1 CoCoME - The Common Component Modeling Example

Sebastian Herold[1], Holger Klus[1], Yannick Welsch[1], Andreas Rausch[1], Ralf Reussner[2], Klaus Krogmann[2], Heiko Koziolek[2], Raffaela Mirandola[3], Benjamin Hummel[4], Michael Meisinger[4], Christian Pfaller[4],

[1] TU Clausthal, Germany
[2] Universitt Karlsruhe, Germany
[3] Politecnico di Milano, Italy
[4] Technische Universität München, Germany

1.1 Introduction and System Overview

The example of use which was chosen as the Common Component Modeling Example (CoCoME) and on which the several methods presented in this book should be applied was designed according to the example described by Larman in [1]. The description of this example and its use cases in the current chapter shall be considered under the assumption that this information was delivered by a business company as it could be in the reality. Therefore the specified requirements are potentially incomplete or imprecise.

The mentioned example describes a *Trading System* as it can be observed in a supermarket handling sales. This includes the processes at a single *Cash Desk* like scanning products using a *Bar Code Scanner* or paying by credit card or cash as well as administrative tasks like ordering of running out products or generating reports. The following section gives a brief overview of such a Trading System and its hardware parts. Its required use cases and software architecture are described later in this chapter.

The Cash Desk is the place where the *Cashier* scans the goods the *Customer* wants to buy and where the paying (either by credit card or cash) is executed. Furthermore it is possible to switch into an express checkout mode which allows only Costumer with a few goods and also only cash payment to speed up the clearing. To manage the processes at a Cash Desk a lot of hardware devices are necessary (compare figure 1).

Using the *Cash Box* which is available at each Cash Desk a sale is started and finished. Also the cash payment is handled by the Cash Box. To manage payments by credit card a *Card Reader* is used. In order to identify all goods the Customer wants to buy the Cashier uses the *Bar Code Scanner*. At the end of the paying process a bill is produced using a *Printer*. Each Cash Desk is also equipped with a *Light Display* to let the Costumer know if this Cash Desk is in the express checkout mode or not. The central unit of each Cash Desk is the *Cash Desk PC* which wires all other components with each other. Also the software which is responsible for handling the sale process and amongst others for the communication with the *Bank* is running on that machine.

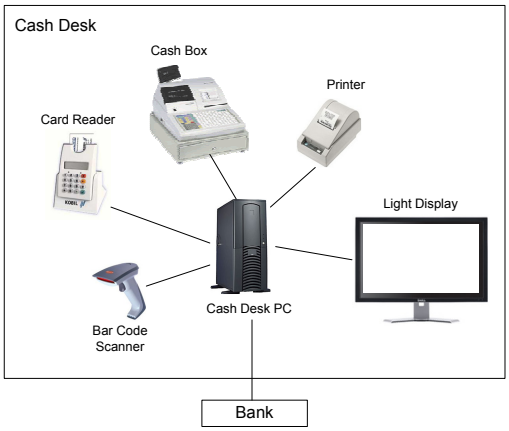

Fig. 1. The hardware components of a single Cash Desk.

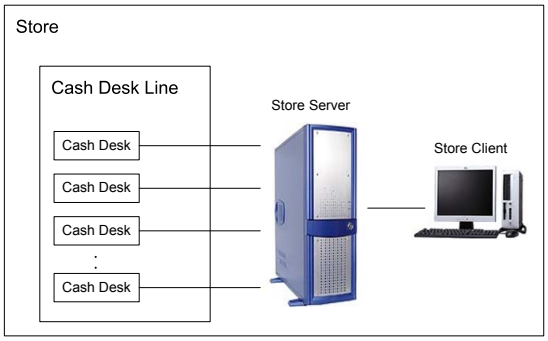

Fig. 2. An overview of entities in a store which are relevant for the Trading System.

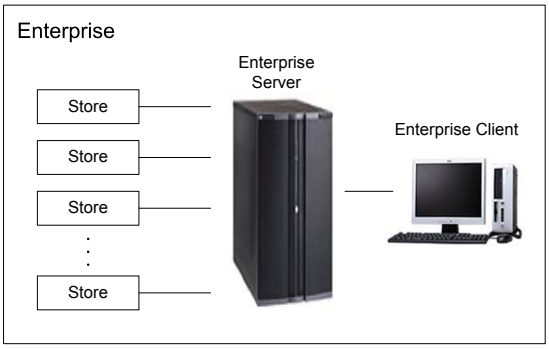

Fig. 3. The enterprise consists of several stores, an enterprise server and an enterprise client.

A *Store* itself contains of several Cash Desks whereas the set of Cash Desks is called *Cash Desk Line* here. The Cash Desk Line is connected to a *Store Server* which itself is also connected to a *Store Client* (compare figure 2). The Store Client can be used by the manager of the Store to view reports, order products or to chance the sales prices of goods. The Store Server also holds the *Inventory* of the corresponding Store.

A set of Stores is organized in an *Enterprise* where an *Enterprise Server* exists to which all Stores are connected (compare figure 3). With the assistance of an *Enterprise Client* the Enterprise Manager is able to generate several kinds of reports.

1.2 Functional Requirements and Use Case Analysis

In this section the considered use cases of the Trading System are introduced which are depicted in figure 4 with the involved actors. Each use case is described using a uniform template which includes a brief description of the use case itself, the standard process flow and its alternatives. Moreover, information like preconditions, postconditions and the trigger of the use cases are given. In the description of the use cases the codes in the squared brackets refer to extra-functional properties in section 1.3.

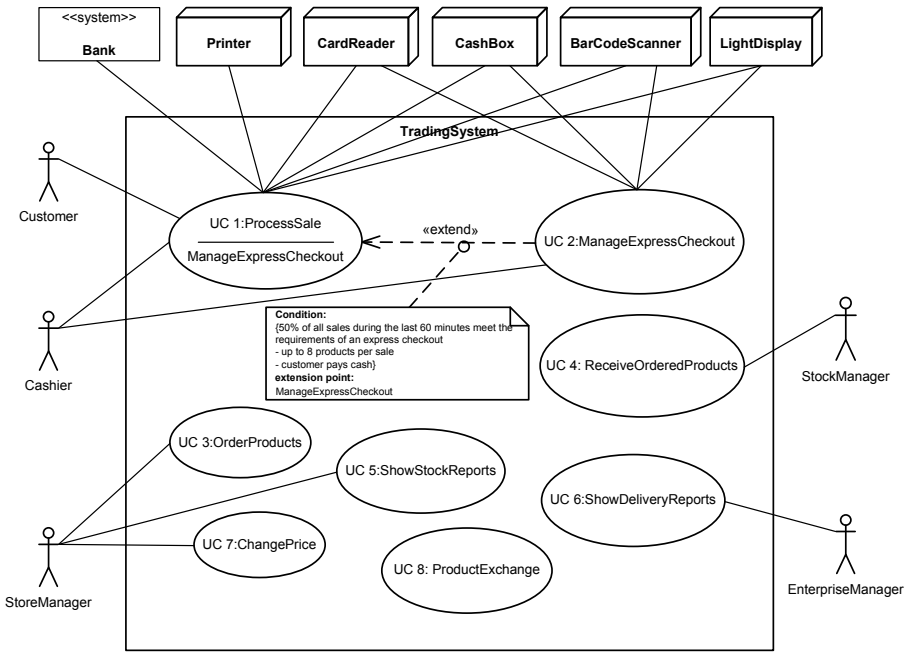

Fig. 4. An overview of all considered use cases of the Trading System.

UC 1 - Process Sale

Brief Description At the Cash Desk the products a Customer wants to buy are detected and the payment - either by credit card or cash - is performed.

Involved Actors Customer, Cashier, Bank, Printer, Card Reader, Cash Box, Bar Code Scanner, Light Display

Precondition The Cash Desk and the Cashier are ready to start a new sale.

Trigger Coming to the Cash Desk a Costumer wants to pay his chosen product items.

Postcondition The Customer has paid, has received the bill and the sale is registered in the Inventory.

Standard Process

1. The Customer arrives at the Cash Desk with goods to purchase. [arr1]
2. The Cashier starts a new sale by pressing the button *Start New Sale* at the Cash Box. [t12-1]
3. The Cashier enters the item identifier. This can be done manually by using the keyboard of the Cash Box [p13-1, t13-1] or by using the Bar Code Scanner [p13-2, t13-2].
4. Using the item identifier the System presents the corresponding product description, price, and running total. [t14-1]
   The steps 3-4 are repeated until all items are registered. [n11-2]
5. Denoting the end of entering items the Cashier presses the button *Sale Finished* at the Cash Box. [t15-1]
   (a) To initiate cash payment the Cashier presses the button *Cash Payment* at the Cash Box. [p15-1,t15a-1]
      i. The Customer hands over the money for payment. [t15a1-1]
      ii. The Cashier enters the received cash using the Cash Box and confirms this by pressing Enter. [t15a2-1]
      iii. The Cash Box opens. [t15a3-1]
      iv. The received money and the change amount are displayed [t15a4-1], and the Cashier hands over the change. [t15a4-2]
      v. The Cashier closes the Cash Box. [t15a5-1]
   (b) In order to initiate card payment the Cashier presses the button *Card Payment* at the Cash Box. [p15-2, t15b-1]
      i. The Cashier receives the credit card from the Customer [t15b1-1] and pulls it through the Card Reader. [t15b1-2]
      ii. The Customer enters his PIN using the keyboard of the card reader and waits for validation. [t15b2-1]
      The step 5.b.ii is repeated until a successful validation or the Cashier presses the button for cash payment. [t15b2-2, n15b2-1]

6. Completed sales are logged by the Trading System and sale information are sent to the Inventory in order to update the stock. [t16-1]
7. The Printer writes the receipt and the Cashier hands it out to the Costumer. [t17-1]
8. The Customer leaves the Cash Desk with receipt and goods.

Alternative or Exceptional Processes

- *In step 3: Invalid item identifier if the system cannot find it in the Inventory.* [p13-4]
  1. The System signals error and rejects this entry. [t13-3]
  2. The Cashier can respond to the error as follows:
     (a) It exists a human-readable item identifier: [p13-5]
        i. The Cashier manually enters the item identifier. [t13-4]
        ii. The System displays the description and price. [t14-1]
     (b) Otherwise the product item is rejected. [p13-6]
- *In step 5.b: Card validation fails.* [p15b2-2]
  1. The Cashier and the Customer try again and again.
  2. Otherwise the Cashier requires the Customer to pay cash.
- *In step 6: Inventory not available.* [p16-1]
  The System caches each sale and writes them into the Inventory as soon as it is available again. [t161-1]

UC 2 - Manage Express Checkout

Brief Description If some conditions are fulfilled a Cash Desk automatically switches into an express mode. The Cashier is able to switch back into normal mode by pressing a button at his Cash Desk. To indicate the mode the Light Display shows different colors.

Involved Actors Cashier, Cash Box, Light Display, Card Reader

Precondition The Cash Desk is either in normal mode and the latest sale was finished (case 1) or the Cash Desk is in express mode (case 2).

Trigger This use case is triggered by the system itself.

Postcondition The Cash Desk has been switched into express mode or normal mode. The Light Display has changed its color accordingly.

Standard Process

1. The considered Cash Desk is in normal mode [p2-1] and just finished a sale which matches the condition of an express checkout sale. Now 50% of all sales during the last 60 minutes fulfill the condition for an express checkout.
   (a) This Cash Desk, which has caused the achievement of the condition, is switched into express mode. [t21a-1]

(b) Furthermore the corresponding Light Display is switched from black into green to indicate the Cash Desk's express mode. [t21b-1]
(c) Paying by credit card is not possible anymore. [t21c-1]
(d) The maximum of items per sale is reduced to 8 and only paying by cash is allowed. [t21d-1]

2. The Cash Desk is in express mode [p2-2] and the Cashier decides to change back into normal mode.
   (a) The Cashier presses the button *Disable Express Mode*. [t22a-1]
   (b) The color of the Light Display is changed from green into black color. [t22b-1]
   (c) Cash and also card payment is allowed and the Costumer is allowed to buy as much goods as he likes. [t22c-1]

UC 3 - Order Products

Brief Description The Trading System provide the opportunity to order product items.

Involved Actors Store Manager

Precondition An Overview over the Inventory is available and the Store Client was started.

Trigger The Store Manager decided to buy new product items for his store.

Postcondition The order was placed and a generated order identifier was presented to the Store Manager.

Standard Process

1. A list with all products [n3-1] and a list with products running out of stock are shown. [n3-2, p3-1, t31-1]
2. The Store Manager chooses the product items to order and enters the corresponding amount. [t32-1]
3. The Store Manager presses the button *Order* at the Store Client's GUI. [t33-1]
4. The appropriate suppliers are chosen and orders for each supplier are placed. An order identifier is generated for each order and is shown to the Store Manager. [t34-1, t34-2, t34-3]

UC 4 - Receive Ordered Products

Brief Description Ordered products which arrive at the Store have to be checked for correctness and inventoried.

Involved Actors Stock Manager

Precondition The Store Client was started and the part Inventory of the Trading System is available.

Trigger The ordered products arrive at the Store.

Postcondition The Inventory is updated with the ordered products.

Standard Process

1. Ordered products arrive at the stock attached by an order identifier which has been assigned during the ordering process. [n4-1]
2. The Stock Manager checks the delivery for completeness and correctness. [p4-1, t42-1]
3. In the case of correctness, the Stock Manager enters the order identifier and presses the button *Roll in received order*. [t43-1]
4. The Trading System updates the Inventory. [t44-1]

Alternative or Exceptional Processes

- *In step 2: Delivery not complete or not correct.* [p4-2]
  The products are sent back to the supplier and the Stock Manager has to wait until a correct and complete delivery has arrived. This action does not recognized by the System.

UC 5 - Show Stock Reports

Brief Description The opportunity to generate stock-related reports is provided by the Trading System.

Involved Actors Store Manager

Precondition The reporting GUI at the Store Client has been started.

Trigger The Store Manager wants to see statistics about his store.

Postcondition The report for the Store has been generated and is displayed on the reporting GUI.

Standard Process

1. The Store Manager enters the store identifier and presses the button *Create Report*. [t51-1]
2. A report including all available stock items in the store is displayed. [t52-1]

UC 6 - Show Delivery Reports

Brief Description The Trading System provides the opportunity to calculate the mean times a delivery from each supplier to an considered enterprise takes.

Involved Actors Enterprise Manager

Precondition The reporting GUI at the Store Client has been started.

Trigger The Enterprise Manager wants to see statistics about the enterprise.

Postcondition The report for the Enterprise has been generated and is displayed to the Enterprise Manager.

Standard Process

1. The Enterprise Manager enters the enterprise identifier and presses the button *Create Report*. [t61-1]
2. A report which informs about the mean times is generated. [t62-1]

UC 7 - Change Price

Brief Description The System provides the opportunity to change the sales price for a product.

Involved Actors Store Manager

Precondition The store GUI at the Store Client has been started.

Trigger The Store Manager wants to change the sales price of a product for his store.

Postcondition The price for the considered product has been changed and it will be sold with the new price now.

Standard Process

1. The System presents an overview over all available products in the store. [t71-1]
2. The Store Manager selects a product item [t72-1] and changes its sales price. [t72-2]
3. The Store Manager commits the change by pressing ENTER. [t73-1]

UC 8 - Product Exchange (on low stock) Among Stores

Brief Description If a store runs out of a certain product (or a set of products; "required good"), it is possible to start a query to check whether those products are available at other Stores of the Enterprise ("providing Stores"). Therefore the Enterprise Server and the Store Servers need to synchronize their data on demand (one scheduled update per day or per hour is not sufficient). After a successful query the critical product can be shipped from one to other Stores. But it has to be decided (using heuristics to compute the future selling frequency),

whether the transportation is meaningful. For example, if the product is propably sold out at all Stores within the same day, a transportation does not make sense.

Expressed in a more technical way one Store Server is able to start a query at the Enterprise Server. The Enterprise Server in turn starts a query for products available at other Stores. As the Enterprise Server does not have the current global data for Stores at any time (due to a write caching latency at the Store Servers) the Enterprise Server has to trigger all Store Servers to push their local data to the Enterprise Server.

Involved Actors This use case is not an end-user use case. Only servers are involved.

Precondition The Store Server with the shortage product is able to connect to the Enterprise Server.

Trigger This use case is triggered by the system itself.

Postcondition The products to deliver are marked as incoming or unavailable, respectively, in the according Stores.

Standard Process

1. A certain product of the Store runs out.
2. The Store Server recognizes low stock of the product. [t82-1]
3. The Store Server sends a request to the Enterprise Server (including an identification of the shortage products, and a Store id) [t83-1]
4. The Enterprise Server triggers all Stores that are "near by" (e. g. ¡300 km) the requiring store, to flush their local write caches. So the Enterprise Server database gets updated by the Store Server. [t84-1, t84-1]
5. The Enterprise Server does a database look-up for the required products to get a list of products (including amounts) that are available at providing Stores. [t85-1]
6. The Enterprise Server applies the "optimization criterion" (specified above) to decide, whether it is meaningful to transport the shortage product from one store to another (heuristics might be applied to minimize the total costs of transportation). This results in a list of products (including amounts) per providing store that have to be delivered to the requiring Store. [t86-1]
7. The Store Server, initially sending the recognition of the shortage product, is provided with the decision of the Enterprise Server. [t87-1]
   (a) The required product is marked as incoming. [t87-2]
8. The Store Server of a near by Store is provided with information that it has to deliver the product. [t88-1]
   (a) The required product is marked as unavailable in the Store. [t88-2]

Alternative or Exceptional Processes

- The Enterprise Server is not available: The request is queued until the Enterprise Server is available and then is send again.
- One or more Store Servers are not available: The Enterprise Server queues the requests for the Store Servers until they are available and then resend them. If a Store Server is not available for more than 15 minutes the request for this Server is canceled. It is assumed, that finally unavailable Store Servers do not have the required product.

Extension on use case 8 - Remove Incoming Status

Brief Description If the first part of use case 8 (as described above) has passed, for moved products an amount marked as incoming remains at the Inventory of the Store receiving the products. An extension allows to change that incoming mark via a user interface at the Store Client if the moved products arrive at a Store.

Precondition The Inventory is available and the Store Client has been started.

Trigger The moved products (according to UC8) arrive at the Store.

Postcondition For the amount of incoming products the status "incoming" is removed in the Inventory.

Standard Process

1. The products arrive at the stock of the Store.
2. For all arriving products the Stock Manager counts the incoming amount.
3. For every arriving product the Stock Manager enters the identifier and its amount into the Store Client.
4. The system updates the Inventory.

Alternative or Exceptional Processes

- If the entered amount of an incoming product is larger than the amount accounted in the Inventory, the input is rejected. The incoming amount has to be re-entered.

1.3 Extra-Functional Properties

The following table includes CoCoME's extra-functional properties in terms of timing, reliability, and usage profile related information. They can be seen as guiding values when conducting QoS-analysis with CoCoME. The results from different methods can be compared more adequately if they are based on the same extra-functional properties.

The extra-functional properties map to certain steps in the use cases and have labels to illustrate the relationship (e.g., t12-3 stands for "use case 1 step 2, time no. 3"). As a notation for the values, we have used the tagged value language from the OMG UML Profile for Schedulability, Performance, and Time [2]. We have used the tags `PAperfValue` [2, p.7-21] for timing values and `RTarrivalPattern` [2, p.4-35] for customer arrival rates, as they allow a fine-grained specification of probability distributions. Note for histogram specifications: "The histogram distribution has an ordered collection of one or more pairs that identify the start of an interval and the probability that applies within that interval (starting from the leftmost interval) and one end-interval value for the upper boundary of the last interval"[2, p.4-34].

The values in the table are either *assumed* ('assm') by us or *required* ('req') from the system as part of the specification. We estimated most of the values based on statistics for typical German super markets, and our own experience. The values should be understood as guiding values and should not restrict CoCoME modelers from using their own extra-functional properties for CoCoME.

CoCoME Overall
n0-1: Number of stores 200
n0-2: Cash desks per store 8
UC1 - Process Sale
arr1: Customer arrival rate per store PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 320.0, 'hr'))
n11-1: Number of open cash desks per store ('assm', 'dist', ('histogram', 0, 0.1, 2, 0.2, 4, 0.4, 6, 0.3, 8, '#Open Cash Desks'))
n11-2: Number of goods per customer ('assm', 'dist', ('histogram', 1, 0.3, 8, 0.1, 15, 0.15, 25, 0.15, 50, 0.2, 75, 0.1, 100, '#Goods per Customer'))
t12-1: Time for pressing button "Start New Sale" PAdemand = ('assm', 'mean', (1.0, 's'))
t13-1: Time for scanning an item PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.9, 0.3, 0.05, 1.0, 0.04, 2.0, 0.01, 5.0, 's'))
t13-2: Time for manual entry PAdemand = ('assm', 'mean', (5.0, 's'))
t13-3: Time for signaling error and rejecting an ID PAdemand = ('req', 'mean', (10, 'ms'))
t13-4: Time for manually entering the item identifier after error PAdemand = ('assm', 'mean', (5.0, 's'))
p13-1: Probability of using the bar code scanner per item 0.99
p13-2: Probability of manual entry per item

Continued on next page

0.01
p13-3: Probability of valid item ID 0.999
p13-4: Probability of invalid item ID 0.001
p13-5: Probability of human-readable item ID 0.9
p13-6: Probability of rejecting an item 0.1
t14-1: Time for showing the product description, price, and running total PAdemand = ('req', 'mean', (10, 'ms'))
t15-1: Time for pressing button "Sale Finished" PAdemand = ('assm', 'mean', (1.0, 's'))
t15a-1: Time for pressing button "Cash Payment" PAdemand = ('assm', 'mean', (1.0, 's'))
t15a1-1: Time for handing over the money PAdemand = ('assm', 'dist', ('histogram', 2.0, 0.3, 5.0, 0.5, 8.0, 0.2, 10.0, 's'))
t15a2-1: Time for entering the cash received and confirming PAdemand = ('assm', 'mean', (2.0, 's'))
p15-1: Probability of cash payment 0.5
p15-2: Probability of credit card payment 0.5
n15b2-1: Number of times a customer has to enter the PIN ('assm', 'dist', ('histogram', 1, 0.9, 2, 0.09, 3, 0.01, 4, 'times entering PIN'))
p15b2-1: Probability of valid CC id 0.99
p15b2-2: Probability of invalid CC id 0.01
t15a3-1: Time for opening the cash box PAdemand = ('assm', 'mean', (1.0, 's'))
t15a4-1: Time until displaying received money and change amount PAdemand = ('req', 'mean', (10, 'ms'))
t15a4-2: Time for handing over the change PAdemand = ('assm', 'dist', ('histogram', 2.0, 0.2, 3.0, 0.6, 4.0, 0.2, 5.0, 's'))
t15a5-1: Time for closing the cash box PAdemand = ('assm', 'mean', (1.0, 's'))
t15b-1: Time for pressing button "Card Payment" PAdemand = ('assm', 'mean', (1.0, 's'))
t15b1-1: Time for receiving the credit card PAdemand = ('assm', 'dist', ('histogram', 3.0, 0.6, 4.0, 0.4, 5.0, 's'))
t15b1-2: Time for pulling the credit card through the reader PAdemand = ('assm', 'mean', (2.0, 's'))

Continued on next page

t15b2-1: Time for entering the PIN PAdemand = ('assm', 'dist', ('uniform', 1.0, 5.0, 's'))
t15b2-2: Time waiting for validation PAdemand = ('assm', 'dist', ('histogram', 4.0, 0.9, 5.0, 0.1, 20.0, 's'))
t16-1: Time for sending sale information and updating stock PAdemand = ('req', 'mean', (100, 'ms'))
t161-1: Time for writing cached sales logs after inventory is back up PAdemand = ('req', 'mean', (2, 's'))
p16-1: Probability of Failure on Demand of Inventory System 0.001
t17-1: Time for printing the receipt and handing it out PAdemand = ('assm', 'mean', (3.0, 's'))
UC2 - Manage Express Checkout
arr-2: Manage Express Checkout arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 1, 'hr'))
p2-1: Probability of being in normal mode 0.8
p2-2: Probability of being in express mode 0.2
t21a-1: Time for switching to express mode PAdemand = ('req', 'mean', (10, 'ms'))
t21b-1: Time for switching light display PAdemand = ('req', 'mean', (10, 'ms'))
t21c-1: Time for deactivating credit card payment PAdemand = ('req', 'mean', (10, 'ms'))
t21d-1: Time for setting the maximum number of items PAdemand = ('req', 'mean', (10, 'ms'))
t22a-1: Time for pressing button "Disable Express Mode" PAdemand = ('assm', 'mean', (1.0, 's'))
t22b-1: Time for switching light display PAdemand = ('req', 'mean', (10, 'ms'))
t22c-1: Time for reactivating credit card payment PAdemand = ('req', 'mean', (10, 'ms'))
UC3 - OrderProducts
arr-3: Order arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 1, 'days'))
n3-1: Number of all products 5000
n3-2: Number of products running out of stock ('assm', 'dist', ('histogram', 100, 0.25, 200, 0.25, 300, 0.25, 400, 0.25, 500 '#Goods out of stock'))

Continued on next page

p3-1: Percentage of out of stock products being reordered 0.98
t31-1: Time until showing the lists of all products and missing products PAdemand = ('req', 'mean', (10, 'ms'))
t32-1: Time for choosing the products to order and entering the amount PAdemand = ('assm', 'mean', (10, 's'))
t33-1: Time for pressing button "Order" PAdemand = ('assm', 'mean', (1, 's'))
t34-1: Time for querying the inventory data store PAdemand = ('req', 'mean', (20, 'ms'))
t34-2: Time for creating a new order entry PAdemand = ('req', 'mean', (10, 'ms'))
t34-3: Time for creating a new product order PAdemand = ('req', 'mean', (10, 'ms'))
UC4 - Receive Ordered Products
arr-4: Order arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 1, 'days'))
n-4: Number of products arriving ('assm', 'dist', ('histogram', 100, 0.25, 200, 0.25, 300, 0.25, 400, 0.25, 500 '#Goods arriving'))
p4-1: Probability of complete and correct order 0.99
p4-2: Probability of incomplete or incorrect order 0.01
t42-1: Time for checking completeness of order PAdemand = ('assm', 'mean', (30, 'min'))
t43-1: Time for pressing button "Roll in received order" PAdemand = ('assm', 'mean', (1, 's'))
t44-1: Time for updating the inventory PAdemand = ('assm', 'mean', (100, 'ms'))
UC5 - Show Stock Reports
arr-5: Show Stock Reports arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 3, 'hr'))
t51-1: Time for entering store id and pressing button "Create Report" PAdemand = ('assm', 'mean', (1, 's'))
t52-1: Time for generating the report PAdemand = ('req', 'mean', (0.5, 's'))
UC6 - Show Delivery Reports
arr-6: Show Delivery Reports arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 1, 'days'))

Continued on next page

t61-1: Time for entering store id and pressing button "Create Report" PAdemand = ('assm', 'mean', (1, 's'))
t62-1: Time for generating the report PAdemand = ('req', 'mean', (0.5, 's'))
UC7 - Change Price
arr-7: Change Price arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 3, 'hr'))
t71-1: Time for generating the overview PAdemand = ('req', 'mean', (10, 'ms'))
t72-1: Time for selecting a product item PAdemand = ('assm', 'mean', (5, 's'))
t72-2: Time for changing the sales price PAdemand = ('assm', 'mean', (5, 's'))
t73-1: Time for pressing button "Enter" PAdemand = ('assm', 'mean', (1, 's'))
UC8 - Product Exchange
arr-8: Show Stock Reports arrival rate PAopenLoad.PAoccurrence = ('unbounded', ('exponential', 1, 'days'))
n8-1: Number of stores nearby for a store server ('assm', 'dist', ('histogram', 10, 0.7, 20, 0.3, 30 '#Shops nearby'))
p8-1: Probability of failure on demand (enterprise server) 0.0001
p8-2: Probability of failure on demand (store server) 0.001
t82-1: Time for store server to detect low stock PAdemand = ('req', 'mean', (10, 'ms'))
t83-1: Time for store server to query enterprise server PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.5, 0.5, 0.5, 1.0, 's'))
t84-1: Time for enterprise server to query one store server PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.5, 0.5, 0.5, 1.0, 's'))
t84-2: Time for flushing the cache of one store server and returning the result PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.5, 0.5, 0.5, 1.0, 's'))
t85-1: Time for database lookup at enterprise server PAdemand = ('req', 'mean', (10, 'ms'))
t86-1: Time for determining which store to deliver from PAdemand = ('req', 'mean', (1, 's'))
t87-1: Time for returning the result to the store server PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.5, 0.5, 0.5, 1.0, 's'))
t87-2: Time for marking goods as incoming at store server PAdemand = ('req', 'mean', (10, 'ms'))
t88-1: Time for sending delivery request to store server

Continued on next page

PAdemand = ('assm', 'dist', ('histogram', 0.0, 0.5, 0.5, 0.5, 1.0, 's'))
t88-2: Time for marking good as unavailable PAdemand = ('req', 'mean', (10, 'ms'))

Table 1: CoCoME - Extra-functional Properties

1.4 Architectural Component Model

In this section, the architecture of the *Trading System* is described in more detail using UML 2.0 ([2]) with additional own notations like multiplicities at ports. After an overview of the structure of the system which introduces single parts, like interfaces and connections between them, an overview of the behavior is given. To show the structure the single components beginning with the topmost, namely the component TradingSystem, and going on with the inner, more detailed components are beheld. For every use case the behavior of the system is visualized by sequence diagrams. Additional, a prototype of the system was implemented. As far as the real subject is meant, the name of it is written separately. The names of software components are written in one word.

Structural View on the Trading System

The structure of the Trading System is designed to integrate an embedded system based on a bus-architecture and an information system based on a layered architecture. Figure 5 shows the super-component TradingSystem and the two components *Inventory* and *CashDeskLine* TradingSystem consists of.

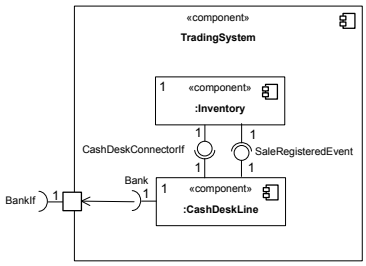

Fig. 5. TradingSystem and its two components Inventory and CashDeskLine.

The information system is represented by the component Inventory, while the component CashDeskLine represents the embedded system. For each instance of TradingSystem exists respectively one instance of Inventory and CashDeskLine which is indicated by the number in the upper left of the components. Also visible is the fact that the communication between the components CashDeskLine and

Inventory is handled by the interfaces *CashDeskConnectorIf* and *SaleRegisteredEvent*. The interface CashDeskConnectorIf defines a method for getting product information like description and price using the product bar code. Events like starting a new sale are registered at an asynchronous event channel. To handle these events the event SaleRegisteredEvent is used. Furthermore, CashDeskLine is connected to the bank via an interface *BankIf* in order to handle the card payment.

Structural View on the component Inventory As already mentioned, the component Inventory represents the information system and is organized as a layered architecture. As shown in figure 6, these layers are *GUI*, *Application* and *Data* which are completed by a component *Database*.

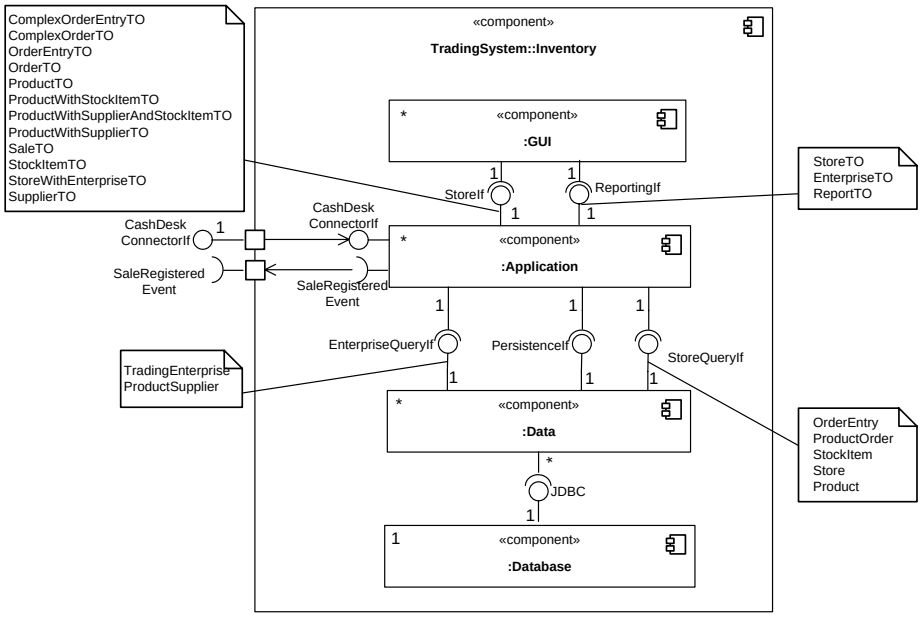

Fig. 6. The inner structure of the component Inventory. The notes show the data types which are relevant for the corresponding interface.

For each instance of Inventory exists only one instance of the component Database where all data is stored. Because of the case having only one instance of Inventory in TradingSystem there in all exists only one instance of Database per instance of TradingSystem. The component Data representing the data layer of a classical three-layer-architecture hides details of the database and provides data access to the application layer represented by the component Application. The communication between the components Database and Data is managed by

JDBC ([3]) in connection with Hibernate ([4]), an implementation of the Java Persistence API ([5]).

The component Data provides the three interfaces *EnterpriseQueryIf*, *StoreQueryIf* and *PersistenceIf*. To get a persistence context the interface PersistenceIf offers an appropriate method. The interface EnterpriseQueryIf contains queries like the mean time to delivery by taking all stores of an enterprise into account. StoreQueryIf defines methods required at a Store like changing the sales price of a product or managing the Inventory.

The component Application contains the application logic. It uses the interfaces defined by the component Data in order to send queries or changes to the database and provides the interfaces *StoreIf* and *ReportingIf* to deliver results of database queries to the component GUI. Between the components Application and Data object-oriented interfaces and between Application and GUI services-oriented interfaces are used (EnterpriseQueryIf and StoreQueryIf respectively StoreIf and ReportingIf). As it is determined as a property of a service-oriented interfaces via the interfaces between Application and GUI no references are passed. Instead, so called *Transfer Objects (TO)* are defined which are used for the data transfer. The component Application itself has references on data objects located in the component Data in order to receive required data.

Data Layer Figure 7 shows an overview of the component Data with its three subcomponents *Enterprise*, *Store* and *Persistence*. These components implement the similar named interfaces EnterpriseQueryIf, PersistenceIf and StoreQueryIf. In figure 7 the various data types the interfaces deal with are shown as notes whereas figure 8 gives a more detailed overview of the data model with attributes and possible navigation paths.

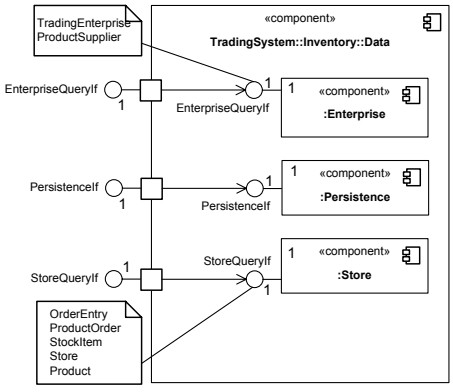

Fig. 7. The inner structure of the data layer of the component Inventory.

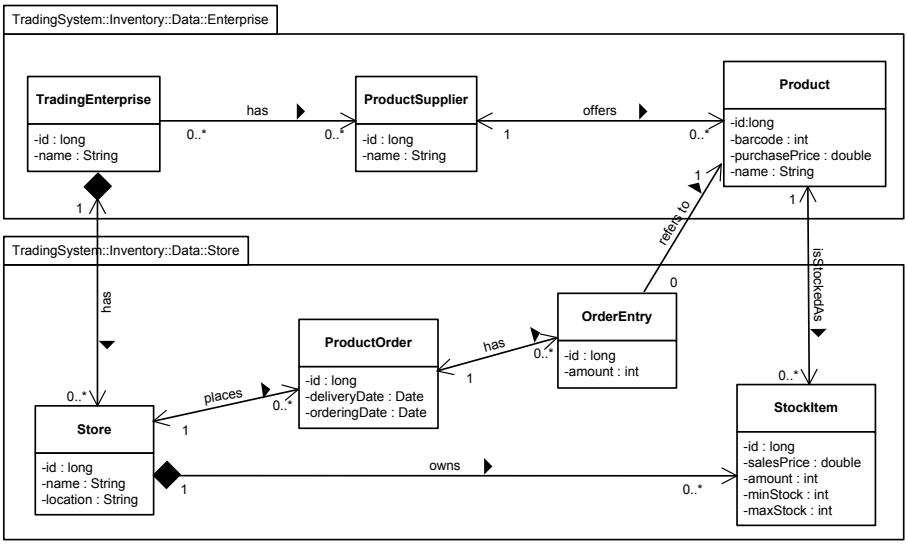

Fig. 8. The data model of the TradingSystem.

Application Layer The component Application representing the application layer consists of the three components *Reporting*, *Store* and *ProductDispatcher* as shown in figure 9. The component Reporting implements the interface ReportingIf whereas the component Store implements the interfaces CashDeskConnectorIf and StoreIf. It also requires the interfaces SaleRegisteredEvent and ProductDispatcherIf. The latter defines a method for the Enterprise Server to search for a product at another Store.

While the communication between Data and Application is realized by passing references of persistent objects to the Application, the Application uses PoJOs (Plain old Java Objects) or Transfer Objects (TO) to pass information to the GUI and to the CashDeskLine. An overview of all Transfer Objects and their relation between each other is shown in figure 10.

GUI Layer As shown in figure 11 the component GUI has the two subcomponents *Reporting* and *store*. The component Reporting implements the visualization of various kinds of reports using the interface ReportingIf to get the data. Whereas the component Store offers the user interface for the Store Manager in order to do managing tasks like ordering products or changing the sale prices.

Structural View on the component CashDeskLine The component *CashDeskLine* represents the embedded part. It is responsible for managing all Cash Desks, their hardware, and the interaction between Cash Desks and between the devices connected with each Cash Desk. The main communication is done using events which are sent through event channels.

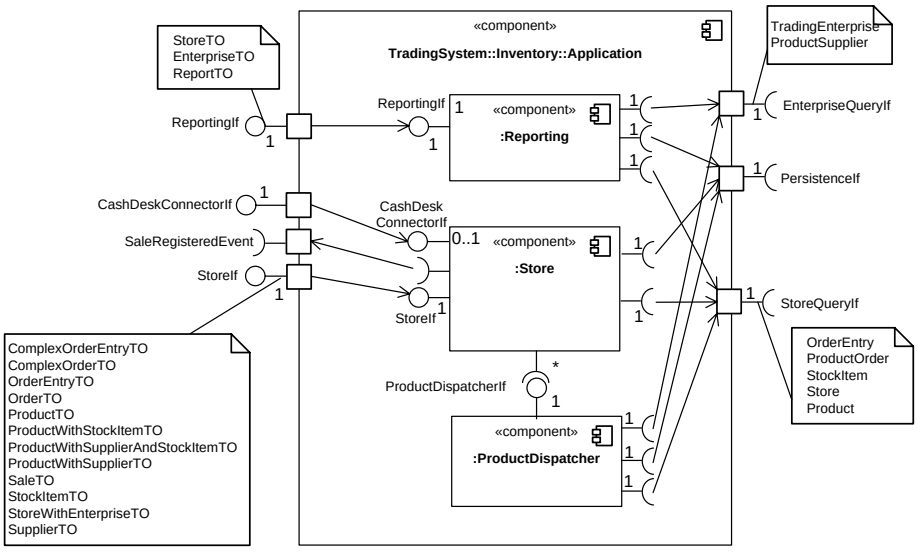

Fig. 9. The inner structure of the application layer of the component Inventory.

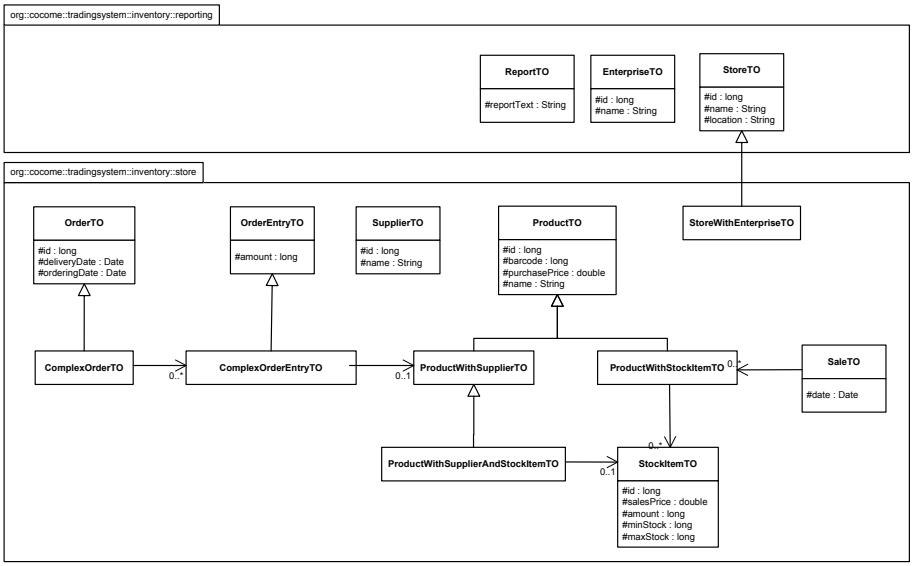

Fig. 10. The transfer objects used for data exchange between the application layer and the GUI layer.

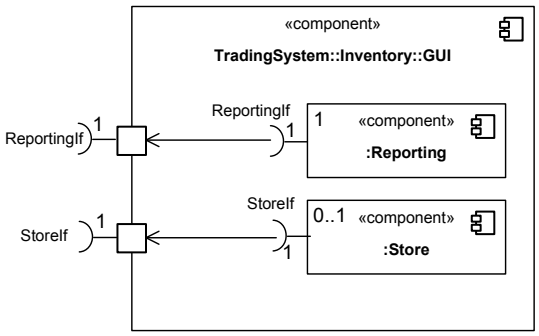

Fig. 11. The inner structure of the GUI layer of the component Inventory.

Figure 12 gives an overview of the structure of the component CashDeskLine. It is shown that CashDeskLine consists of several instances of CashDesk and a component *EventBus* which manages two instances of *EventChannel*, namely *cashDeskChannel* and *extCommChannel* which are shared by all instances of CashDesk. The channel cashDeskChannel is used by the CashDesk to enable communication between all device controllers which are connected to a CashDesk, like *CashDeskApplication*, *LightDisplayController* or *CashDeskGUI*. Each controller itself is connected to the according hardware device and so builds the bridge between the hardware and the middleware. The channel extCommChannel is used by the component CashDeskApplication to write the information about completed sales into the Inventory. Additionally, this channel is used for the communication between the components *Coordinator* and CashDesk. The Coordinator itself is responsible for dealing with managing express checkouts, whereas its task is to decide if a Cash Desk has to be switched into express mode (see use case 2).

As shown in figure 12, the component CashDeskApplication requires the interface *CashDeskConnectorIf*. This interface is provided by the component Inventory and is used to get the product description and sales price by transferring the bar code of a product. These information are required during the scanning of product items the customer wants to buy (see use case 1).

Figure 13 shows again the components CashDesk consists of and, in addition, the events each component sends and for which types of events each component is registered at the channel. The semicircles indicate events the component can handle while the circles indicate events which are sent by the component. For example, the controller *CardReaderController* handles the event *ExpressModeEnabledEvent* while sending the events *CreditCardScannedEvent* and *PINEnteredEvent*.

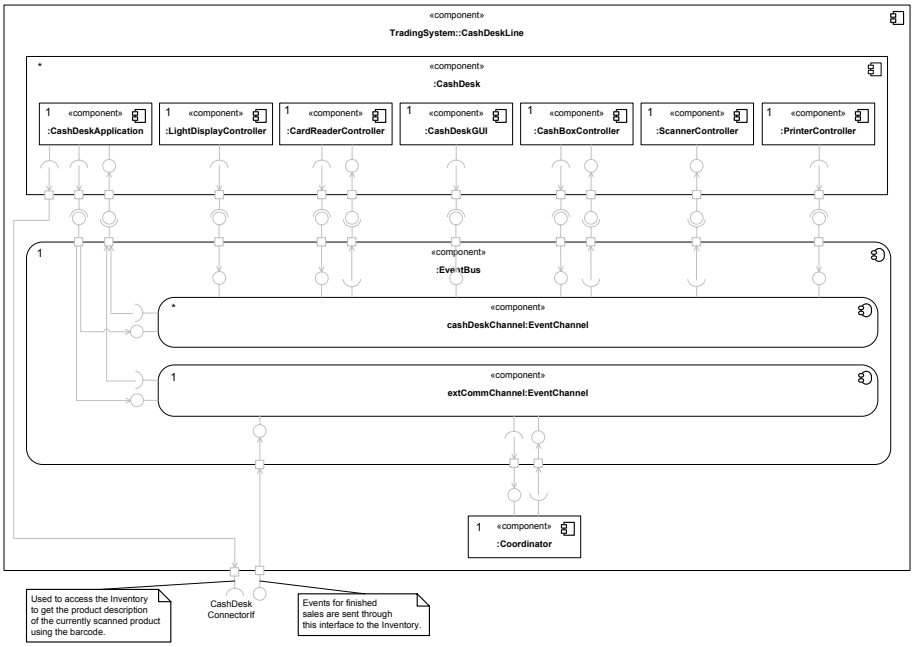

Fig. 12. The inner structure of the component CashDeskLine.

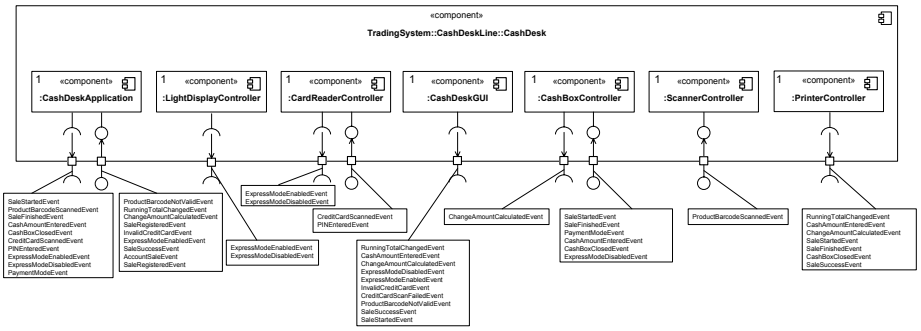

Fig. 13. A more detailed view on the component CashDesk and its published and subscribed events.

Deployment View on the Trading System

The deployment view of the Trading System zooms in on which devices are considered and where the different components are instantiated. Each Cash Desk is linked to an own Cash Desk PC. This Cash Desk PC is connected to several devices like Bar Code Scanner or Card Reader. The controllers of these devices run on the component *CashDeskPC* as well as all other subcomponents of the component CashDesk shown in figure 14. Furthermore, on each CashDeskPC an event channel is established which is used for the communication between the peripheral devices.

In each Store exists one Store Server to which all Cash Desk PCs in this Store are connected. At the component StoreServer the four components Coordinator, extCommChannel, Application and Data and their subcomponents are located. The first two components were presented in the section before and are responsible for managing the express checkout respectively for the communication. The component Data representing the data layer is connected via JDBC to the component Database which is placed at the component EnterpriseServer. Representing the application layer the component Application is communicating with the component Inventory::GUI deployed at the component *StoreClient*. In addition to the component Database, the component Data and the component Reporting are deployed at EnterpriseServer. The component Data is also connected to the component Database.

Behavioral View on the Trading System

This section provides a more detailed view of the realization of each use case introduced before by using UML 2.0 sequence diagrams which show the interaction between actors and components. First an important notation aspect is pointed out because of changes in the notation for sequence diagrams in UML 2.0. This considers the notation for synchronous and asynchronous method calls where synchronous method calls are depicted using filled arrowheads, compared to asynchronous method calls which are depicted using unfilled arrowheads.

Behavioral View on UC 1 - Process Sale The sequence diagrams in figures 15, 16 and 17 show the sale process including the communication between the various involved components. The sale process starts when the Cashier presses the button *Start Sale* at his Cash Box. Then the corresponding software component CashBox calls a method at the component CashBoxController which publishes the *SaleStartedEvent* using the *cashDeskChannel* (compare figure 12). The three components CashDeskApplication, PrinterController and CashDeskGUI react to events of the kind SaleStartedEvent. In order to receive these they have to register themselves at the channel cashDeskChannel for these events and implement the according event handlers. These event handlers are called by the messaging middleware which JMS ([6]) is in the implementation of the prototype. At the Cash Desk the Printer starts printing the header of the receipt initiated by the component PrinterController and initiated by the component

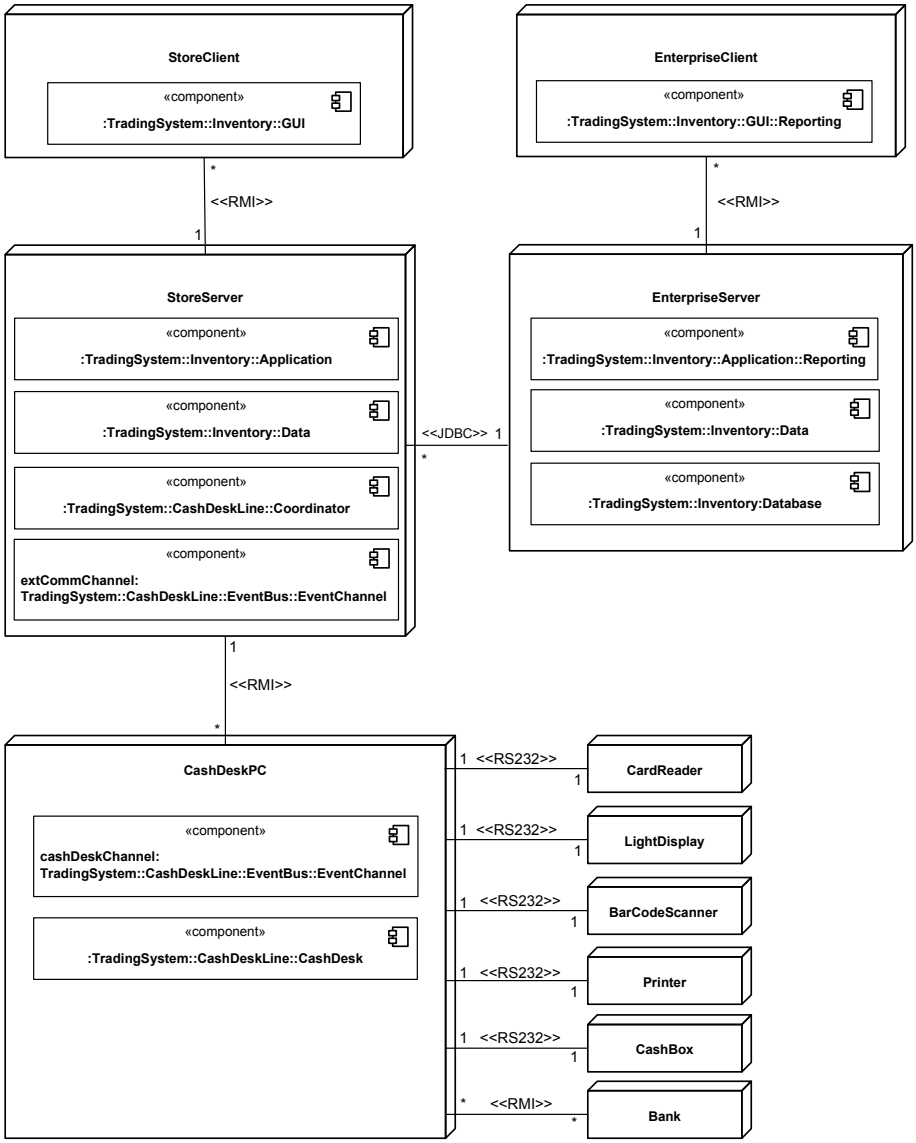

Fig. 14. The deployment of the Trading System.

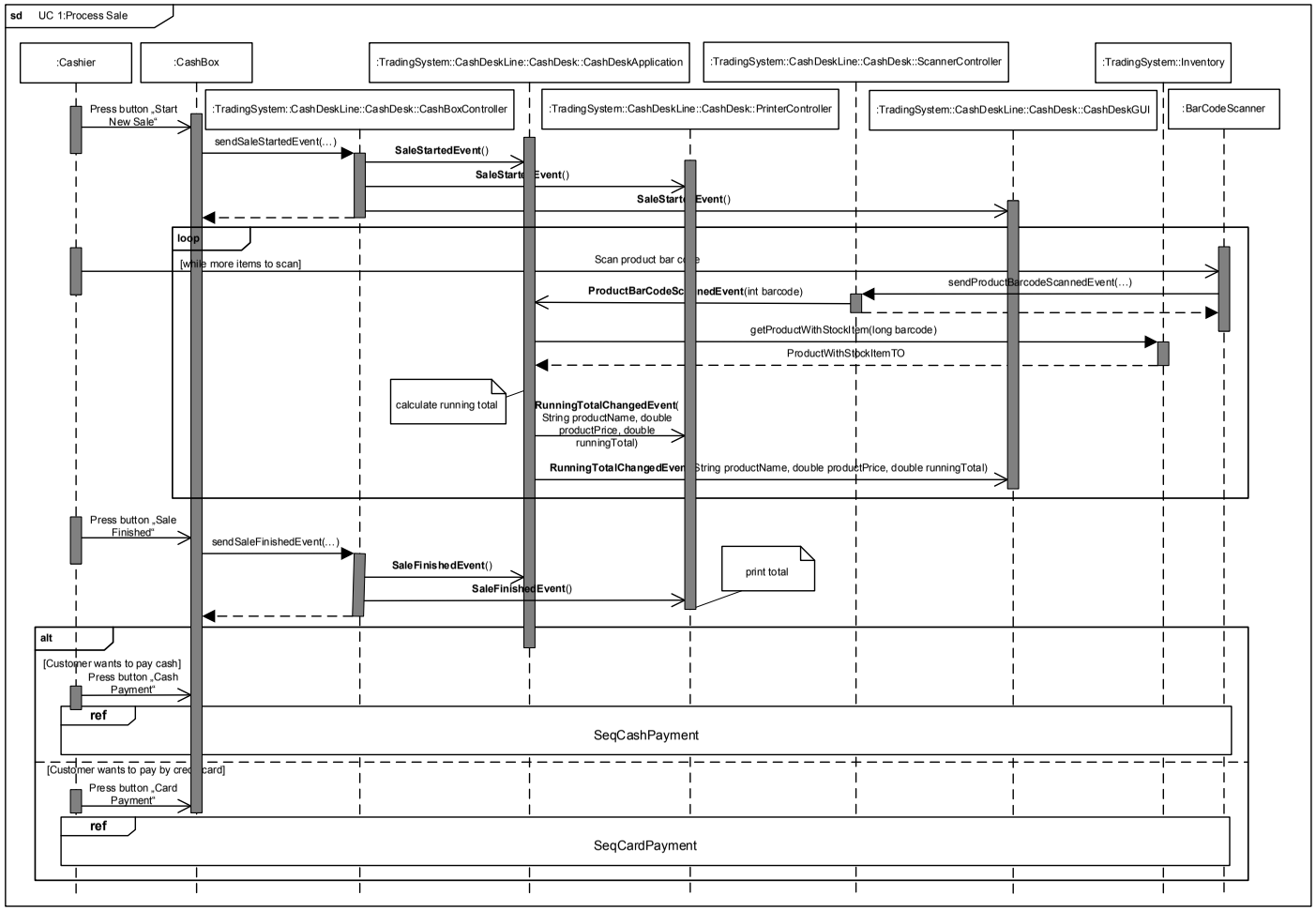

Fig. 15. Sequence diagram of the main sale process (UC 1).

CashDeskGUI a text at the Cash Desk indicates the start of a new sale. Some components connected with the channel cashDeskChannel implement a finite state machine, like CashDeskApplication or PrinterController in order to react appropriately on further incoming events.

In the next phase of the selling process the desired products are identified using the Bar Code Scanner which submitts the data to the corresponding controller *ScannerController* which in turn publishes the event *ProductBarCodeScannedEvent*. The component CashDeskApplication gets the product description from the Inventory and calculates the running total and announces it on the channel.

After finishing the scanning process, the Cashier presses the button *Sale Finished* at the Cash Box. Now the Cashier can choose the payment method based on the decision of the costumer by pressing the button *Cash Payment* or *Card Payment* at his Cash Desk. Figure 16 and 17 illustrate the sequences for each payment method which shall not described in detail here.

Behavioral View on UC 2 - Manage Express Checkout The basic idea behind the process of managing express checkouts is to hold a statistic about sales using the component *Coordinator*. If the condition for an express checkout is fulfilled, the Coordinator releases the event *ExpressModeEnabledEvent* and due to this a Cash Desk will change into express mode. The Cashier is allowed to decide to switch back into normal mode by simply pressing the button *Disable Express Mode* at his Cash Desk. This causes the event *ExpressModeDisabledEvent* which forces the devices to switch back into normal mode. The sequence diagram in figure 18 shows the described process in more detail.

Behavioral View on UC 3 - Order Products This use case deals with ordering products from a supplier if they are running out of stock at a store. To initiate an order, the Store Manager can select the products which have to be ordered in the desired amount and then presses the button *Order* at the StoreGUI. As result of this use case the Store Manager gets a set of order identifiers. Not only one identifier is returned, because one order can be split and placed at different suppliers. These identifiers are used by the Stock Manager in use case 4 while receiving and accounting the products. Figure 19 shows the sequence diagram of this process with more details.

Behavioral View on UC 4 - Receive Ordered Products If the Stock Manager has ordered products, they will arrive at the Store. To roll in the product items the StockManager enters the order identifier assigned during ordering and presses the button *Roll in received order*. This results in system actions like setting the order's delivery date and rising the amount of the stored products as it can be tracked in the sequence diagram in figure 20.

Behavioral View on UC 5 - Show Stock Reports If the Store Manager wants to see if products are running out of stock in his store, he can get the

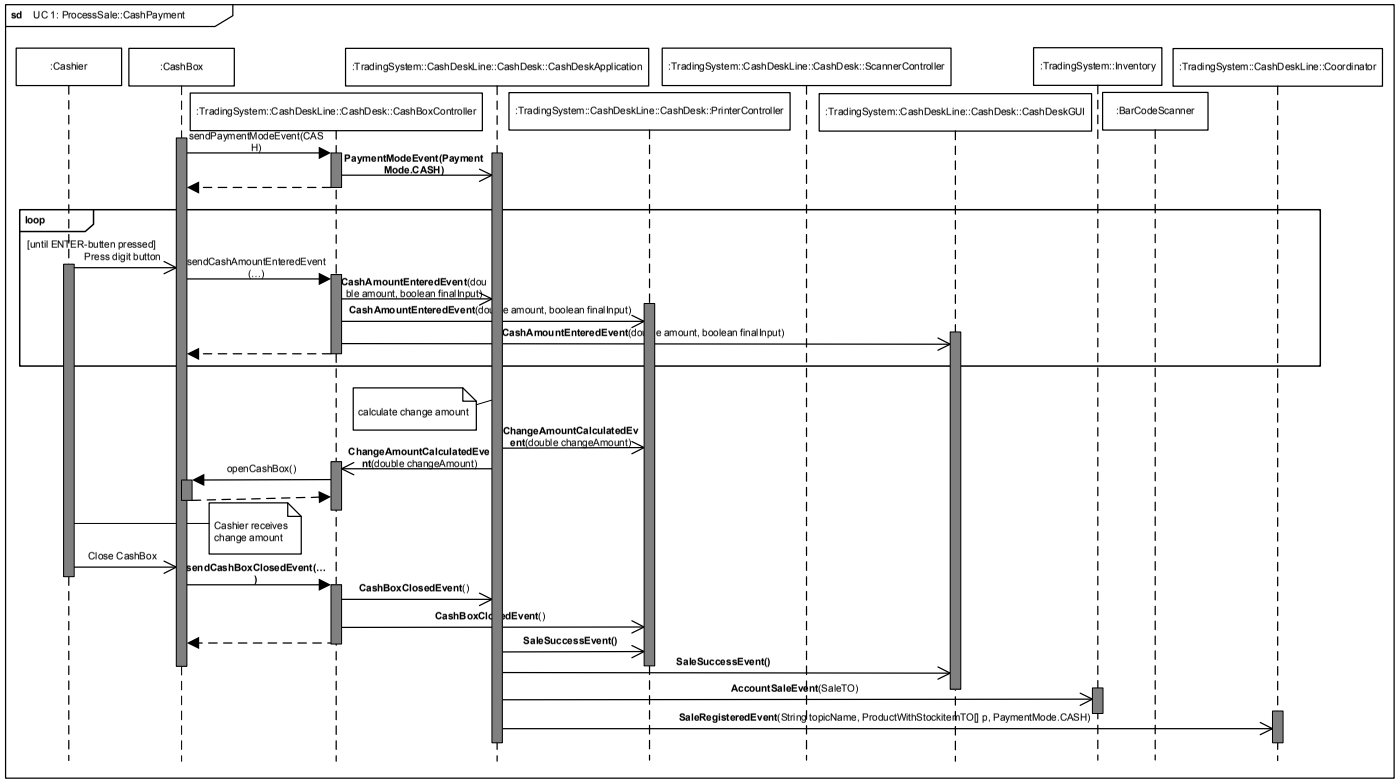

Fig. 16. Sequence diagram of the process of cash payment (UC 1).

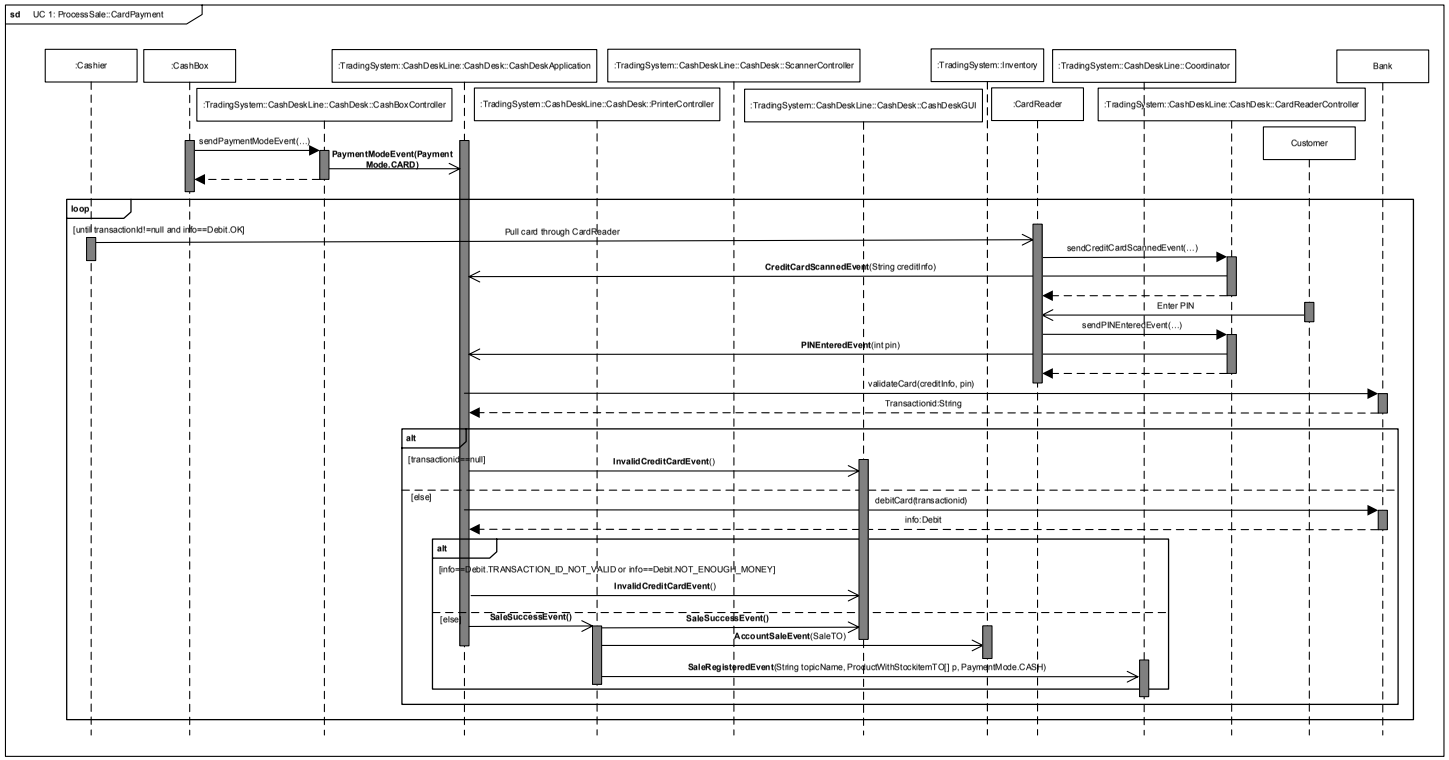

Fig. 17. Sequence diagram of the process of card payment (UC 1).

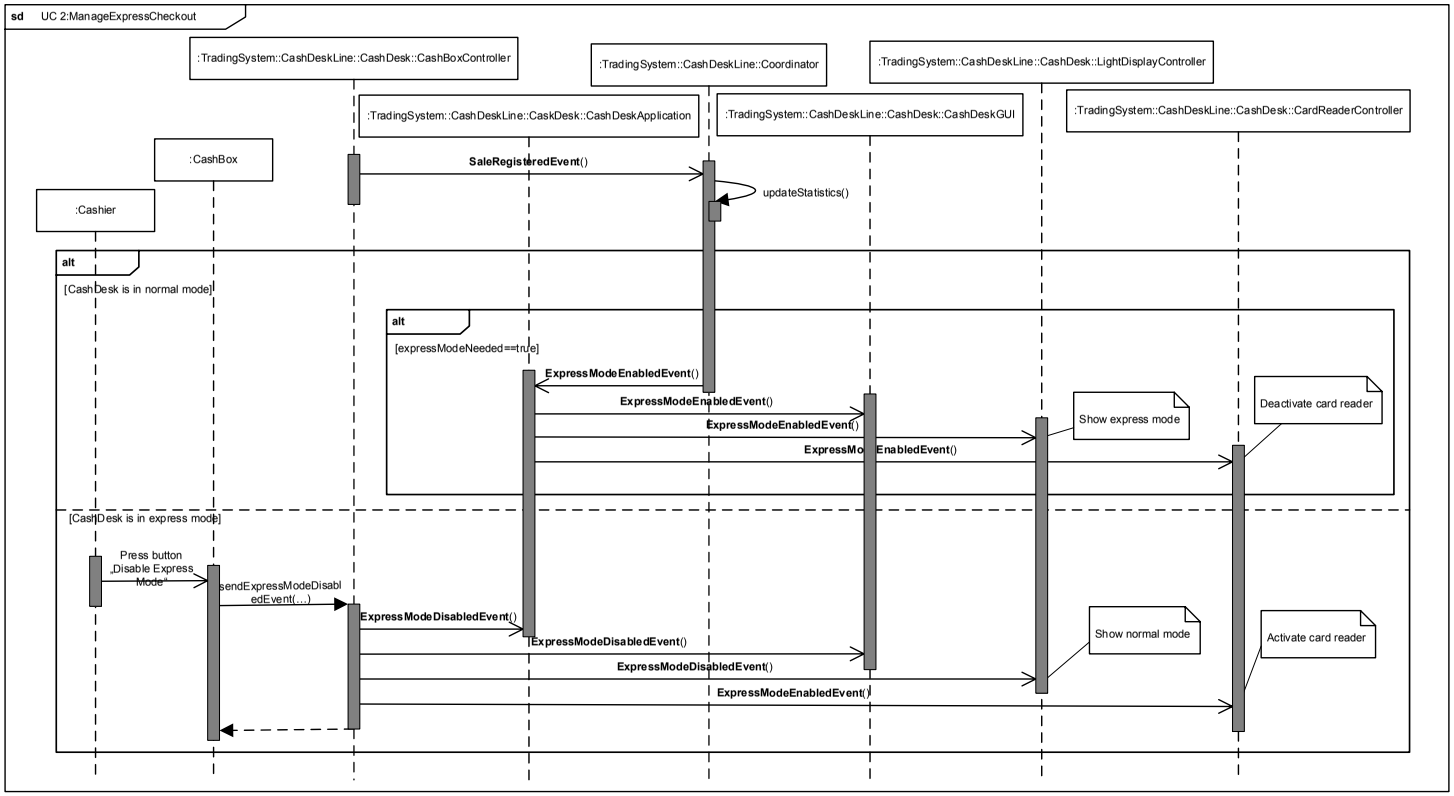

Fig. 18. Sequence diagram of managing the express checkout functionality (UC 2).

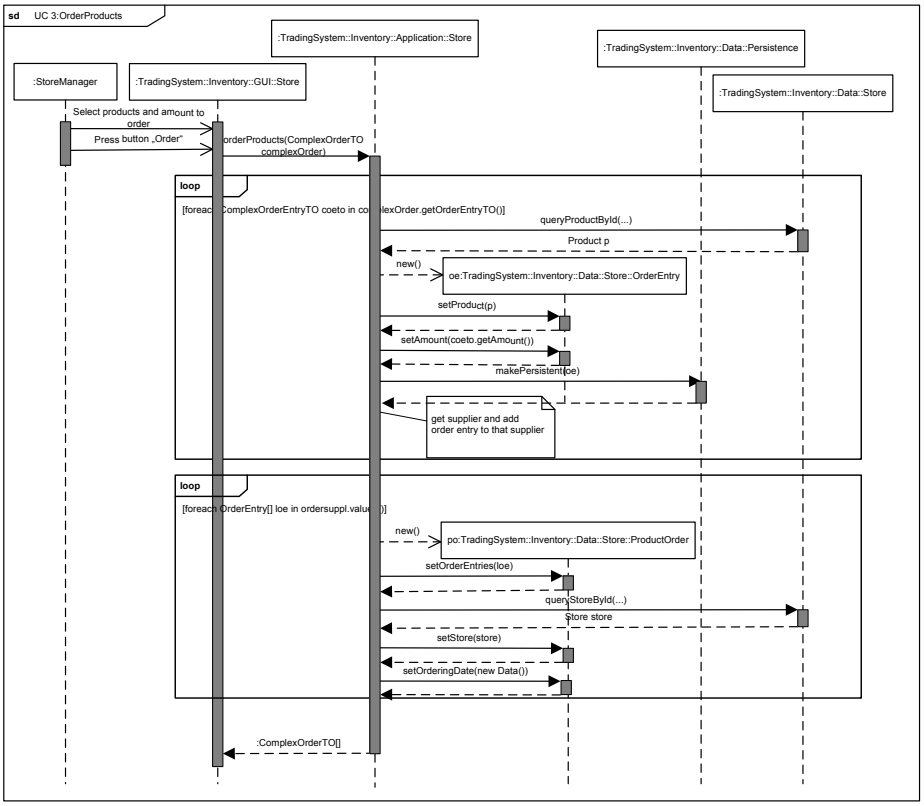

Fig. 19. Sequence diagram of ordering products (UC 3).

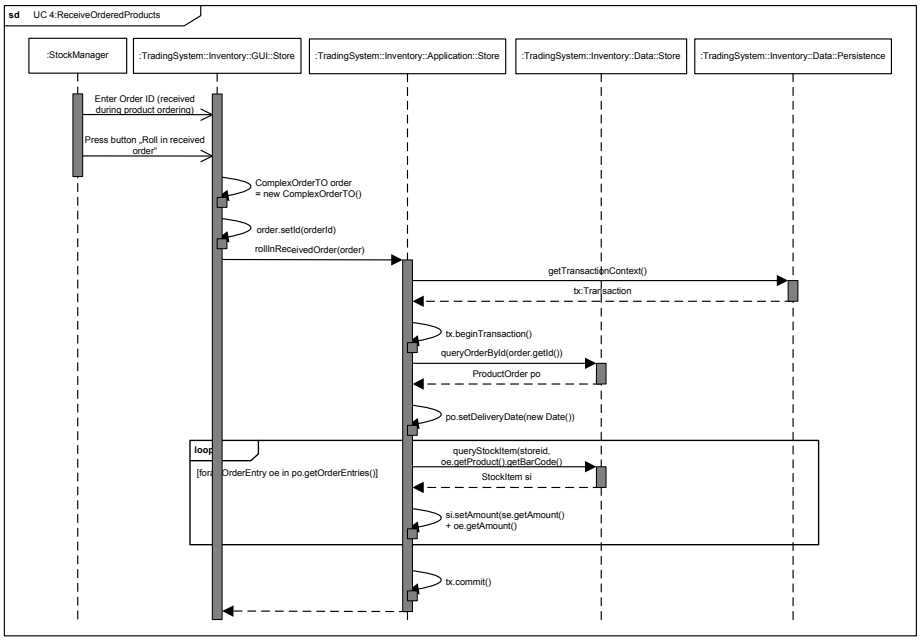

Fig. 20. Sequence diagram of receiving ordered products (UC 4).

according information using the component *GUI::Reporting*. The Store Manager therefore simply enters the store identifier and presses the button *Create Report*. Then the method *getStockReport()* is called at the component *Application::Reporting* which itself accesses the data layer component *Data::Store* in order to get the required information and to create the report as depicted in figure 21. The result is a object *ReportTO* which is then sent to the component GUI::Reporting and shown to the Store Manager at the GUI.

Behavioral View on UC 6 - Show Delivery Reports This use case is very similar to use case 5 but in this case the Enterprise Manager wants to know the mean time to the delivery of certain products. The Enterprise Manager therefore enters an enterprise identifier and presses the button *Create Report*. As depicted in figure 22 this report is created using the data layer component *Data::Enterprise* to get the required information.

Behavioral View on UC 7 - Change Price The Store Manager is able to change the sales price using the Store GUI at the Enterprise Client. Therefor the Store Manager simply selects the desired product and changes the price in the shown table. After pressing *Enter*, the new price will persistently be written into the database as described in figure 23.

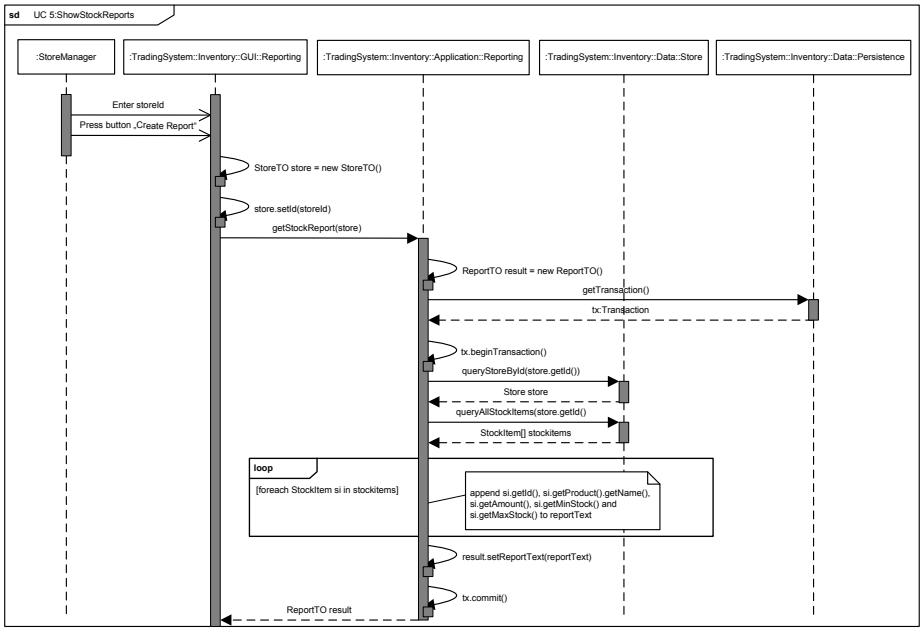

Fig. 21. Sequence diagram of getting stock reports (UC 5).

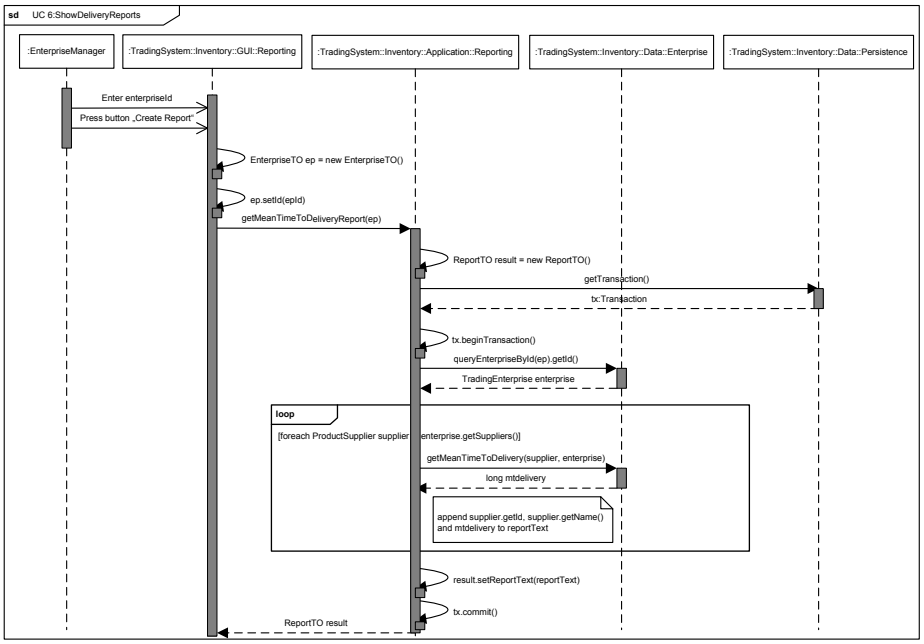

Fig. 22. Sequence diagram of getting delivery reports (UC 6).

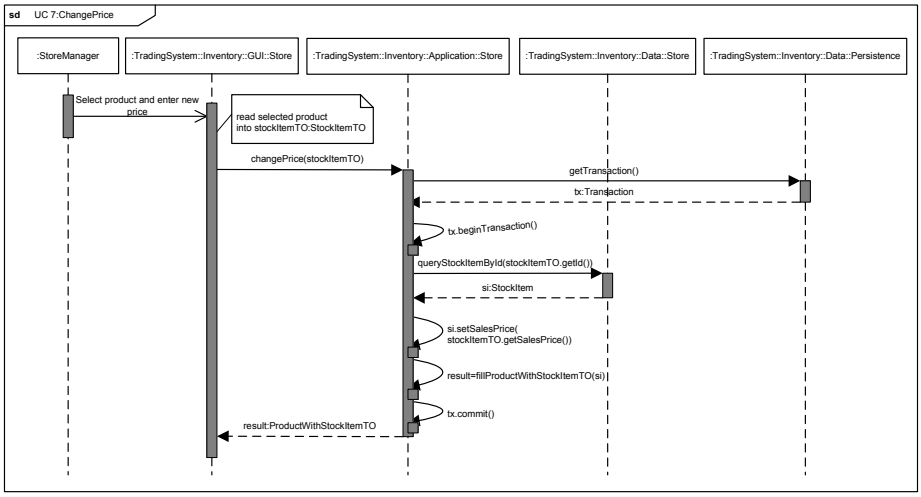

Fig. 23. Sequence diagram of changing the sales price of a product (UC 7).

Behavioral View on UC 8 - Product Exchange among Stores

The main aspect of use case 8 is a more complex interaction of distributed components and servers. If the stock of a certain product or a number of products of a Store runs low, the application Inventory of the Store Server can start a request for that product. The component ProductDispatcher of the application running on the Enterprise Server initiates a cache flush of all Store Servers to update the central database at the Enterprise Server with the latest stock data of all Stores. The component ProductDispatcher is also responsible for calculating an optimal solution for transporting the required products from a number of Stores to the requesting Store. To determine the optimal solution only the stocks of Stores are searched which are chosen by a heuristic for effort and costs. If a cost-effective solution is found, the transportation of the required goods is initiated and the requesting Store is informed about the results of its query. The sequence diagram in Figure 24 gives an overview on the dynamics in use case 8.

1.5 Implementation

In this section some important implementation aspects are briefly introduced. This includes the code design and structure as well as some hints for how to start the prototype of the Trading System.

Design

The code follows a specific structure in order to identify software components easily. A component is mapped to a Java package. The interfaces a component

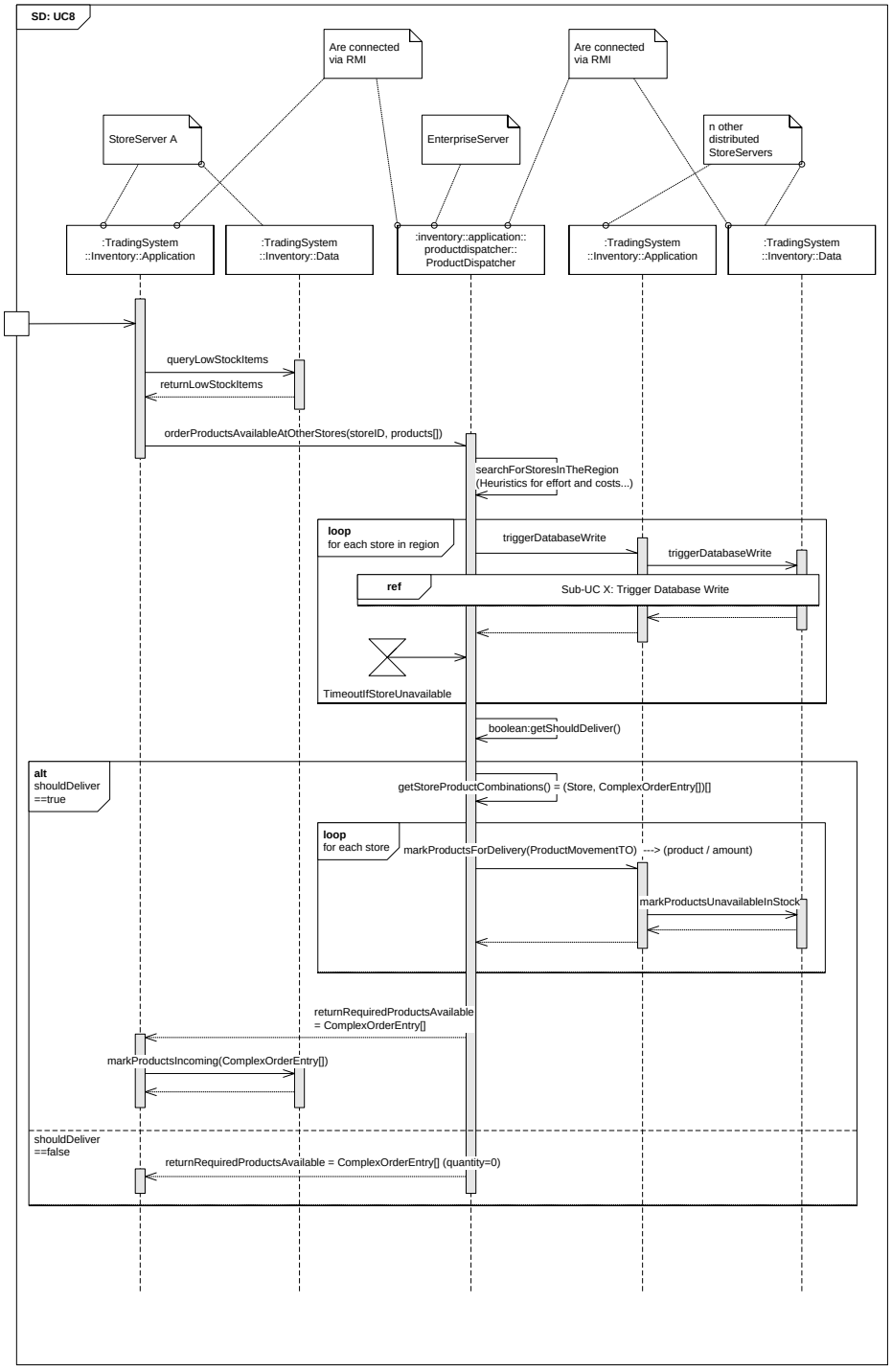

Fig. 24. Sequence diagram of exchanging products among stores (UC 8).

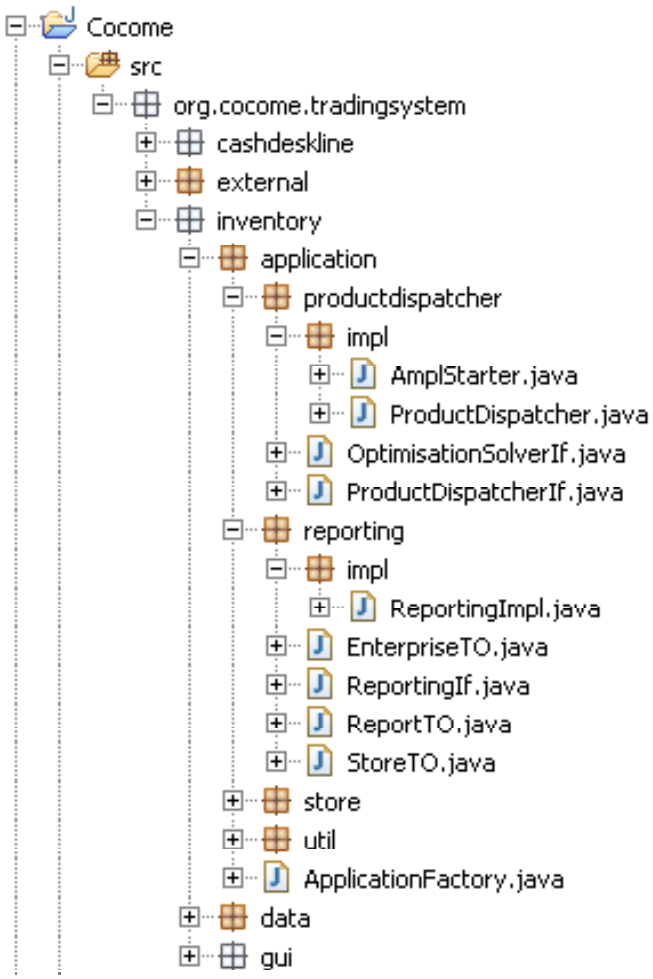

Fig. 25. Excerpt of classes and packages to depict the code structure.

supports is located in that package. The implementation classes of the interfaces are located in a subpackage called *impl*. In that way the diagrams presented in this chapter can be mapped to their corresponding code in the implementation of the Trading System.

How to start the Trading System?

The prototype can be started using Ant ([7]). The required targets are located in a file named build.xml. Furthermore a configuration file tradingsystem.properties exists where, for example, the number of Stores or Clients can be maintained. Further and detailed information can be found in the file readme.txt and in the code comments.

1.6 System Tests

In this section an overview of the system testing framework for the Trading System is given. The system tests are intended to be used in two ways: First to further detail certain aspects of the system and second to allow the different modeling teams to test their resulting implementation with a common set of (preferably automated) tests.

Test Architecture and Organisation

For a precise specification of the test scenarios and an easy way for automation of test execution it was decided to write the test cases using the *Java* programming language and the *JUnit* ([8]) testing framework as far as possible. Even when specifying the test scenarios in Java these were specified using self-explanatory identifiers and method names thus the test scenarios are also easily human readably and understandable.

Not for all of the before described use cases it was considered useful to specify a precise test scenario. Where the test conditions are not described detailed enough to judge the outcome of a test programmatically, an informal textual test script is provided instead.

The provided testing framework consists of different layers as shown in figure 26. The system tests use a common testing interface. This interface is implemented by test drivers for each implementation. The test drivers map the calls from the tests to the respective implementation.

To enable reusing the tests for several implementations of the same system, the test scenarios are implemented only against a set of interfaces describing the interactions of the Trading System with its environment. For example there is an Interface for the *Bar Code Scanner* defining a `scanBarcode` method, which is used for simulating the scanning of a product's bar code, while the interface for the *user display* provides a `isPriceShown` method which is used to check if the correct price is actually presented to the customer.

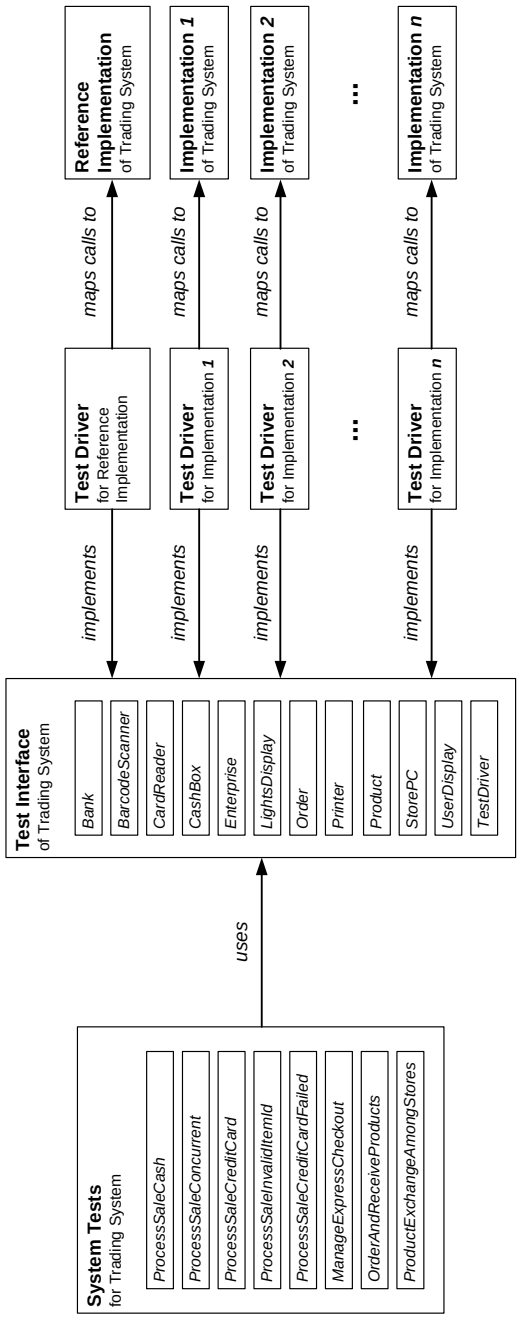

Fig. 26. Architecture of the system testing framework

The link to the system's implementation is built by an implementation-specific test driver implementing these interfaces and mapping all interface calls to corresponding system actions. This configuration allows the tests to work with unknown realizations of the system, as the knowledge about the actual implementation (which messages to send and how to identify the current state) is encapsulated in the test driver, which in turn needs to be adapted to the used implementation. Thus if the resulting implementation of a participating team should be tested, only an appropriate test driver is required, *i.e.,* a suitable implementation of the test interfaces. The specified test scenarios will remain unchanged and may potentially be executed against any trading system implementation.

As a proof-of-concept a test driver for the reference implementation of the trading system is provided. Further details on the testing system and how to write a specific test driver may be found in the corresponding source code and the *JavaDoc* documentation which comes with the source code.

Test Scenarios

The single test scenarios are based on the description of the use cases of the trading system. It was not intended to fulfill any coverage criteria or completeness in testing but rather to give examples of how such tests could look like and to provide a formalization of the use cases.

Basically there are two kinds of test scenarios: Formally described test cases written as executable Java test classes and informal descriptions of tests. The choice of the representation of a test case depends on the kind of use case.

The use cases consisting of a sequence of executions steps with a defined result are given as Java test classes using the test interfaces. These test cases could be executed automatically (by means of *JUnit*) for a specific implementation. However, the source code of these tests can also be interpreted as a formalized test plan which could be followed manually.

The remaining use cases which were not explicit enough but mainly set up a requirement for an entire piece of functionality (such as use case 5) were treated by describing the test case informally. Test cases of this form are intended to be checked manually. In table 2 the pass and fail conditions for these tests are specified.

Table 2 describes the single test cases. It states from which use case the test is derived and labels each test case with an identifier (this refers to the respective Java class of the test). For the test cases specified in Java a short description of the scenario is given. For details the reader may consult the source code of the test scenarios. For the informal stated test cases the pass and fail criteria are given instead.

Use Case	Test Case Id	Type	Description
1	ProcessSaleCash	Java	Purchase of some goods and cash payment, no exceptions.
1	ProcessSale Concurrent	Java	Concurrent purchase of some goods at n different cash desks, no exceptions.
1	ProcessSale CreditCard	Java	Purchase of some goods and credit card payment, no exceptions.
1	ProcessSale InvalidItemId	Java	Invalid item id read, manual entry of item id. (Exception in step 3 of use case 1)
1	ProcessSale CreditCardFailed	Java	Wrong PIN entry for credit card; card validation fails. (Exception in step 5.2 of use case 1)
2	ManageExpress Checkout	Java	System switches to express mode. Express mode is shown at cash desk, credit card payment not possible.
3	ShowProducts ForOrdering	informal	Store shall provide functionality to generate a report about products which are low on stock. **Test:** Check if System offers functionality for reporting products low on stock. **PASS:** Generation of report with products low on stock possible **FAIL:** Generation of report with products low on stock NOT possible
3, 4	OrderAnd ReceiveProducts	Java	Products low on stock will be ordered and correct delivery will be recorded in the inventory.

Use Case	Test Case Id	Type	Description
5	ShowStockReports	informal	System shall provide functionality to present a report including all available stock items in store or a report of cumulated available product items of a specified enterprise. **Test:** Check if System offers functionality for generation of stock reports **PASS:** Generation of stock reports possible **FAIL:** Generation of stock reports NOT possible
6	ShowDeliveryReports	informal	System shall provide functionality to present a report showing mean time to delivery for each supplier of a specific enterprise. **Test:** Check if System offers functionality for generation of a delivery report **PASS:** Generation of delivery report possible **FAIL:** Generation of delivery report NOT possible
7	ChangePrice	informal	System shall provide functionality to change sales price of a product. **Test:** Check if System offers functionality for change sales price of a product **PASS:** Change of sales price for product item possible **FAIL:** Change of sales price for product item NOT possible
8	ProductExchange AmongStores	Java	After a sale which leads to a product being low on stock of the store, product exchange between stores should take place.

Table 2: Test Scenarios for Trading System

1. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition). Prentice Hall PTR (2004)
2. OMG, Object Management Group: UML Profile for Schedulability, Performance and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02 (2005)
3. SUN Microsystems: The Java Database Connectivity (JDBC). (http://java.sun.com/javase/technologies/database/index.jsp)
4. JBoss (Red Hat Middleware): Hibernate. (http://www.hibernate.org)
5. SUN Microsystems: Java Persistence API. (http://java.sun.com/javaee/technologies/persistence.jsp)
6. SUN Microsystems: Java Message Service. (http://java.sun.com/products/jms/)
7. Apache: The Apache Ant Project. (http://ant.apache.org)
8. JUnit: JUnit. (http://www.junit.org)