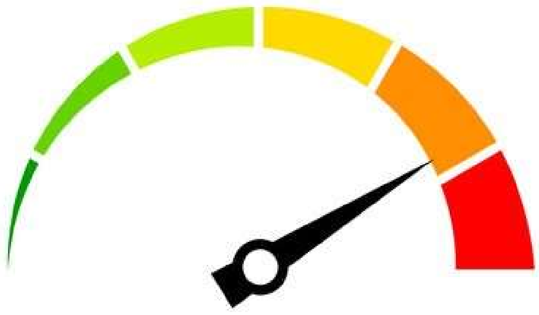
California State University Fullerton
CPSC 462

Object Oriented Software Design Design Model for the



High Velocity Sales Technology System

Ryan McDonald
Senior Architect
rtmcdonald96@csu.fullerton.edu

Alexander Frederick
Senior Technical Director
afrederick2@csu.fullerton.edu

Benjamin Baesu
Senior Localization SW Designer
BBaesu@csu.fullerton.edu

Revision History:

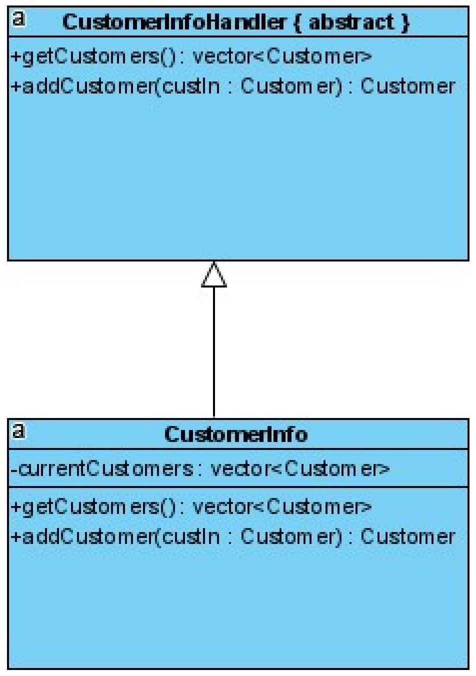
Version	Date	Summary of Changes	Author
1.0	November 9, 2020	• Initial Release	Alexander Frederick Ryan McDonald
1.1	December 7, 2020	• Added Payment Handler • Added CheckPayment • Edited: Dynamic views to better match package naming and function definitions	Alexander Frederick Ryan McDonald

Table of Contents

1 Static View

1.1 Customer Information

1.1.1 Software Class Diagram

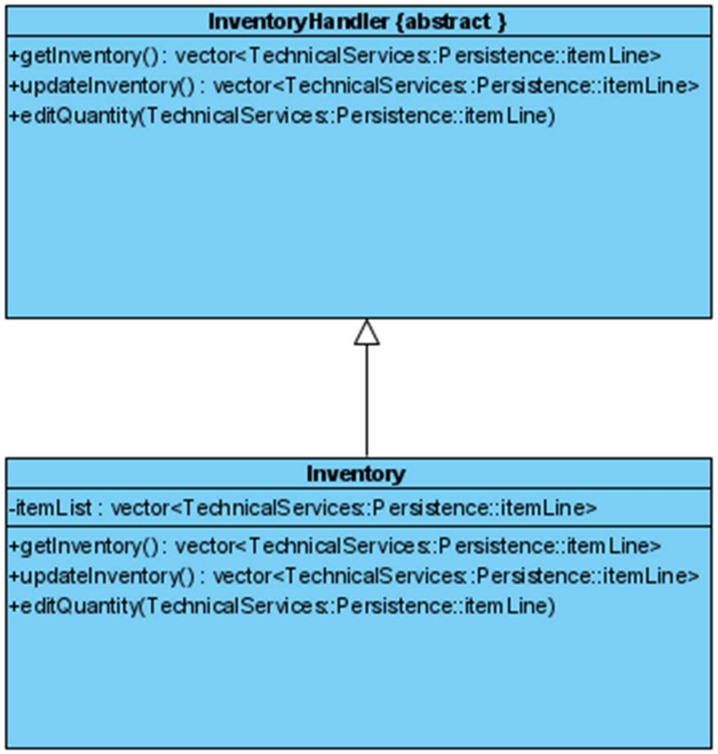

CustomerInfoHandler { abstract }
+getCustomers() : vector<Customer>
+addCustomer(custIn : Customer) : Customer
CustomerInfo
-currentCustomers : vector<Customer>
+getCustomers() : vector<Customer>
+addCustomer(custIn : Customer) : Customer

1.1.2 Description

CustomerInfo is meant to act as a controller over customer data, it acquires the data from the technical services layer and returns it back to the session controller in the Domain layer. It also is meant to handle the addition or removal of the data from the session.

1.2 Inventory Handler

1.2.1 Software Class Diagram

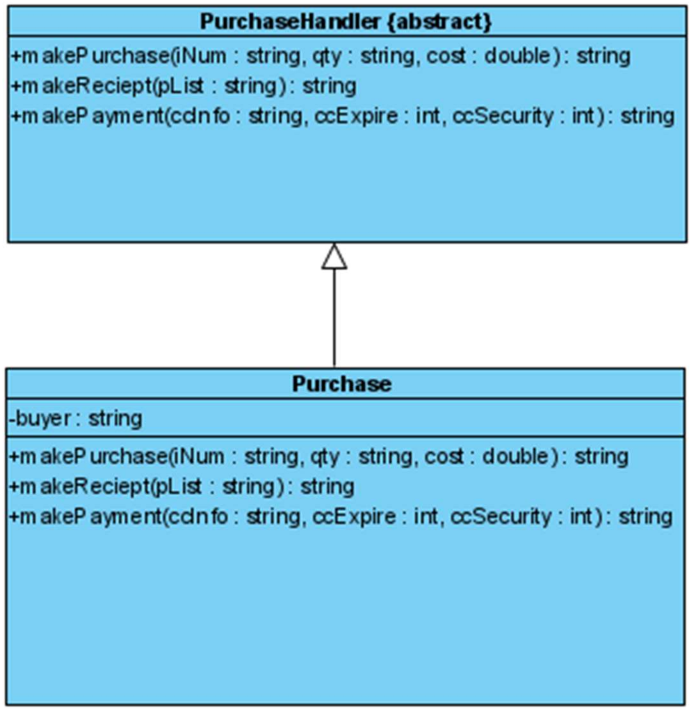

InventoryHandler {abstract }
+getInventory() : vector<TechnicalServices::Persistence::itemLine>
+updateInventory() : vector<TechnicalServices::Persistence::itemLine>
+editQuantity(TechnicalServices::Persistence::itemLine)
Inventory
-itemList : vector<TechnicalServices::Persistence::itemLine>
+getInventory() : vector<TechnicalServices::Persistence::itemLine>
+updateInventory() : vector<TechnicalServices::Persistence::itemLine>
+editQuantity(TechnicalServices::Persistence::itemLine)

1.2.2 Description

The InventoryHandler acquires the inventory data from the technical services layer, which then can be edited or viewed by the session controller object. It is meant to be used as an information expert for the inventory data.

1.3 Purchase Handler

1.3.1 Software Class Diagram

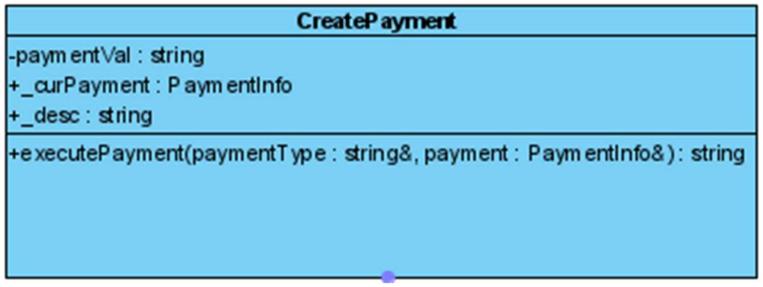

PurchaseHandler {abstract}
+makePurchase(iNum : string, qty : string, cost : double) : string
+makeReciept(pList : string) : string
+makePayment(ccInfo : string, ccExpire : int, ccSecurity : int) : string
Purchase
-buyer : string
+makePurchase(iNum : string, qty : string, cost : double) : string
+makeReciept(pList : string) : string
+makePayment(ccInfo : string, ccExpire : int, ccSecurity : int) : string

1.3.2 Description

PurchaseHandler creates purchase objects from data passed from the session into it. It then computes the information required to produce a receipt and contact the 3rd party payment software to ensure purchase is completed.

1.4 CreatePayment

1.4.1 Software Class Diagram

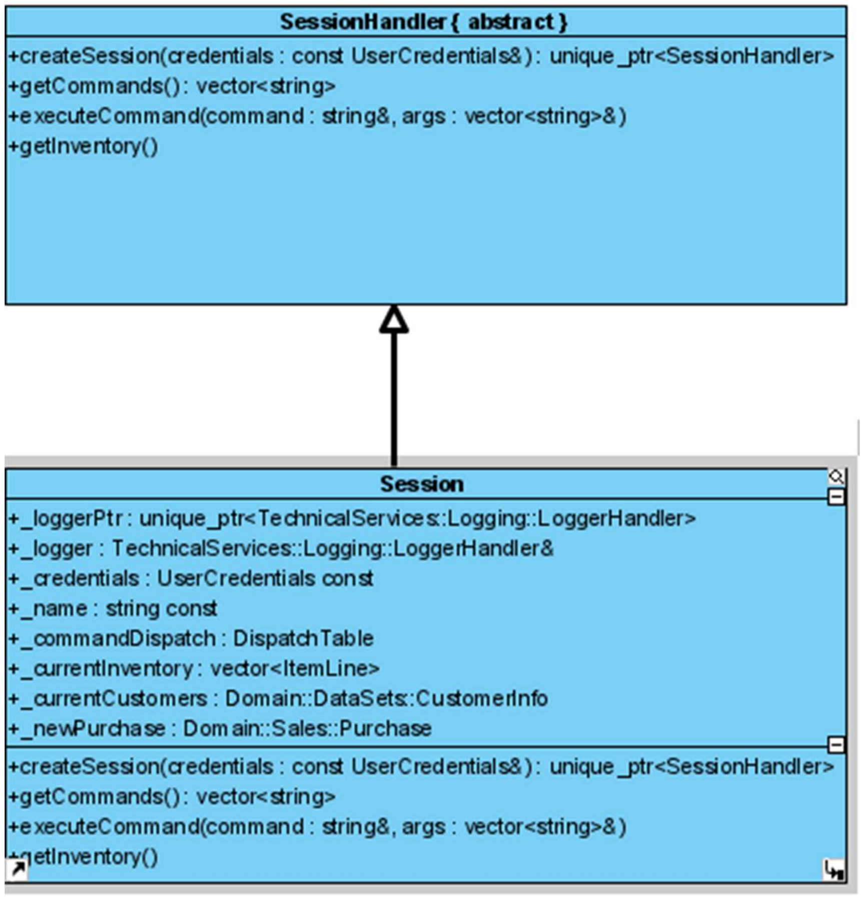

CreatePayment
-paymentVal : string
+_curPayment : PaymentInfo
+_desc : string
+executePayment(paymentType : string&, payment : PaymentInfo&) : string

1.4.2 Description

CreatePayment is used to interface the purchase functionality to the technical services layer payment functionality. This creates a lower level of coupling and results in the ability to more fluidly make adjustments to the payment system in case of loss of external payment interface.

1.5 Session Handler

1.5.1 Software Class Diagram

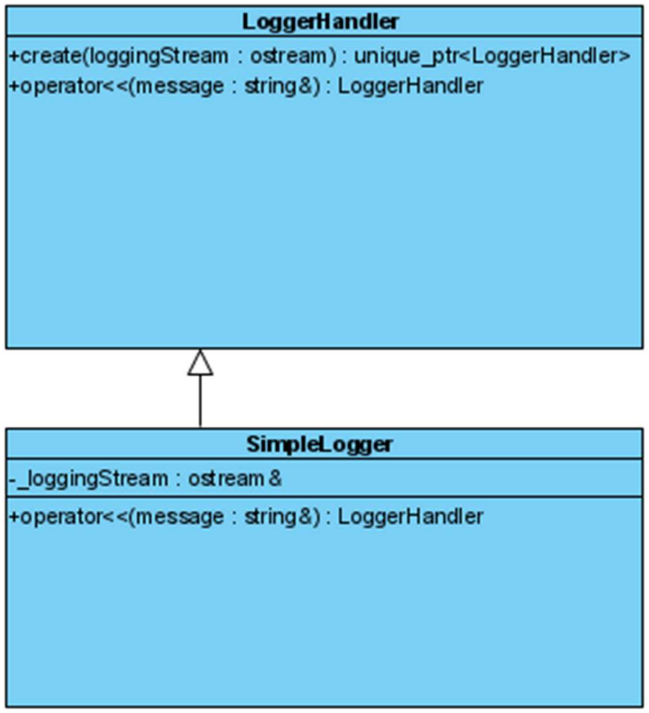

SessionHandler{ abstract }
+createSession(credentials : const UserCredentials&) : unique_ptr<SessionHandler>
+getCommands() : vector<string>
+executeCommand(command : string&, args : vector<string>&)
+getInventory()
Session
+_loggerPtr : unique_ptr<TechnicalServices::Logging::LoggerHandler>
+_logger : TechnicalServices::Logging::LoggerHandler&
+_credentials : UserCredentials const
+_name : string const
+_commandDispatch : DispatchTable
+_currentInventory : vector<ItemLine>
+_currentCustomers : Domain::DataSets::CustomerInfo
+_newPurchase : Domain::Sales::Purchase
+createSession(credentials : const UserCredentials&) : unique_ptr<SessionHandler>
+getCommands() : vector<string>
+executeCommand(command : string&, args : vector<string>&)
+getInventory()

1.5.2 Description

Session Handler is the overall controller/creator for the application. It will maintain objects that refer to the UserCredentials roles, and perform operations specific to those roles. All functionality should be based within the Session objects control in order to maintain a simple and robust hierarchy.

1.6 Logger Handler

1.6.1 Software Class Diagram

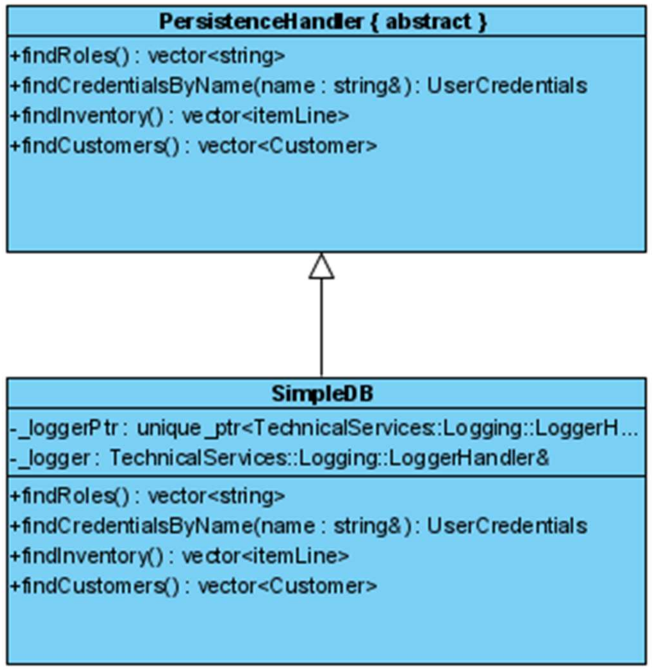

LoggerHandler
+create(loggingStream : ostream) : unique_ptr<LoggerHandler>
+operator<<(message : string&) : LoggerHandler
SimpleLogger
-_loggingStream : ostream&
+operator<<(message : string&) : LoggerHandler

1.6.2 Description

LoggerHandler is to be used to log actions taken within the domain of the system. Useful for troubleshooting, pathing of objects, and maintaining a list of actions performed.

1.7 Persistence Handler

1.7.1 Software Class Diagram

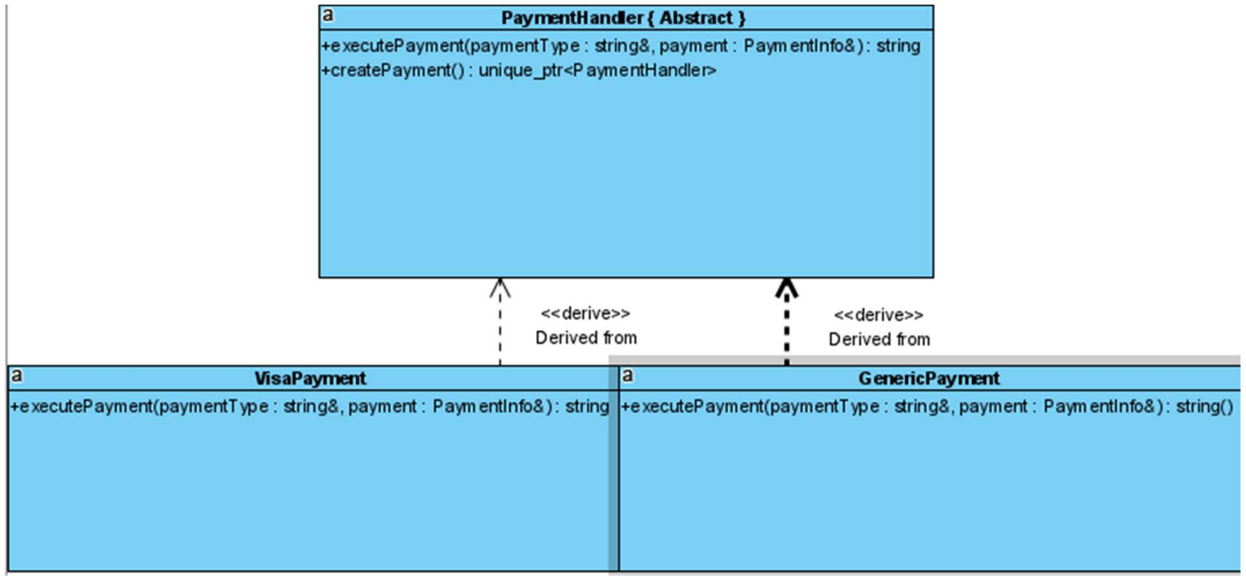

PersistenceHandler { abstract }
+findRoles() : vector<string>
+findCredentialsByName(name : string&) : UserCredentials
+findInventory() : vector<itemLine>
+findCustomers() : vector<Customer>
SimpleDB
-_loggerPtr : unique_ptr<TechnicalServices::Logging::LoggerH...
-_logger : TechnicalServices::Logging::LoggerHandler&
+findRoles() : vector<string>
+findCredentialsByName(name : string&) : UserCredentials
+findInventory() : vector<itemLine>
+findCustomers() : vector<Customer>

1.7.2 Description

PersistenceHandler is a mock-up of a simple database for our application to pull information from. Should be replaced by an external database system provided by another service.

1.8 Payment Handler

1.8.1 Software Class Diagram


a
PaymentHandler { Abstract }
+executePayment(paymentType : string&, payment : PaymentInfo&) : string
+createPayment() : unique_ptr<PaymentHandler>
<<derive>>
Derived from
<<derive>>
Derived from
a
VisaPayment
+executePayment(paymentType : string&, payment : PaymentInfo&) : string
a
GenericPayment
+executePayment(paymentType : string&, payment : PaymentInfo&) : string()

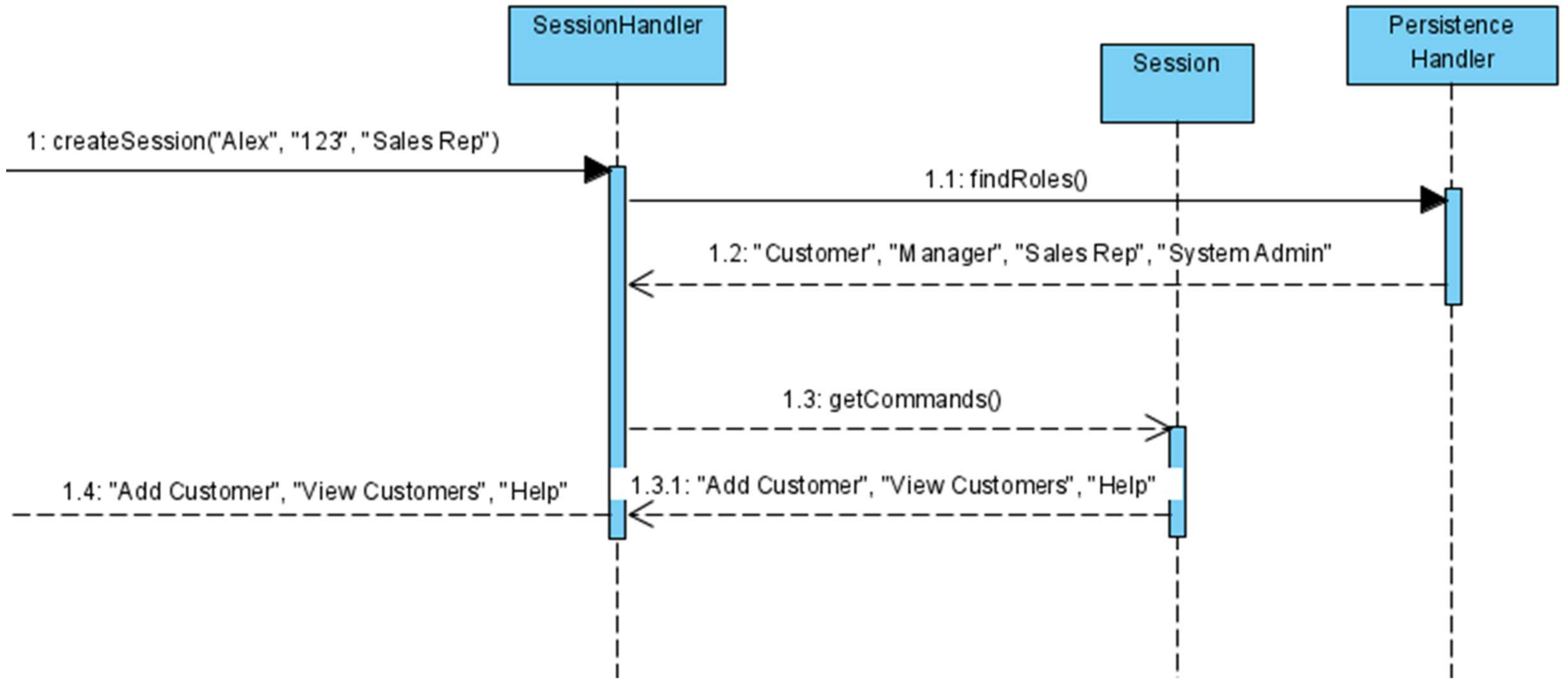
1.8.2 Description

Payment Handler is the abstract factory object that we are using to create a 3rd party payment interface. This object creates 1 of 2 payment objects based on the Component.Payment value within the .Dat file contained within the source code folders.

2 Dynamic View

2.1 Create Session Sequence of Execution

2.1.1 Software Interaction Diagram


SessionHandler
Session
Persistence Handler
1: createSession("Alex", "123", "Sales Rep")
1.1: findRoles()
1.2: "Customer", "Manager", "Sales Rep", "System Admin"
1.3: getCommands()
1.3.1: "Add Customer", "View Customers", "Help"
1.4: "Add Customer", "View Customers", "Help"

2.1.2 Description

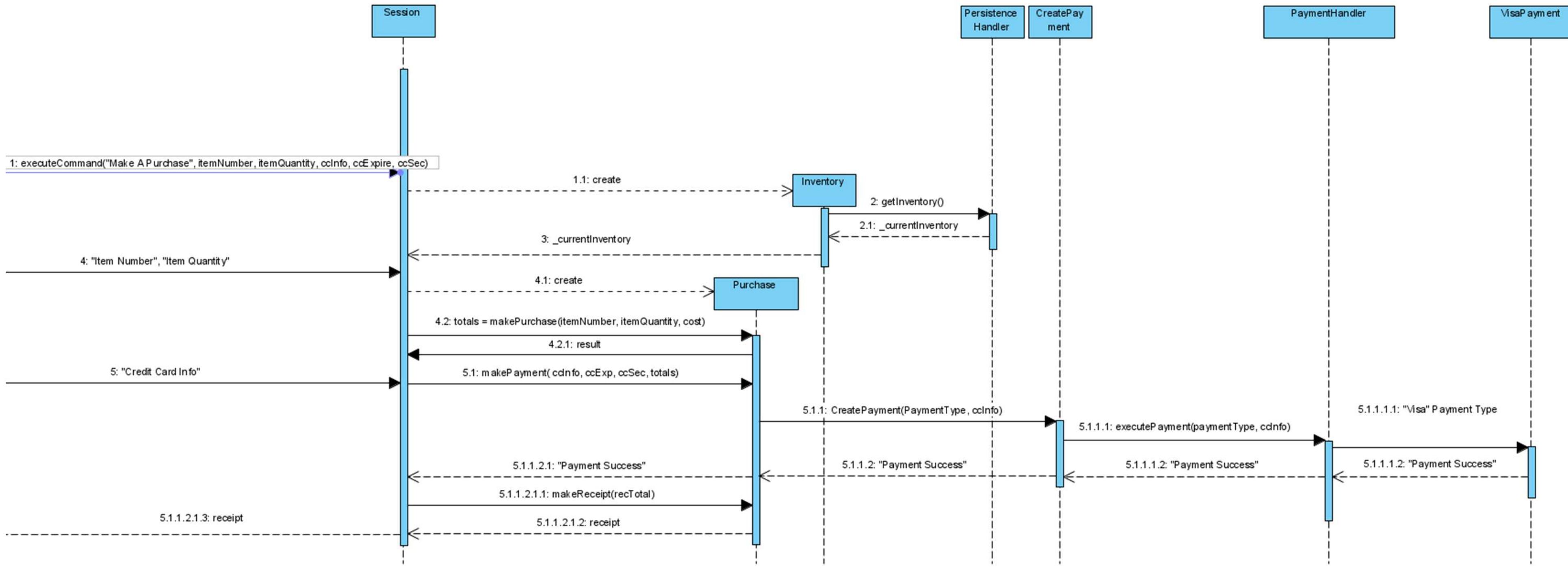
Upon the start of our system, we expect a createSession call to be received by the SessionHandler. The SessionHandler will then find the roles available from PersistenceHandler. Upon receiving the roles, we will authenticate the user and match it to a role in order to getCommands from the session object. We will then return the menu choices available to that role to the user.

2.1.3 SSD Traceability

This message is the start of any of the SSDs in our Use Cases. In every SSD we need to begin by creating a session for the user that is accessing our system.

2.2 Make A Purchase Sequence of Execution

2.2.1 Software Interaction Diagram


Session
Persistence Handler
CreatePay ment
PaymentHandler
VisaPayment
1: executeCommand("Make A Purchase", itemNumber, itemQuantity, ccInfo, ccExpire, ccSec)
1.1: create
Inventory
2: getInventory()
2.1: _currentInventory
3: _currentInventory
4: "Item Number", "Item Quantity"
4.1: create
Purchase
4.2: totals = makePurchase(itemNumber, itemQuantity, cost)
4.2.1: result
5: "Credit Card Info"
5.1: makePayment(ccInfo, ccExp, ccSec, totals)
5.1.1: CreatePayment(PaymentType, ccInfo)
5.1.1.1: executePayment(paymentType, ccInfo)
5.1.1.1.1: "Visa" Payment Type
5.1.1.2.1: "Payment Success"
5.1.1.2: "Payment Success"
5.1.1.1.2: "Payment Success"
5.1.1.1.2: "Payment Success"
5.1.1.2.1.1: makeReceipt(recTotal)
5.1.1.2.1.3: receipt
5.1.1.2.1.2: receipt

2.2.2 Description

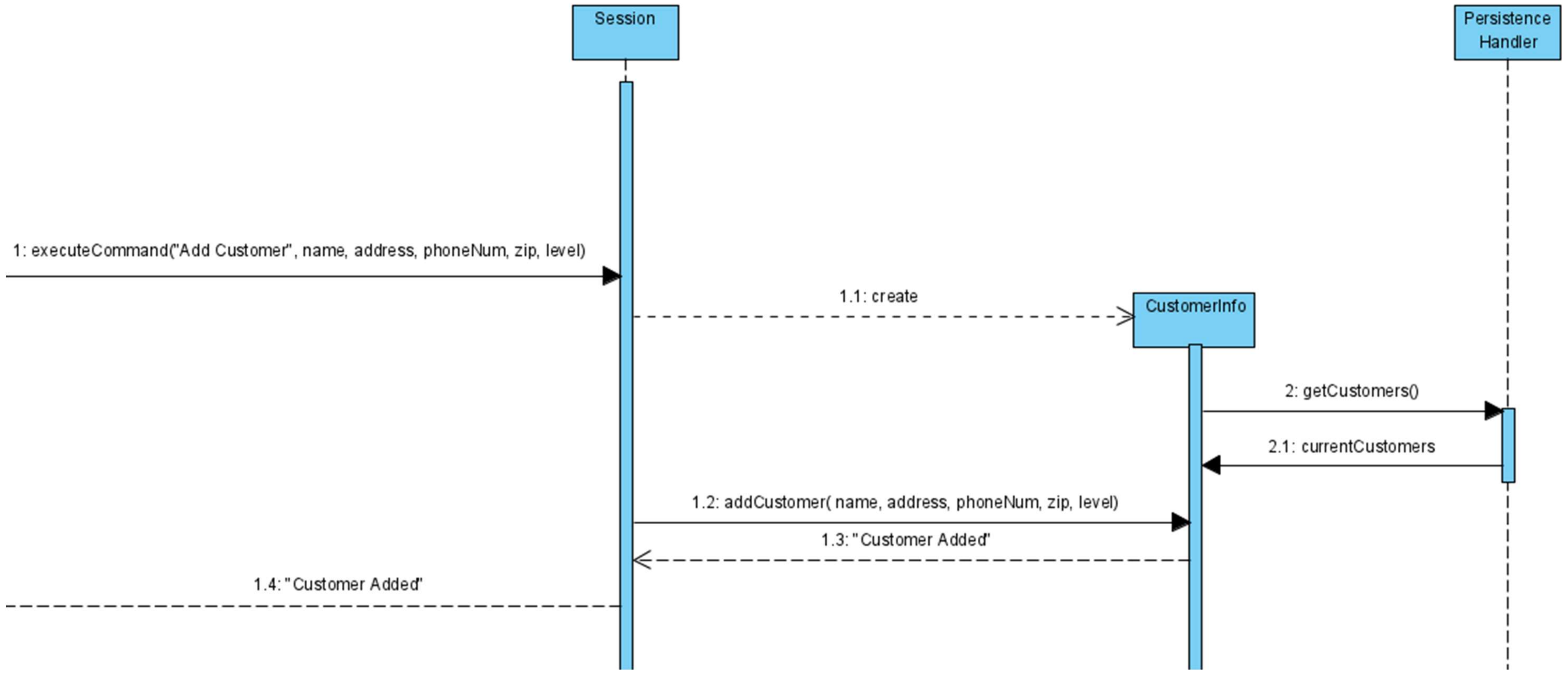
2.2.3 The make a purchase sequence begins with the session receiving command execute command with “Make A Purchase” and the arguments waiting to be filled: item number for the item they want, the quantity, and their credit card information. The session instantiates a new Inventory object. The inventory object calls the getInventory function to gather the current inventory from the persistence handler. The session receives the item number and quantity from the customer. Then the session instantiates a new Purchase object. The Purchase object is passed the item number, quantity and cost to generate a total with tax in the makePurchase function. Once the result is returned to the session, the session receives the credit card information, and the purchase is passed the credit card information and the total cost in order to make a payment. Make Payment is then called by the Session with the credit card information and the total cost. This calls the create payment command which executes the execute payment command with the payment handler which reaches out to the VisaPayment interface for 3rd party payment authorization. Once the payment is made a “Payment success is returned to the Payment Handler, which returns to the Create Payment, which returns to Purchase, which returns to the Session. After receiving the successful payment, the Session calls the makeReceipt command with the total cost. The Purchase object returns the receipt to the Session which in turn returns the receipt.

2.2.4 SSD Traceability

The source SSD for this software interaction diagram is in the Use Case Model Annex 1. This interaction is an example of what happens in Scenario 1 at the end of the document. The name of this message corresponds to the Make A Purchase command within the session component in the domain of our system. This software interaction diagram goes through process from what would happen once the customer enters the Make A Purchase command and corresponding information.

2.3 Add Customer Sequence of Execution

2.3.1 Software Interaction Diagram



2.3.2 Description

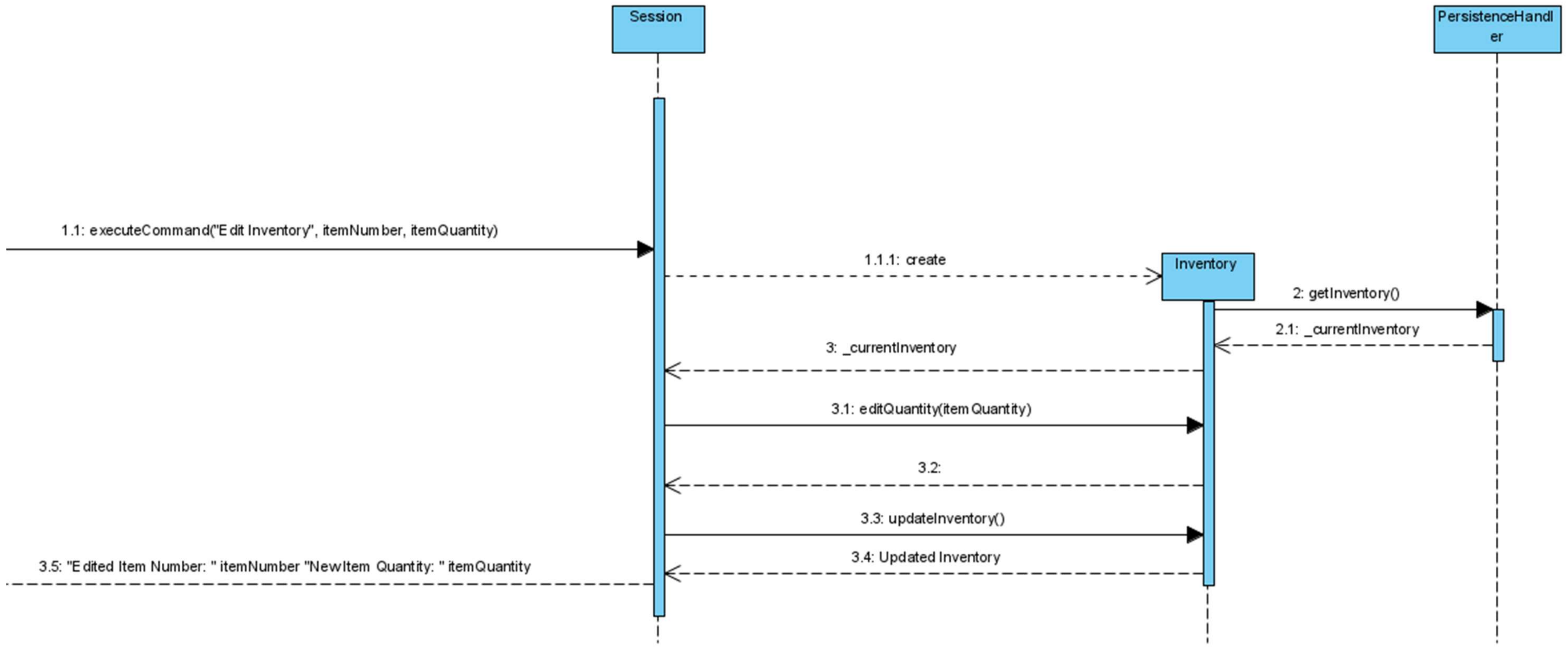
2.3.3 The Add Customer sequence begins with the session receiving the command executeCommand, with “Add Customer” and the name, address, phone number, zip code, and access level of the customer to be added . The session instantiates a CustomerInfo object which upon creation calls the getCustomers function to retrieve the current customer list from the persistent data. The UI passes this information to the Session object. Our Session object calls the addCustomer function and passes the customer information to the CustomerInfo object. This object adds the customer and returns a “Customer Added” success message to the session. The session passes this message on.

2.3.4 SSD Traceability

The source SSD for this software interaction diagram is in the Use Case Model Annex 2. This particular interaction is the Scenario 1 at the end of the document. The name of this message corresponds to the “Add Customer” command within the session component in the domain of our system. This software interaction diagram goes through process from what would happen once the sales rep enters the Add customer command and corresponding information.

2.4 Edit Inventory Sequence of Execution

2.4.1 Software Interaction Diagram


Session
PersistenceHandler
1.1: executeCommand("Edit Inventory", itemNumber, itemQuantity)
1.1.1: create
Inventory
2: getInventory()
2.1: _currentInventory
3: _currentInventory
3.1: editQuantity(itemQuantity)
3.2:
3.3: updateInventory()
3.4: Updated Inventory
3.5: "Edited Item Number: " itemNumber "NewItem Quantity: " itemQuantity

2.4.2 Description

2.4.3 The Edit Inventory sequence begins with the session receiving the command executeCommand, containing "Edit Inventory", the item number of the item to change and the quantity change it to. Upon receiving this command the session instantiates an inventory object, this inventory object calls the getInventory function which retrieves the current inventory from the persistent data. After the current inventory is received the session calls the editQuantity function for the item within the inventory that matches the itemNumber passed by the UI and edits the quantity for that item. Once the quantity has been edited, the session calls the updateInventory command which updates the inventory contained within the inventory object so that it is up to date. The inventory object returns the updated inventory to our session and then the session returns the edited item number along with the quantity that was edited.

2.4.4 SSD Traceability

The source SSD for this software interaction diagram is in the Use Case Model Annex 3. This interaction is an example of what happens in Scenario 1 at the end of the document. The name of this message corresponds to the Edit Inventory command within the session component in the domain of our system. This software interaction diagram goes through process from what would happen once the manager enters the Edit Quantity command and orresponding information.