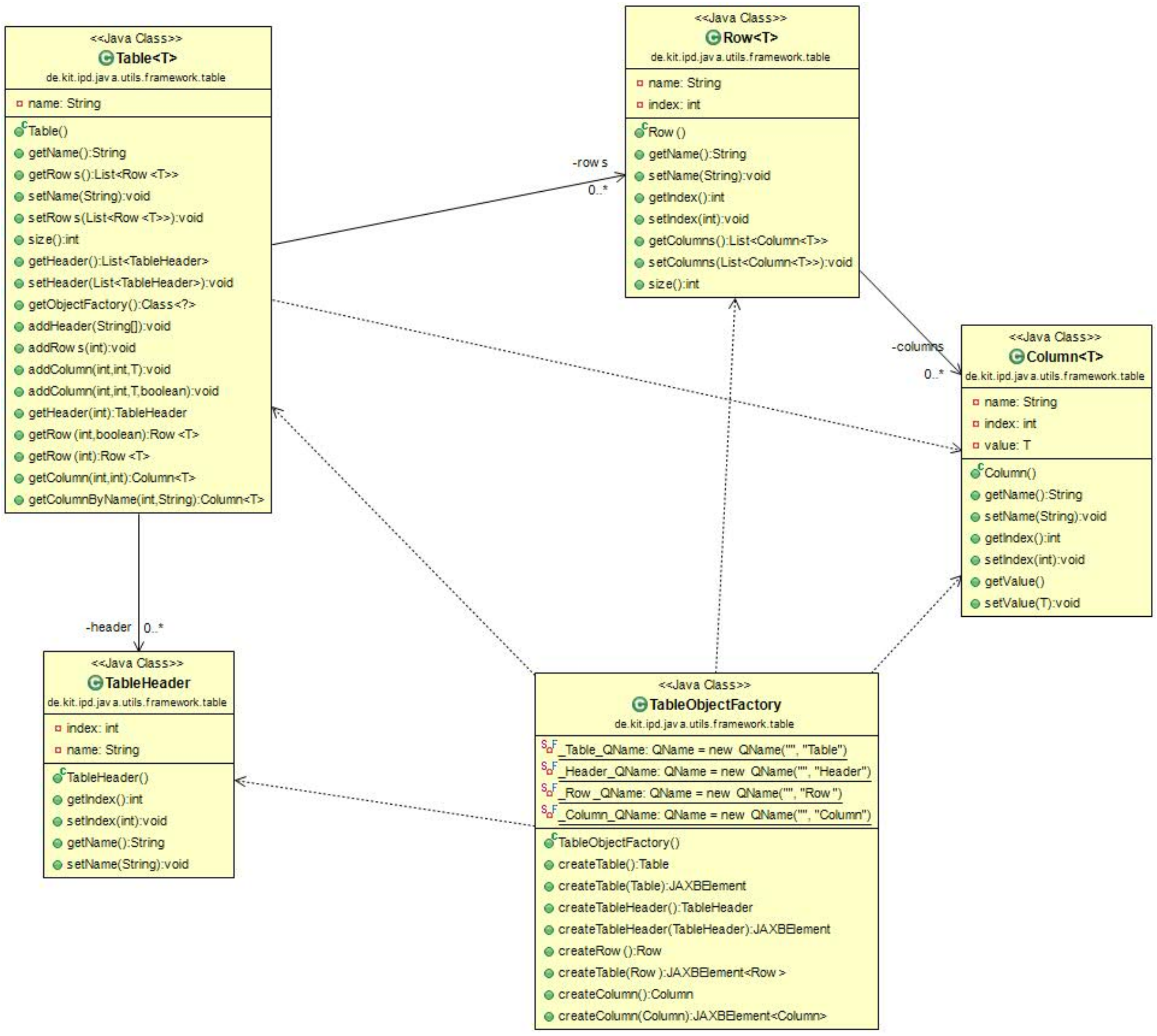


Figure 7.3: The Lexer-Parser-Framework

The **Table-Framework** is used to represent a CSV file as a model. It is a simple framework consisting of the classes shown in Fig. 7.4. The basic idea is as follows;

The Column class holds the actual value and is embedded in the Row class which is holding all the columns within a row. The Row class itself is embedded in the Table class which provides a method to get Row objects or even some convenient methods to get a Column directly. Hence one gets the row first and from there the column. Row and Column have different attributes to facilitate the search, e.g. for name. The Table itself has also TableHeaders. These represent the header of the CSV file. The API is intuitive providing getter- and setter-methods.



<<Java Class>>
Table<T>
de.kit.ipd.java.utils.framework.table
name: String
Table()
getName():String
getRows():List<Row<T>>
setName(String):void
setRows(List<Row<T>>):void
size():int
getHeader():List<TableHeader>
setHeader(List<TableHeader>):void
getObjectFactory():Class<?>
addHeader(String[]):void
addRows(int):void
addColumn(int,int,T):void
addColumn(int,int,T,boolean):void
getHeader(int):TableHeader
getRow(int,boolean):Row<T>
getRow(int):Row<T>
getColumn(int,int):Column<T>
getColumnByName(int,String):Column<T>
-rows
0..*
<<Java Class>>
Row<T>
de.kit.ipd.java.utils.framework.table
name: String
index: int
Row()
getName():String
setName(String):void
getIndex():int
setIndex(int):void
getColumns():List<Column<T>>
setColumns(List<Column<T>>):void
size():int
-columns
0..*
<<Java Class>>
Column<T>
de.kit.ipd.java.utils.framework.table
name: String
index: int
value: T
Column()
getName():String
setName(String):void
getIndex():int
setIndex(int):void
getValue()
setValue(T):void
-header
0..*
<<Java Class>>
TableHeader
de.kit.ipd.java.utils.framework.table
index: int
name: String
TableHeader()
getIndex():int
setIndex(int):void
getName():String
setName(String):void
<<Java Class>>
TableObjectFactory
de.kit.ipd.java.utils.framework.table
_Table_QName: QName = new QName("", "Table")
_Header_QName: QName = new QName("", "Header")
_Row_QName: QName = new QName("", "Row")
_Column_QName: QName = new QName("", "Column")
TableObjectFactory()
createTable():Table
createTable(Table):JAXBElement
createTableHeader():TableHeader
createTableHeader(TableHeader):JAXBElement
createRow():Row
createTable(Row):JAXBElement<Row>
createColumn():Column
createColumn(Column):JAXBElement<Column>

Figure 7.4: The Table-Framework

8 Conclusion

In this technical report we gave an overview of the CoCoME platform for collaborative empirical research on information system evolution. The platform consists of the three parts evolution subject, evolution scenario, and evolution life-cycle. The CoCoME system serves as an evolution subject from which we described three variants – Plain Java, service-oriented, and hybrid cloud-based variant. For each variant we present detailed architecture and design diagrams. We present several evolution scenario that specify changes to the evolution subject. An evolution life-cycle is described to integrate activities and their relationships required to implement the proposed evolution scenarios.

In the future, the subject CoCoME will be further modified to create new and evolve existing artifacts by new evolution scenarios such as the introduction of mobile clients and the usage of micro services.

Bibliography

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, architectures and applications.* Data-centric systems and applications. Springer, Berlin, 2004.

[2] S. Demeyer, T. Mens, and M. Wermelinger. Towards a Software Evolution Benchmark. In *4th International Workshop on Principles of Software Evolution*, pages 174–177. ACM, 2001.

[3] R. Fielding. A little rest and relaxation. In *The International Conference on Java Technology (JAZOON07), Zurich, Switzerland*, 2007.

[4] U. Goltz, R. H. Reussner, M. Goedicke, W. Hasselbring, L. Märtin, and B. Vogel-Heuser. Design for future: managed software evolution. *Computer Science - Research and Development*, 30(3):321–331, 2014.

[5] W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. iObserve: integrated observation and modeling techniques to support adaptation and evolution. Technical Report 1309, CAU Kiel, 2013.

[6] R. Heinrich. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):13–22, 2016.

[7] R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. The CoCoME platform: A research note on empirical studies in information system evolution. *International Journal of Software Engineering and Knowledge Engineering*, 25(09&10):1715–1720, 2015.

[8] R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. A platform for empirical research on information system evolution. In *27th International Conference on Software Engineering and Knowledge Engineering*, pages 415–420, 2015.

[9] R. Heinrich, K. Rostami, J. Stammel, T. Knapp, and R. Reussner. Architecture-based analysis of changes in information system evolution. In *Softwaretechnik-Trends*, volume 35(2), 2015.

[10] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, et al. Cocome-the common component modeling example. In *The Common Component Modeling Example*, pages 16–53. Springer, 2008.

[11] N. Juristo and O. Gómez. Replication of software engineering experiments. *Empirical software engineering and verification*, pages 60–88, 2012.

[12] T. Knapp. KAMP analysis applied to CoCoME. In *Seminar thesis, SDQ Chair, KIT*, 2012.

[13] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[14] Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

[15] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: Making the right architectural decision. In *17th International Conference on World Wide Web*, pages 805–814. ACM, 2008.

[16] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen. The future of empirical methods in software engineering research. In *Future of Software Engineering*, pages 358–378. IEEE, 2007.