An Exercise in Object-Oriented Analysis and Design

By

Herbert Williams III

A MASTER OF ENGINEERING REPORT

Submitted to the College of Engineering at

Texas Tech University in

Partial Fulfillment of

The Requirements for the

Degree of

MASTER OF ENGINEERING

Approved

Dr. A. Ertas

Dr. T. Maxwell

Dr. M. Tanik

Dr. Chris Letchford

October 14, 2006

ACKNOWLEDGEMENTS

I would like to express my thanks to all the teachers who have influenced my love of learning and computer programming. The road to reach this milestone has been long with many stops along the way. Every class in which the teachers have allowed me to express my creativity and apply lessons learned has resulted in my desire to continue expanding my education and attempting new challenges.

I would also like to thank my patient wife and coworkers who have all been overwhelming supportive and understanding over the course of my studies. There have been many long days in class and nights at the computer, but I never had return to an angry wife or frustrated team.

TABLE OF CONTENTS

DISCLAIMER

ABSTRACT

This paper is an application of object-oriented analysis and design (OOAD) techniques as presented in educational texts and practiced in commercial software development. After a discussion of OOAD's place among software engineering techniques, a problem is defined from which a software solution can be engineered. Having defined this problem, requirements are gathered, analysis and design are performed, and finally some discussion is had regarding implementation decisions. An appendix entry is included which describes research performed in learning the presentation implementation options available to Java developers. This extended research was performed because of the open ended implementation options available to Java developers in the presentation tier.

The purpose of this paper is not to document in detail each OOAD technique available. Instead, the purpose is to apply commonly used and defined methods to a problem more complicated than those presented by basic texts. This serves as a personal learning experience and provides a medium to practice techniques which are important to a thoughtful design, but often overlooked outside of educational circles when faced with schedule restrictions.

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

Software Engineering is a very broad discipline which is involved with the complete lifecycle of software from the earliest stages of specification through the end of maintenance support. The activities software engineering is involved with are not only technical, but are also concerned with management activities such as process and requirements.

Particular attention is paid to process definition in software engineering, and many models have been proposed which all aim to achieve the same task, define a "framework for the tasks that are required to build high-quality software." [Pressman 2001]. Traditional software engineering approaches proposed methods for developing structured applications. Most of these approaches often followed an iterative lifecycle, accomplishing portions of a project at a time. More advanced versions evolved out of these to address customer communication, problem detection, and reuse issues which arose with the simpler models.

All models provide a similar set of stages which address the necessary steps to produce a quality software application. A generalization of these would be Requirements Gathering, Analysis, Design, Implementation, Testing, and Deployment [Alhir 2003].

Depending on the model, these may be broken down into more specific activities such as Risk Analysis, Customer Communication or Product Evaluation. Also, these tasks often repeat numerous times throughout the lifecycle of the application.

Currently, a very popular application of software engineering is designing object-oriented applications. Simply known as object-oriented software engineering, this approach views the problem domain as a set of objects that have attributes and behaviors [Pressman 2001]. Object-oriented software engineering saw its early roots form in the late 1960's but took nearly twenty years to become widely used.

As with traditional software engineering, many processes have been proposed for developing object-oriented (OO) software. In his text *Software Engineering: A Practitioners Approach,* Pressman recognizes that since OO systems tend to evolve over time, and suggests using an evolutionary model which stresses reuse.

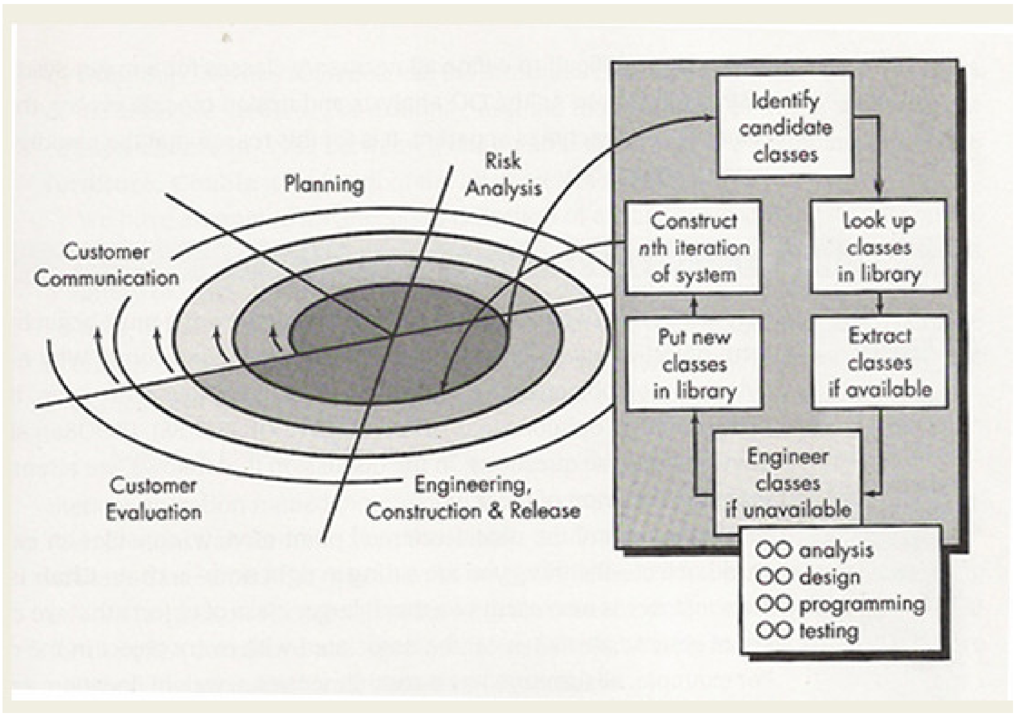

Planning
Risk Analysis
Customer Communication
Customer Evaluation
Engineering, Construction & Release
Identify candidate classes
Construct nth iteration of system
Look up classes in library
Put new classes in library
Extract classes if available
Engineer classes if unavailable
OO analysis
OO design
OO programming
OO testing

Figure 1 OO Process Model [Pressman 2001]

Figure 1 OO Process Model [Pressman 2001] shows Pressman's suggested process model. Extra emphasis is given to the "Engineering, Construction, & Release" section. This shows how the model promotes reuse by identifying, locating, and developing classes for the system.

The primary difference between this model and traditional models is the OO specific analysis, design, programming, and testing. While similar at a high level to traditional programming techniques, these four steps in OO development vary drastically in practice.

Object-oriented analysis and design (OOAD) is in itself an enormously robust topic with many texts written addressing it. In fact, in Sommerville's *Software Engineering*, object-oriented software engineering is not specifically addressed. This text chooses to spend its time on object-oriented analysis design techniques since the other lifecycle stages so closely resemble traditional software engineering practices. Both Pressman and Sommerville follow similar techniques when describing the general process of OOAD. Analysis techniques help the engineer define and understand the problem domain by modeling object interfaces and behaviors [Bray 1997]. Design, happening after analysis, creates both a system design and object design with details of how the internals of each work defined.

Modeling, performed during analysis and design, is done by creating language neutral diagrams which depict the relationships of objects in the system. Unified Modeling Language (UML) is the standardized "visual language for modeling and communicating about systems through the use of diagrams and supporting text." [Alhir 2003]. UML is used extensively in modern software engineering texts and serves as an excellent tool for OOAD.

The goal of this paper is to serve as an application of object-oriented analysis and design principles as they would be used in a software engineering project. The paper will follow a sample application from functional requirement gathering, object analysis, object design, and into the implementation phase. The purpose of this exercise is to apply OOAD techniques presented in both academic and corporate publications to a real world scenario. The intent is not to explain every OOAD method available in detail, but instead focus on the analysis and design work performed and captured in the analysis and design artifacts. Instructional material often uses very simplistic examples which avoid some of the design pitfalls more complicated applications face. The problem selected for this paper presents some associations which require a more thoughtful design than is typically presented.

CHAPTER II
BACKGROUND

One of the major events every year in the wine industry is the release of the Wine Futures in France. These futures are the current year's wines which will not actually be released to the market for another two years. This event gives customers and importers the opportunity to sample these wines and start planning for their release.

Even though there is a two year wait for the wine to be released, sales of the wines from the vineyards to the customers begin immediately after the event is over. These transactions do not happen directly between the two parties. Both France and America have a tiered system of representatives acting as agents and middlemen. In America, there are three parties involved: the Importer, the Distributor, and the Customer. Sometimes, the importers also act as distributors eliminating one rung of the ladder on the United States side.

The importer places orders with the representative in France takes orders from customers in America. In France there are two parties who are of interest to this problem: the Chateau (or vineyard) and the Negotiante. The Negotiante acts as a chateau's agent in selling the wine to the importers. To complicate matters, Negotiantes can represent multiple vineyards and one vineyard can have multiple Negotiantes. This results in American importers receiving offers from multiple Negotiantes, sometimes for the same wine. Each of these offers has a price and quantity available associated with it which must be tracked. It is common that one wine may be offered at different prices and quantities from different Negotiantes.

On the other side of the coin, the importer takes requests from Customers desiring many wines at different quantities. In anticipation of these requests, the importer purchases wine from the negotiante to take advantage of the offer before it is gone. If the order from the customer requests more of a specific chateau than the importer has purchased, the importer must buy more from the negotiantes. After having enough wine purchased to fulfill the order, the importer must map the requests from the customers to a specific negotiantes. This is required since the negotiante will be sent a purchase order with the names of specific customers to ship to. Multiple negotiantes may be assigned to different portions of an offer depending on the quantities the importer purchased from the offers. All of this complexity needs to be sorted out by the importer to keep track of all the orders which will not even be delivered for two years.

The solution to the wine importer's problem has the potential to be a full enterprise application with detailed design required from the business logic to the presentation. This paper shall focus on the object-oriented analysis and design of the system.

CHAPTER III
REQUIREMENTS COLLECTION

Requirement gathering is an activity performed by all software projects. This activity is independent of how the solution is designed and implemented, be it through traditional or object-oriented means. Software requirements can be broken down into three categories: user requirements, system requirements, and software design specification [Sommerville 2001]. User requirements are a description of services the system is expected to provide, system requirements are detailing system services and constraints, and software design specification is an abstract of the software design. In addition, requirements can also be categorized as functional, non-functional, or domain. As their names suggest, functional requirements describe services the system should provide, non-functional describe constraints on the system's services, and domain define requirements on the application domain. While all requirement types are vitally important to an application, this paper will focus on functional user requirements to provide guidance when performing the analysis and design of the system. This will be done to provide scope to the paper and keep the focus on the OOAD processes.

3.1 Use Cases

One common tool used during requirement elicitation is the use case. A use case is "a functional requirement that is described from the perspective of the users of a system" [Alhir 2003]. In *Software Engineering: A Practitioner's Approach* Pressman notes that use cases achieve the following objectives:

- To define the functional and operational requirements of the system by defining a scenario if usage that is agreed upon by the end-user and the software engineering team.
- To provide a clear and unambiguous description of how the end-user and the system interact with one another.
- To provide a basis for validation testing.

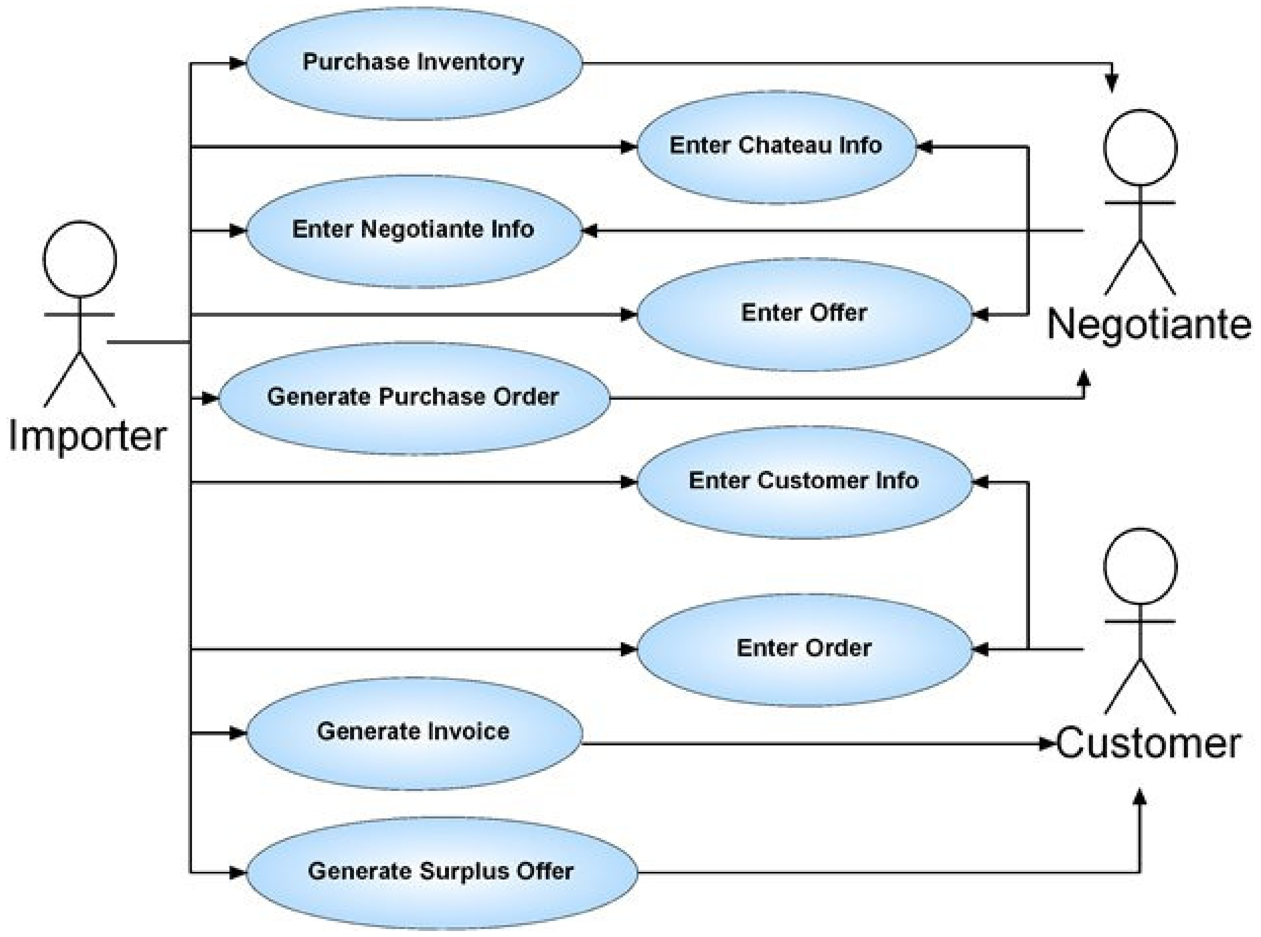


Figure 2 System Use Case

The use case diagram shown in Figure 2 shows the external actors to the system and the functional actions which can be performed on it. The three actors on the system are the importer, the negotiante, and the customer. The importer has the ability to perform all the actions on the system in the case the Negotiante or Customer do not have access. The use cases have been divided into three groups based on the primary actor. Details of each use case were captured using a standard use case template.

3.1.1 Importer Use Cases

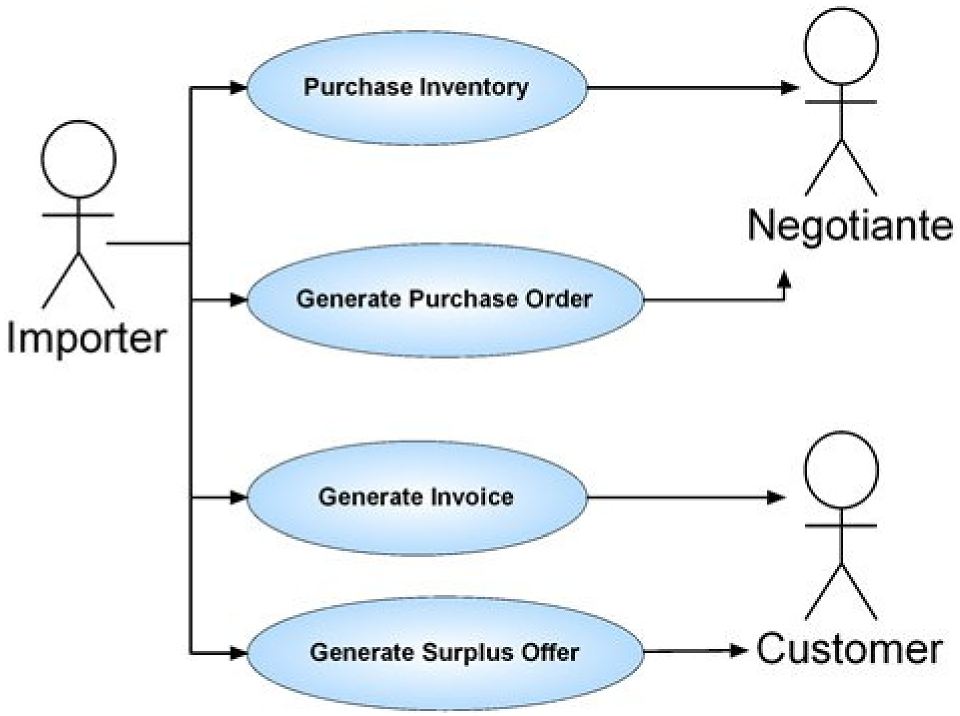


Figure 3 Importer Use Case Diagram

Figure 3 Importer Use Case contains the use cases in which the importer is the primary actor interacting with the system. Each of the use cases results in an output being received by either the negotiante or the customer.

Table 1 Purchase Inventory Use Case

Name:	Purchase Inventory
Actors:	Importer, Negotiante
Description:	The importer purchases wine from the negotiantes. Even though the wine will eventually be going to a specific customer, the importer wants to take advantage of the offer's availability, so they purchase the wine in anticipation that customers will want it. The system should keep track of these transactions and maintain a virtual inventory of what chateaus have been purchased.
Preconditions:	An offer must exist in the system to purchase from.
Basic Course of Events:	• The importer opens all offers. • The importer selects the offer and quantity to purchase • The negotiante is notified of the purchase.

Postconditions:	The offer is saved in the system and inventory updated.
Alternatives:	None
Exceptions:	None
Issues:	If the negotiante is not connected to the system, the purchase would need to be carried out external to the system and the transaction recorded through the use case.

Table 2 Generate Purchase Order Use Case

Name:	Generate Purchase Order
Actors:	Importer, Negotiante
Description:	Purchase orders are slips created and sent to negotiantes as a written contract of wine purchases. One of these is created per American customer for each negotiante. Each contains the products being purchased with the price, quantity, and size of each product included.
Preconditions:	The importer has purchased wine from a negotiante. A customer has ordered wine. The importer has associated a customer order with a negotiantes wine.
Basic Course of Events:	• The importer opens the negotiante list. • The importer selects a negotiante to work with. • The importer has the system generate purchase orders for the selected negotiante. • The system sends the purchase orders to the negotiante
Postconditions:	The negotiante receives the purchase orders
Alternatives:	None
Exceptions:	None
Issues:	If the negotiante is not connected to the system, the purchase order will need to be sent by hand.

Table 3 Generate Invoice Use Case

Name:	Generate Invoice
Actors:	Importer, Customer
Description:	Invoices are bills created and sent to the customer for the orders they purchased. One invoice is created per customer. The system should be able to create these invoices with details of each order included.

Preconditions:	The importer fulfills all customer orders
Basic Course of Events:	• The importer opens the customer list. • The importer selects a customer to work with. • The importer has the system generate an invoice for the selected customer. • The system sends the invoice to the customer
Postconditions:	The customer receives the invoice.
Alternatives:	None
Exceptions:	None
Issues:	If the customer is not connected to the system, the invoice will need to be sent by hand.

Table 4 Generate Surplus Offer Use Case

Name:	Generate Surplus Offer
Actors:	Importer, Customer
Description:	If the importer has surplus inventory after satisfying customer orders, an offer can be sent out to customers to try and offload the surplus. The details of this offer include specifics of the wines and their prices.
Preconditions:	• The importer has purchased wine from a negotiante. • A customer exists in the system. • Surplus inventory exists in the system.
Basic Course of Events:	• The importer selects the surplus to sell. • The importer specifies a quantity and price. • The importer selects the customers to send the offer. • The importer generates a surplus offer. • The importer has the system generate and offer. • The system sends the surplus offer to the customers.
Postconditions:	The customers receive the surplus offer.
Alternatives:	None
Exceptions:	None

Issues:	If the customer is not connected to the system, the surplus offer will need to be sent by hand.

3.1.2 Negotiante Use Cases

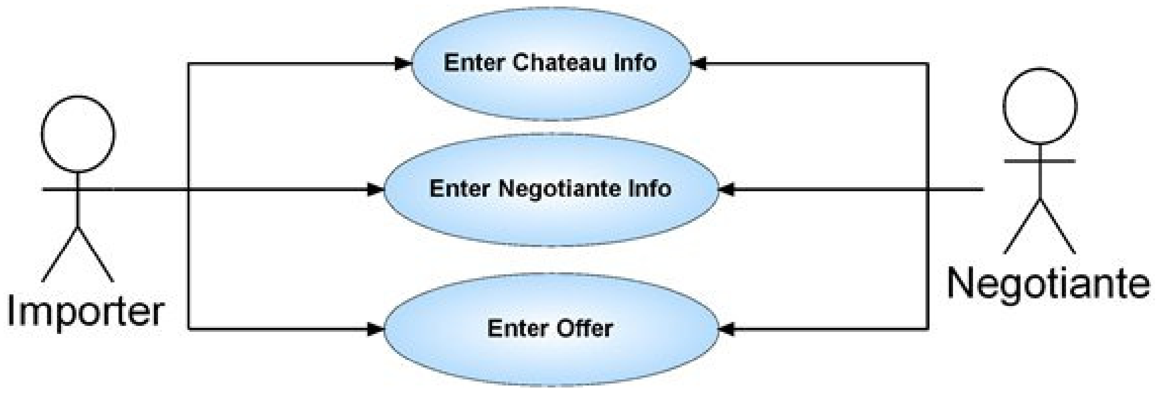


Figure 4 Negotiante Use Case Diagram

Figure 4 Negotiante Use Case Diagram contains the use cases primarily initiated by the Negotiante. The importer also has access to the system to perform the same actions as an agent of the Negotiante.

Table 5 Enter Chateau Information Use Case

Name:	Enter chateau information
Actors:	Importer, Negotiante
Description:	This use case captures the data entry required to store a Chateau in the system. Since Negotiantes represent the chateaus, they can enter information in addition to the importer. The details that matter to the user of the system besides its name are the Robert Parker score, the Wine Spectator score, and the Appalachian.
Preconditions:	None
Basic Course of Events:	• The operator begins a new customer entry • The operator enters the relavant information • The operator saves the entry.
Postconditions:	The chateau information is saved.
Alternatives:	If the chateau already exists in the system, an update is performed.

Exceptions:	None
Issues:	None

Table 6 Enter Negotiante Information Use Case

Name:	Enter negotiante information
Actors:	Importer, Negotiante
Description:	This use case captures the data entry required to store a negotiante. More information is kept regarding the negotiante since the importer works directly with them to purchase wine. The details which need to be stored in the system are the contact information, bank information, payment terms, and invoices received.
Preconditions:	None
Basic Course of Events:	• The operator begins a new negotiante entry • The operator enters the relavant information • The operator saves the entry.
Postconditions:	The negotiante information is saved.
Alternatives:	If the negotiante already exists in the system, it is updated.
Exceptions:	None
Issues:	None

Table 7 Enter Offer Use Case

Name:	Enter Offer
Actors:	Importer, Negotiante
Description:	Importers do not purchase individual chateaus from the vineyards which produce them. Instead the wine is sold through the negotiantes. Negotiantes send offers out to importers specifying a price and quantity available for each chateau they have to sell. It is quite common for the same wine to be sold through different Negotiantes in different quantities and prices. The system should be able to record these offers from negotiantes and the details associated with them.

Preconditions:	The negotiante and chateau need to exist in the system.
Basic Course of Events:	• The actor begins a new offer • The actor selects an existing chateau • The actor enters size, and quantity of the offer • The actor saves the offer.
Postconditions:	The offer is saved in the system.
Alternatives:	If the offer already exists, it is modified.
Exceptions:	None
Issues:	1. If the chateau or negotiante did not exist in the system, Enter Chateau or Enter Negotiante use case will need to be performed.

3.1.3 Customer Use Cases

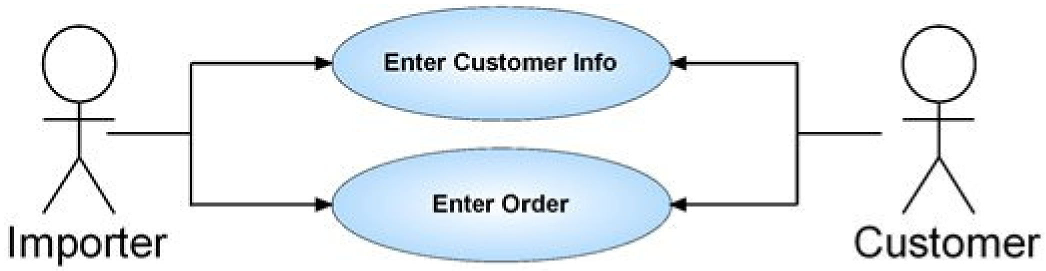


Figure 5 Customer Use Case Diagram

Figure 5 Customer Use Case Diagram contains the use cases primarily initiated by the customer. The importer can perform the same actions as an agent to the customer in the event the customer is not connected.

Table 8 Enter Customer Information Use Case

Name:	Enter customer information

Actors:	Importer, Customer
Description:	This use case captures the data entry required to store a customer. This can be done by the importer or the customer. The majority of customer data is captured when an order is placed, but the one piece maintained as a separate record is the contact information of the customer.
Preconditions:	None
Basic Course of Events:	• The operator begins a new customer entry • The operator enters the contact information • The operator saves the entry.
Postconditions:	The customer information is saved.
Alternatives:	If the customer already exists, an update is performed.
Exceptions:	None
Issues:	None

Table 9 Enter Order Use Case

Name:	Enter Order
Actors:	Customer, Importer
Description:	When a customer places an order to the importer for a specific wine, this information needs to be captured in the system. An order includes information such as price, quantity, size, and the wine desired.
Preconditions:	The customer and chateau desired must exist in the system.
Basic Course of Events:	• The actor begins a new order. • The actor selects the quantity, size, and price to purchase • The actor enters delivery and payment preferences. • The actor saves the order.
Postconditions:	The order is saved in the system updated.
Alternatives:	None
Exceptions:	None

Issues:	What if an update is desired? Are they allowed to modify it? The importer would be the only one with permission to perform that operation.

3.2 Formal Requirement Definition

The use case exercise helped identify the functional requirements of the system and provided an easy means to capture user input as to how the system should work. The next step was to formally write the requirements. As mentioned before, the scope of this paper focuses on the functional requirements of the system.

Formal requirements provide a sort of contract between the software developer and customer. This contract is satisfied when the developer can demonstrate to the customer that the solution satisfies each requirement in the document. As a result, the requirements provide a structured outline of test cases which the developer can use to verify and validate the system.

To create the formal functional requirements, the guidelines defined during the System Engineering Principles class taught by Tom Kohlman and Greg Norby was followed. The class material defined nine attributes of a good requirement:

1. Necessary
2. Implementation free
3. Concise
4. Unambiguous
5. In standard construct
6. Verifiable
7. Complete
8. Consistent with other requirements
9. Feasible.

A common practice in requirement creation is defining them in a tiered format. This allows a set of broad set of requirements to define the system and eliminates too many items being specified within one requirement. When a high level requirement involves more than one satisfying condition, a sub-level is created to allow multiple conditions to be listed as new requirements.

Following the recommendations of Kohlman and Norby and standard requirement conventions, the use cases were formalized into functional requirements. The following high level requirements were first identified, and then the sub requirements were engineered to provide further detail.

R1. The user shall be able to enter chateau information.
R2. The user shall be able to enter negotiante information.
R3. The user shall be able to enter customer information.
R4. The user shall be able to record all offers from a negotiantes
R5. The system shall keep track of inventory purchased from negotiantes
R6. The user shall be able to enter order requests from customers.
R7. The system shall automatically associate orders with inventory purchased.
R8. The user shall be able to create a purchase order per customer per negotiante.
R9. The user shall be able to create an invoice per customer.
R10. The user shall be able to send offers out based on stored information.
R11. The system shall associate satisfied orders with negotiantes.

R1 The user shall be able to enter chateau information
- **R1.1** The chateau information shall allow entry of its Robert Parker score
- **R1.2** The chateau information shall allow entry of its Wine Spectator score
- **R1.3** The chateau information shall allow entry of its Appalachian

R2 The user shall be able to enter negotiante information
- **R2.1** The negotiante information shall allow entry of their contact information
- **R2.2** The negotiante information shall allow entry of their bank information
- **R2.3** The negotiante information shall allow entry of their payment terms
- **R2.4** The negotiante information shall allow entry of invoices received from them

R3 The user shall be able to enter customer information.
- **R3.1** The customer information shall allow entry of contact information

R4 The user shall be able to record all offers from a Negotiantes
- **R4.1** The offer shall include quantity available in crates
- **R4.2** The offer shall include a price per bottle in euros
- **R4.3** The offer shall include the size of bottles being offered

R5 The system shall keep track of inventory purchased from negotiantes
- **R5.1** The user shall be able to record wine purchases from negotiantes
- **R5.2** The importer shall be able to specify the quantity in cases they are purchasing from negotiante
- **R5.3** The importer shall be able to specify the size bottle they are purchasing from negotiante
- **R5.4** The importer shall be able to specify the purchase price per bottle from negotiante

R6 The user shall be able to enter order requests from customers.
- **R6.1** The order request shall include price to customer inventory purchased from per bottle
- **R6.2** The order request shall include number of cases desired

- **R6.3** The order request shall include size of bottles desired
- **R6.4** The order request shall include the wine desired
- **R6.5** The order request shall include payment method (such as financed price)
- **R6.6** The order request shall include payment schedule details

R7 The system shall automatically associate orders with inventory purchased.

- **R7.1** The system shall keep track of orders unsatisfied from inventory
- **R7.2** The system shall keep track of orders satisfied from inventory

R8 The user shall be able to create a purchase order per customer per negotiante

- **R8.1** The purchase order shall contain negotiante information
- **R8.2** The purchase order shall contain the product being purchased
- **R8.3** The purchase order shall contain the quantity of product purchased.
- **R8.4** The purchase order shall contain the price of product being purchased
- **R8.5** The purchase order shall contain the size of product being purchased
- **R8.6** The purchase order shall contain the customer the product is going to

R9 The user shall be able to create an invoice per customer

- **R9.1** The invoice shall contain the products being purchased
- **R9.2** The invoice shall contain the quantity of product purchased.
- **R9.3** The invoice shall contain the price of product purchased.
- **R9.4** The invoice shall contain the size of product purchased.

R10 The user shall be able to send offers out based on stored information

- **R10.1** The offers shall include the quantity of product available
- **R10.2** The offers shall include the price of product available
- **R10.3** The offers shall include the size of product available

R11 The system shall associate satisfied orders with Negotiantes.

- **R11.1** The system shall determine which Negotiantes will deliver a specific Order.

CHAPTER IV
OBJECT ORIENTED ANALYSIS

4.1 Definition

Once requirements are defined for a system, new questions arise regarding how to model the solution. When taking an object-oriented approach, what are the relevant objects and how do they relate to each other? How do they interact in the system? How can these models be specified so that the end result is an effect design? [Pressman 2001].

These questions are answered through object-oriented analysis (OOA). The process of OOA extends from the following basic principles of analysis modeling [Pressman 2001]:

- The information domain is modeled
- Function is described
- Behavior is represented
- Data, functional, and behavioral models are partitioned to expose greater detail
- Early models represent the essence of the problem while later models provide implementation details.

Many methods exist to perform OOA, and not all of them are required to perform a successful analysis. Often, a software project is under schedule and budget constraints and must make a decision early on regarding how much time to spend on OOA. For this paper system sequence diagrams, domain models, and operation contracts will be created to describe the system.

4.2 System Sequence Diagrams

Sequence diagrams “depict the dynamic behavior of elements that make up a system as they interact over time.” [Alhir 2003] Included as part of UML behavior modeling, the diagrams are organized along two axes. The horizontal axis depicts the objects involved and the vertical axis represents time proceeding down the page. Sequence diagrams can be used to show the interactions between any set of objects, such as classes or systems.

System sequence diagrams are a subset of sequence diagrams which show the behavior of a system with outside actors. They provide further analysis of the functional events of the system and help transition from requirement analysis to domain analysis. The events shown in a system sequence diagram are only those between the actor and the system. These actions will be translated into system interfaces. An important note is that system sequence diagrams are an OOA tool and independent of design or implementation of the system. As such, the system is represented as one object even if the final solution is multi-tiered with interfaces between each piece.

For each use case built during requirement analysis, a sequence diagram was created to show the interactions of the actors with the system. These follow very closely with the **Basic Course of Events** section of each use case. When creating the diagrams some common themes, such as querying the system for information were identified and generalized.

4.2.1 Importer System Sequence Diagrams

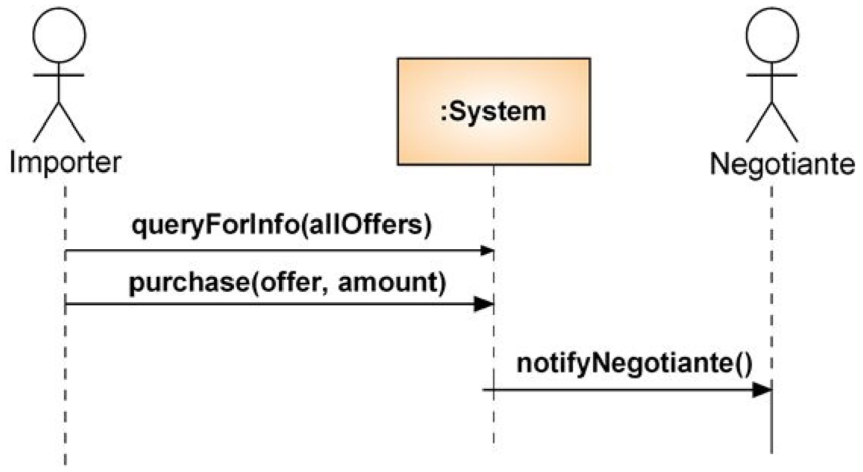


Figure 6 Purchase Inventory System Sequence Diagram

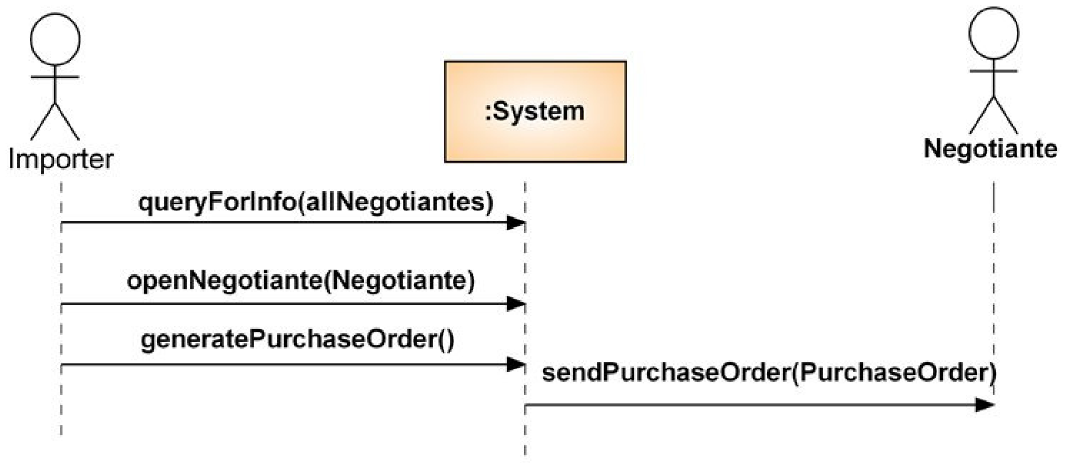

Importer
:System
Negotiante
queryForInfo(allNegotiantes)
openNegotiante(Negotiante)
generatePurchaseOrder()
sendPurchaseOrder(PurchaseOrder)

Figure 7 Generate Purchase Order System Sequence Diagram

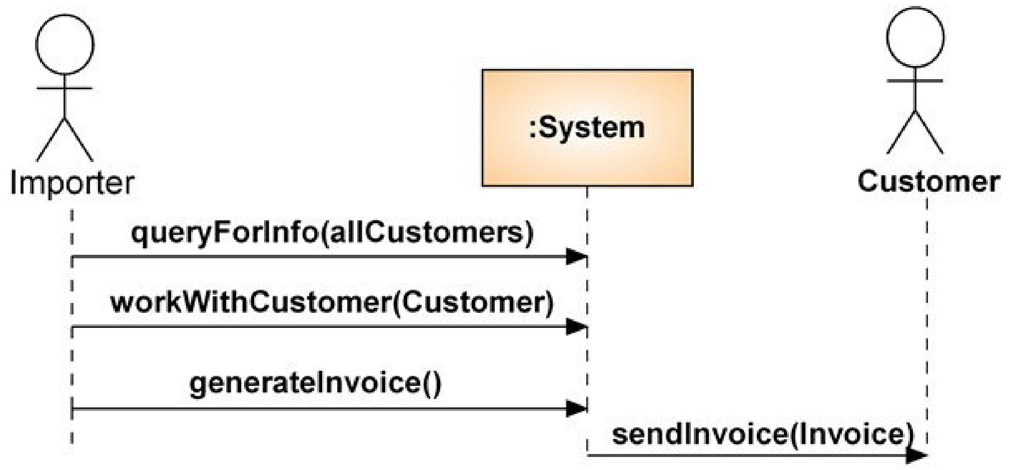


Figure 8 Generate Invoice System Sequence Diagram

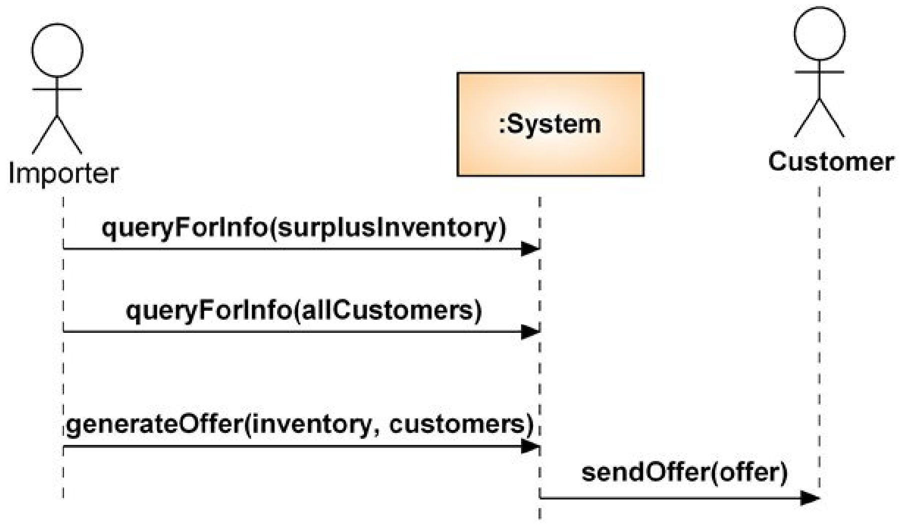

Importer
:System
Customer
queryForInfo(surplusInventory)
queryForInfo(allCustomers)
generateOffer(inventory, customers)
sendOffer(offer)

Figure 9 Generate Surplus Offer System Sequence Diagram

4.2.2 Negotiante System Sequence Diagrams

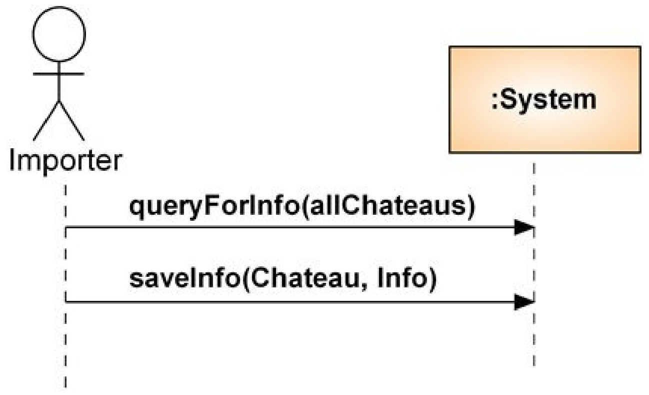


Figure 10 Enter Chateau Info System Sequence Diagram

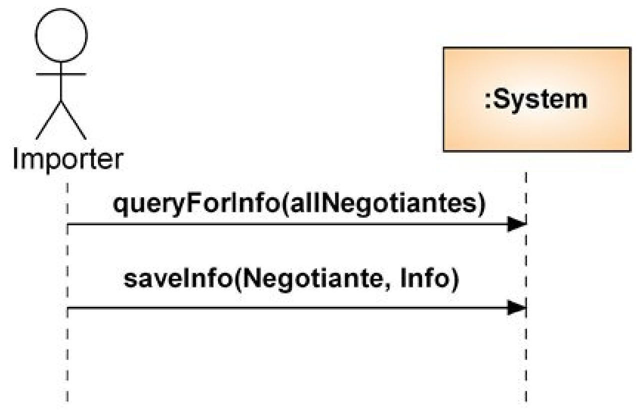


Figure 11 Enter Negotiante Info System Sequence Diagram

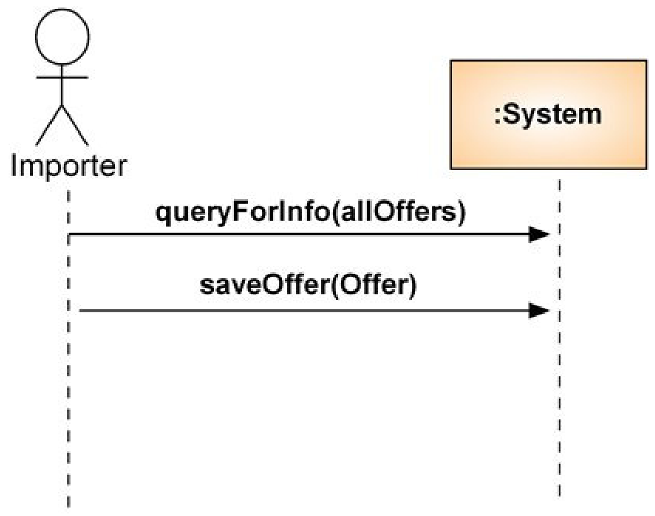


Figure 12 Enter Offer System Sequence Diagram

4.2.3 Customer System Sequence Diagrams

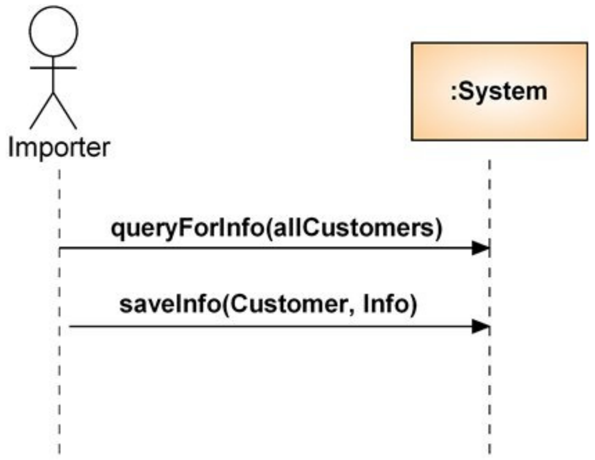


Figure 13 Enter Customer Info Sequence Diagram

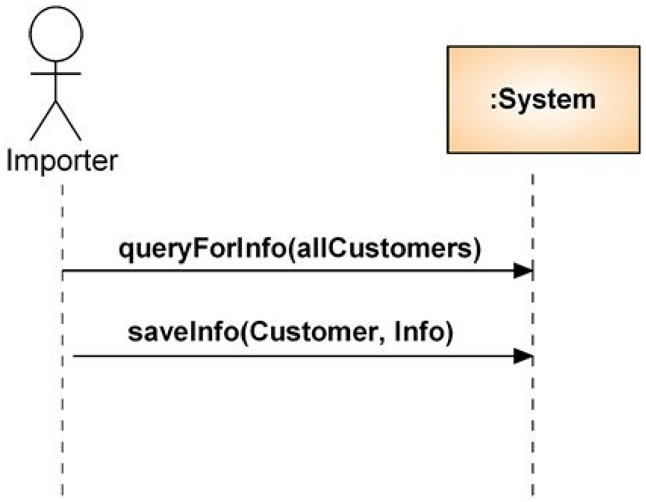


Figure 14 Enter Customer Order Sequence Diagram

4.3 Domain Model

Domain models represent concepts and their logical associations in the real-world problem domain. It is important to note that the domain model is not a picture of software components or classes. They both help to understand a complex domain and provide a set of candidate classes and associations for design. [Valtech 2003]

The domain modeling process begins with generating a list of candidate concepts. Examining the problem description and requirements, the following candidate concepts were identified.

Table 10 Candidate Classes

Chateau	Size	Robert Parker Score
Customer	Price	Wine Spectator Score
Negotiante	Amount	Transaction
Importer	Purchase Order	France
Wine	Invoice	America
Bottle	Futures	Inventory
Order	Surplus	Offer
Contact Information	Bank Information	Payment Terms
Appalachian	Case	Bottle
Payment Schedule	Payment Method	

Having defined this list, the domain model can be created. The model is created by modeling the real world and decomposing the domain into objects. These objects are represented by rectangular boxes with a title at the top and attributes beneath. Next, noteworthy relationships are identified between the objects and represented with a bidirectional line. Multiplicity is shown on each end where it connects to the object. Each line is labeled with a meaningful verb describing the relationship. [Valtech 2003].

In Figure 15, the domain model for this system is shown. The model was drawn as described in the problem statement and requirements. Not all possible relationships are shown, since that would clutter the diagram and render it unusable. An important note is that a domain model is not wrong or right, but more or less helpful during object-oriented design [Valtech 2003].

Many of the candidate classes where used in the final model. Since this is a model of the real world and not a computer system, boxes in the model represent real objects. For example, the Negotiante, Importer, and Customer are actual people which are part of the system. They may later be represented by database tables or classes, but in this model they are present to show their responsibilities in the system. Each one of these individuals has a specification object associated with them which is used to describe their characteristics. These are used in the system because it is common in the real world to record details about an object without keeping the object itself. For example, an order contains details of the customer and not the customer itself. This may seem like a trivial formality, but the better the understanding of the system in the real world; the better the software system can be designed.

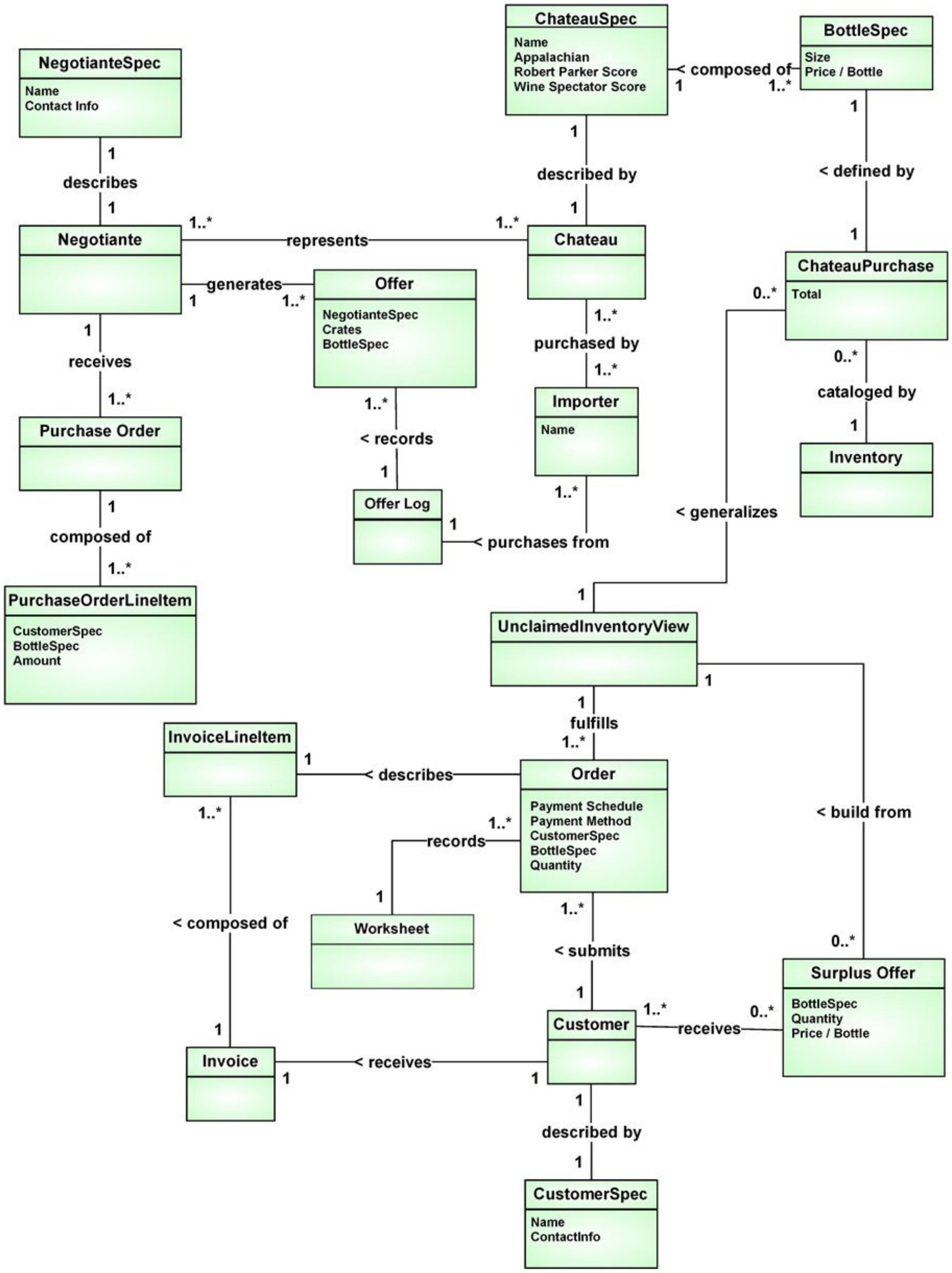


Figure 15 Domain Model

4.4 Operation Contracts

The use of operation contracts as an analysis tool helps describe the changes in the state of the system when system operations are invoked. To create operation contracts, the already created system sequence diagrams and domain model are used to identify the system operations. These operations are used as system operations and the responsibilities of each are noted in a table.

An operation contract is not concerned with how the system performs the operation, but with how the system is modified when the operation is performed. A system operation can be viewed as a black box. The preconditions are items which must be present in the system before the operation can be called, then the operation enters the "black box", finally the postconditions capture the state of the system once the operation is complete. [Valtech 2003]

Following are the operation contracts for the system. Each contract is specified in its own table with each row a different section of the contract.

Table 11 Purchase Operation Contract

Name:	purchase (Offer, amount)
Responsibilities:	Adds the amount of the chateau in the offer to the inventory.
Cross References:	Use Case: Purchase Inventory
Preconditions:	Offer exists in Offer Log
Postconditions:	
(instance creation)	InventoryItem was created if it didn't exist for Offer.NegotianteSpec
(association formed)	InventoryItem added to Inventory
(attribute modification)	InventoryItem.Amount was set to amount

Table 12 QueryForInfo Operation Contract

Name:	queryForInfo(Filter)

Responsibilities:	Queries system for all information matching specified filter and return to user.
Cross References:	All Use Cases
Preconditions:	None
Postconditions:	
(instance creation)	Info object is created for each item in data store matching the specified filter
(association formed)	None
(attribute modification)	Info attributes populated when loaded from the system.

Table 13 SaveInfo Operation Contract

Name:	saveInfo(Info)
Responsibilities:	Saves the given piece of info to the datastore based on the type of information given.
Cross References:	Use Case: Enter Customer Info Use Case: Enter Offer Info Use Case: Enter Negotiante Info Use Case: Enter Offer Info Use Case: Enter Order Info Requirements: R1,R2,R3,R4,R6
Preconditions:	None
Postconditions:	
(instance creation)	Info created in data store
(association formed)	None
(attribute modification)	None

Table 14 WorkWithCustomer Operation Contract

Name:	workWithCustomer(Customer)
Responsibilities:	Sets the given customer as the current customer being worked with.
Cross References:	Use Case: Generate Invoice Requirement: R9
Preconditions:	Customer must exist in system.

Postconditions:	
(instance creation)	
(association formed)	Association formed on Customer as working customer.
(attribute modification)	None

Table 15 Generate Invoice Operation Contract

Name:	generateInvoice()
Responsibilities:	Creates a new invoice for the specified customer
Cross References:	Use Case: Generate Invoice Requirement: R9
Preconditions:	Customer must exist in system. Customer must have a satisfied order in system.
Postconditions:	
(instance creation)	Invoice is created. Line Item created per order for the customer
(association formed)	Orders associated with Line Items
(attribute modification)	None

Table 16 OpenNegotiante Operation Contract

Name:	openNegotiante(Negotiante)
Responsibilities:	System set given Negotiante as working negotiante and opens information for use by user.
Cross References:	Use Case: Generate PurchaseOrders Requirement: R8
Preconditions:	Negotiante must exist in system.
Postconditions:	
(instance creation)	None
(association formed)	Negotiante set as working Negotiante
(attribute modification)	Negotiante info populated if not already.

Table 17 GeneratePurchaseOrders Operation Contract

Name:	generatePurchaseOrders(Negotiante)
Responsibilities:	Creates a new purchase order for each customer associated with a ChateauPurchase purchased from the negotiante

Cross References:	Use Case: Generate Purchase Orders Requirement: R8
Preconditions:	ChateauPurchase must be purchased from the Negotiante. ChateauPurchase must be associated with an order. Order must exist.
Postconditions:	
(instance creation)	A PurchaseOrder was created per customer associated with a Negotiante purchase. PurchaseOrderLineItem created per Chateau purchased by customer.
(association formed)	PurchaseOrder associated with Customer. PurchaseOrderLineItem associated with each different ChateauPurchase associated with Customer and Negotiante
(attribute modification)	PurchaseOrderLineItem fields populated with values from associated objects.

Table 18 GenerateSurplusOffer Operation Contract

Name:	generateSurplusOffer(BottleSpec, Amount, SalePrice, Customer[])
Responsibilities:	Creates a new SurplusOffer
Cross References:	Use Case: Generate Surplus Offer Requirement: R10
Preconditions:	Amount of BottleSpec must exist in UnclaimedInventoryView. Customers must exist
Postconditions:	
(instance creation)	SurplusOffer created.
(association formed)	SurplusOffer associated with each Customer
(attribute modification)	SurplusOffer attributes were set to amount and SalePrice

CHAPTER V
OBJECT ORIENTED DESIGN

5.1 Definition

There is a thin line between object-oriented analysis (OOA) and object-oriented design (OOD). In fact, when performed correctly, latter stages of object-oriented analysis can seamlessly transition into the early stages of OOD. Object-oriented design takes the analysis model and transforms it into a blueprint which can be used to build the system. OOD can be divided into two separate parts, system design and object design.

System Design defines a series of layers which when integrated together perform system operations. In addition to these layers, system design also focuses on user interface design, data management functions, and task management facilities [Pressman 2001]. Object Design focuses on the details of each class such as their attributes, operations, and messages. Figure 5.1 shows how the artifacts collected during object-oriented analysis can be used during OOD.

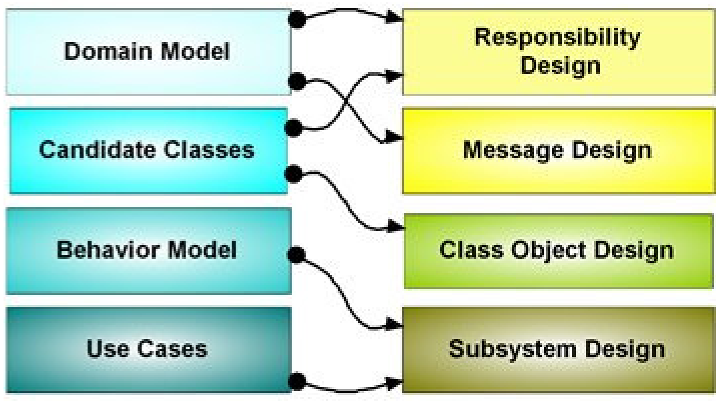


Figure 16 OOA to OOD Transition model [Pressman 2001]

5.2 System Design

5.2.1 Subsystem Design

A fundamental task early on in OOD is partitioning the system into subsystems based on common functionality or physical location. A multi-tiered architecture is common among applications because it decouples sections of a system from each other. Defined interfaces are the only link between the two, which leaves the implementation compartmentalized within the subsystem. For the system being designed in this paper, a three tier system will work well. These tiers are a presentation layer, business logic layer, and data management layer.

Each layer has its own design and implementation methodologies, and often organizations divide their teams and activities based on the layers defined in the system. Before specific design of each layer begins, an understanding of how the layers interact is needed. During OOA, system sequence diagrams were created from use cases to model how the customer would interact with the system. The interfaces identified were captured in operation contracts which detailed what happened inside the system when a new command was received.

Sequence diagrams can again be used during design to model how the three layers interact over time. Each system sequence diagram can be expanded to show how the interfaces. To ensure the interfaces into each layer are logical, each layer should be defined to be either stateful or stateless. Stateful means that the layer maintains a session with the client and persist state information and responds to events based on the state of the session. Stateless means that a session is not kept and each request from a client is handled solely based on the inputs provided through the interface.

The presentation layer is responsible for interfacing with the end user. When input is received, it must be validated and entered into the system through the business logic layer. The presentation layer is almost always stateful. In order for a user interface to be responsive and user friendly, it is necessary to remember what has already occurred. Otherwise, the user would have to reenter the same information on every interaction.

The business logic layer is the middle piece of the architecture and is the only component with knowledge of how to communicate with the data management layer. This layer can be implemented as either stateful or stateless. There are appropriate times to use either approach, but in general the business logic layer should be designed to be stateless unless a specific usage scenario requires a stateful transaction. [Perrone 2003]

The data management layer is responsible for information persistence and provides query capability to quickly retrieve it back. Data management systems typically deal in sessions, commonly referred to as transactions. Within this transaction, the client can query or manipulate the existing data. When a transaction is complete, changes are either rolled back or committed.

The following expanded system sequence diagrams show how each tier will be involved in the system operations identified during analysis. The presentation layer is assumed stateful, the business logic layer stateless, and the data management layer stateful within the context of the operation.

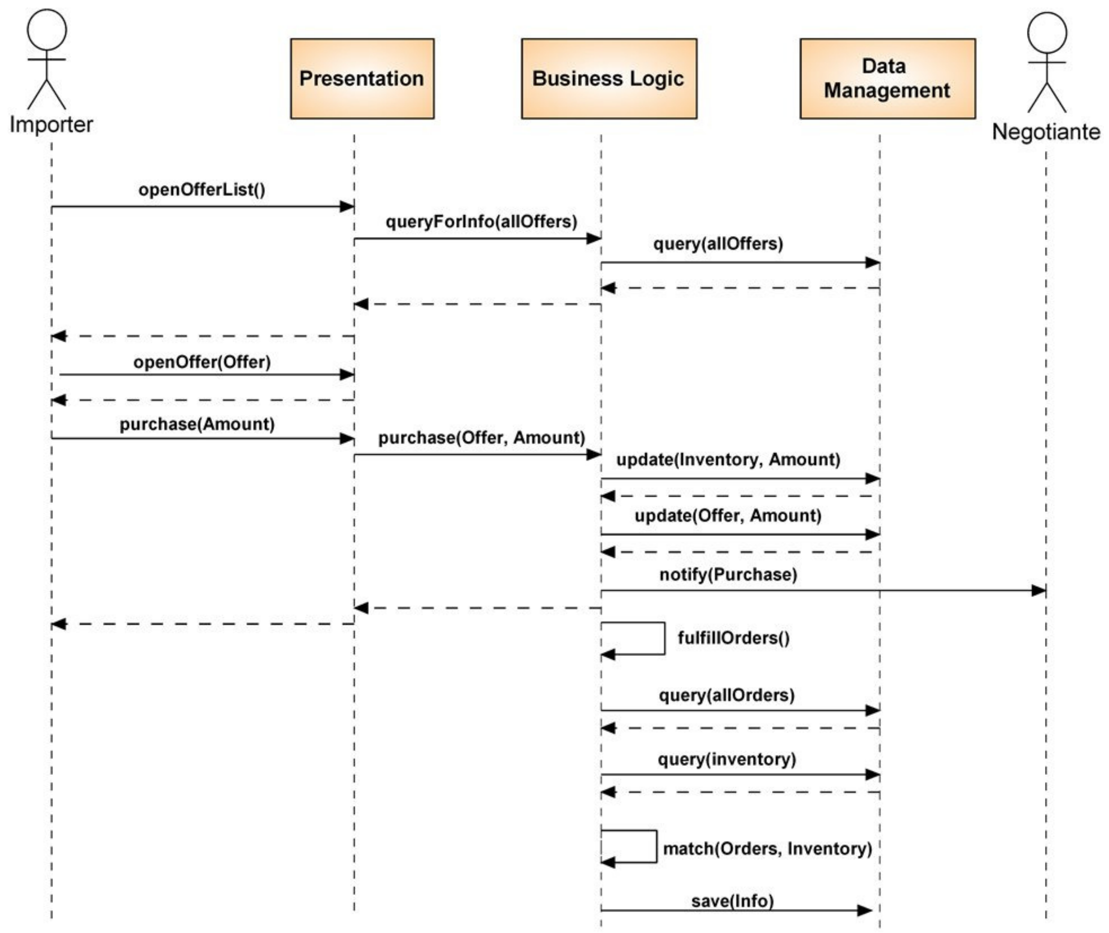


Figure 17 Expanded Purchase Inventory Sequence Diagram

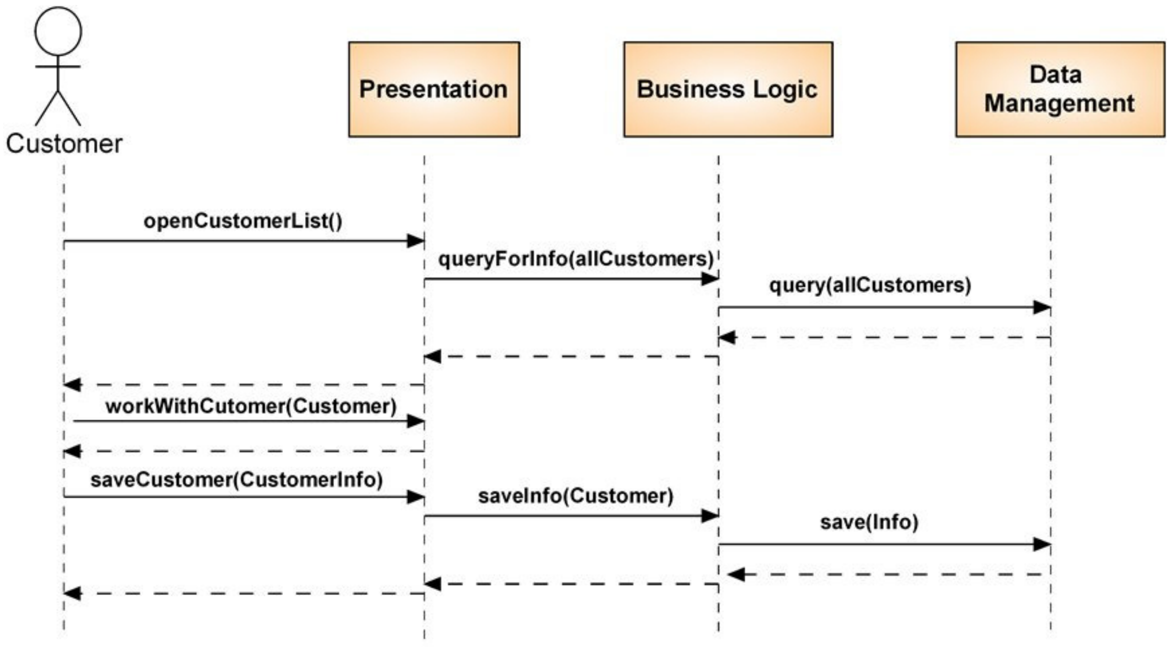


Figure 18 Expanded Enter Customer Sequence Diagram

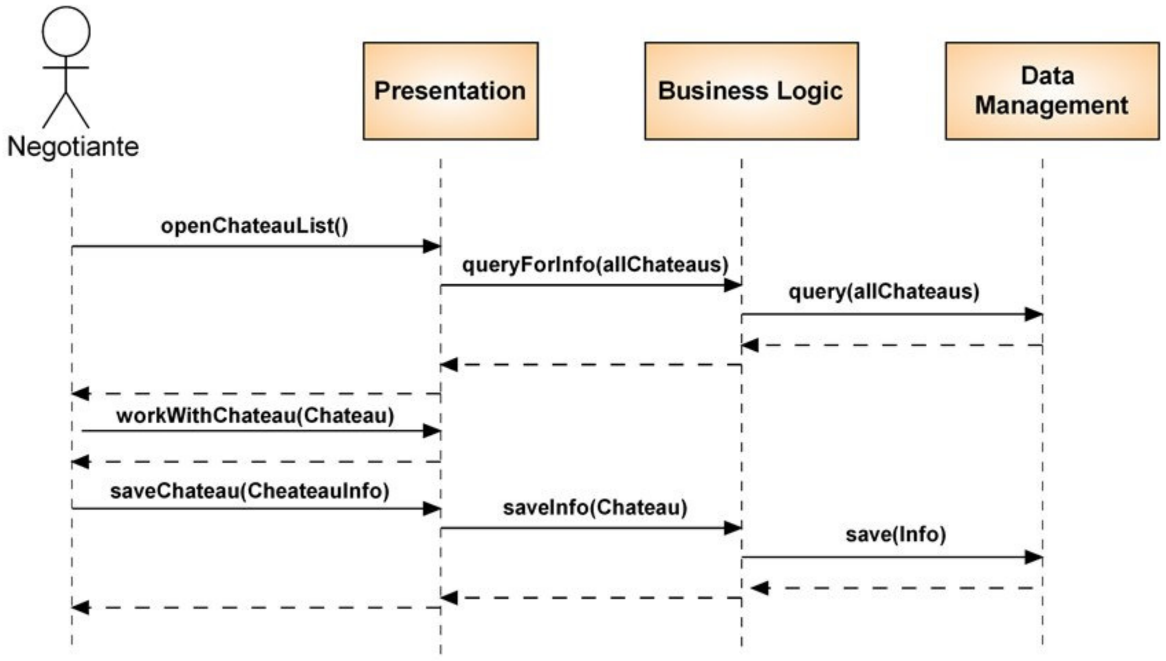


Figure 19 Expanded Enter Chateau Sequence Diagram

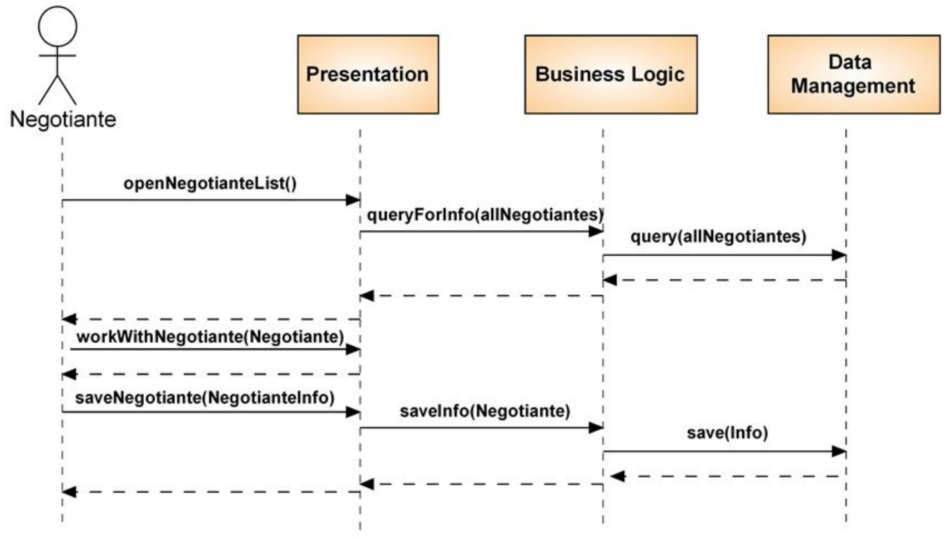


Figure 20 Expanded Enter Negotiante Sequence Diagram

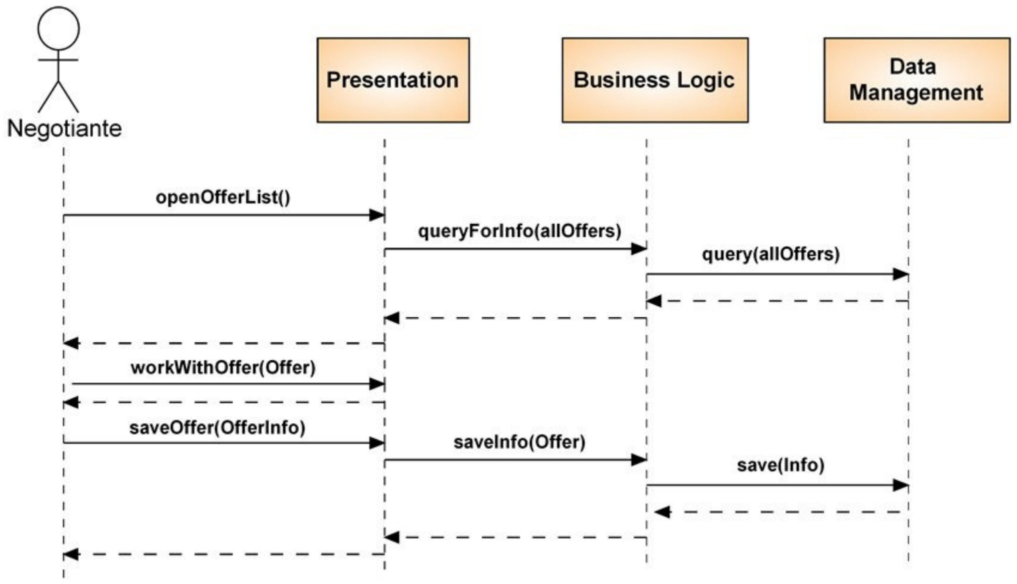


Figure 21 Expanded Enter Offer Sequence Diagram

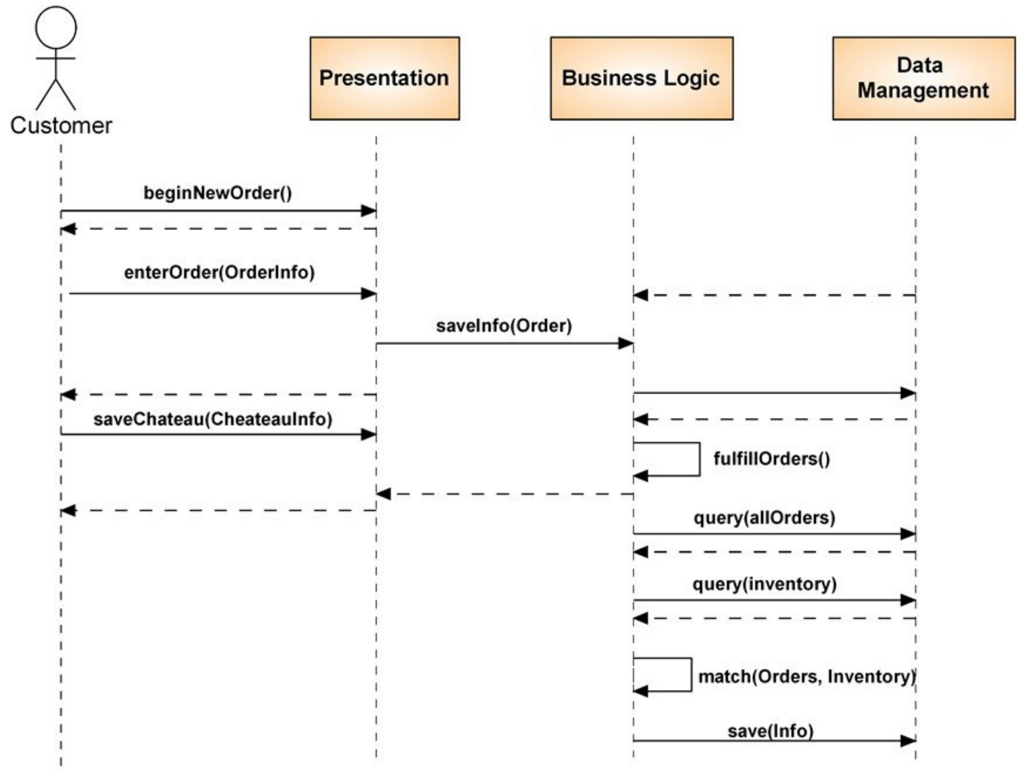


Figure 22 Expanded Enter Order Sequence Diagram

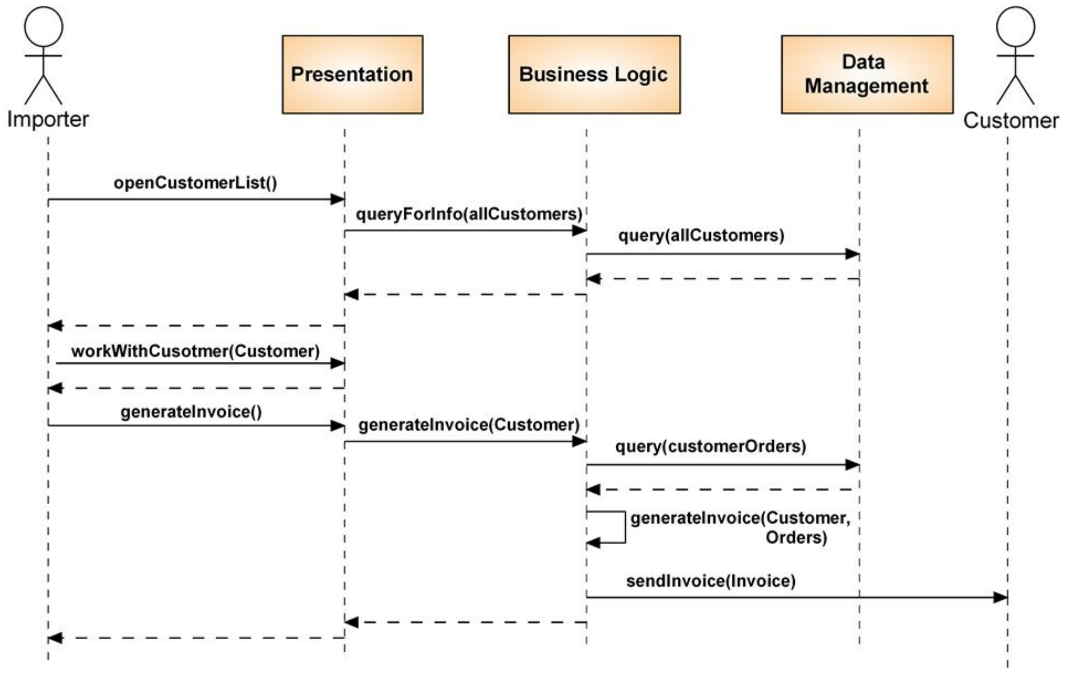


Figure 23 Expanded Generate Invoice Sequence Diagram

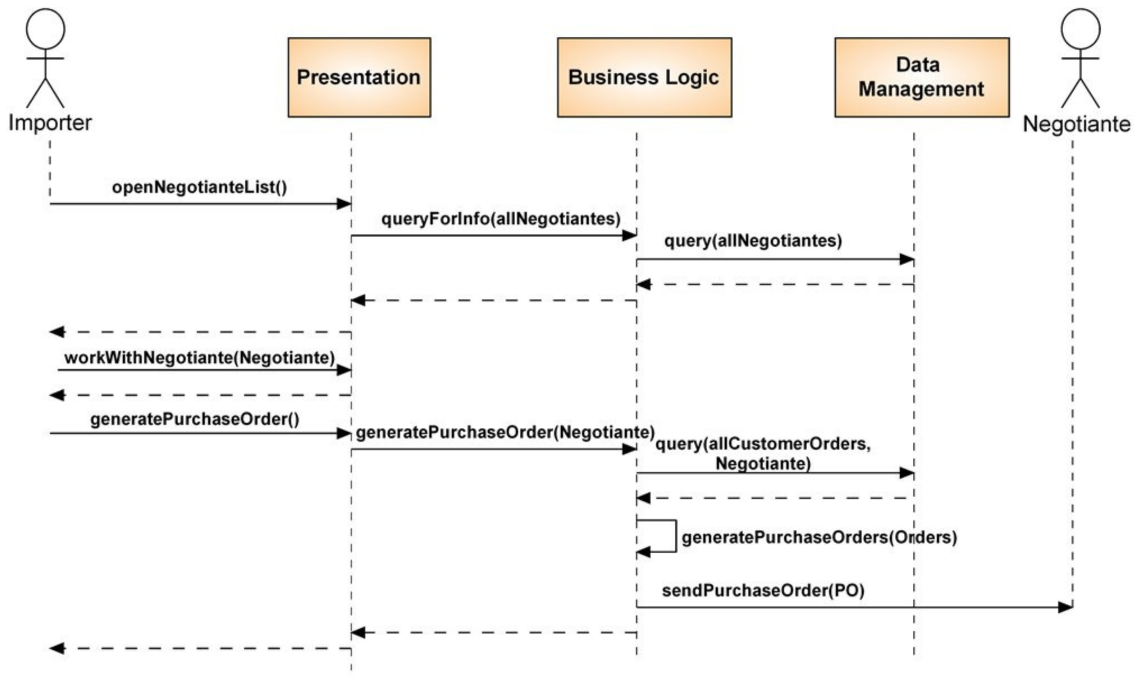

Importer
Presentation
Business Logic
Data Management
Negotiante
openNegotianteList()
queryForInfo(allNegotiantes)
query(allNegotiantes)
workWithNegotiante(Negotiante)
generatePurchaseOrder()
generatePurchaseOrder(Negotiante)
query(allCustomerOrders, Negotiante)
generatePurchaseOrders(Orders)
sendPurchaseOrder(PO)

Figure 24 Expanded Generate Purchase Order Sequence Diagram

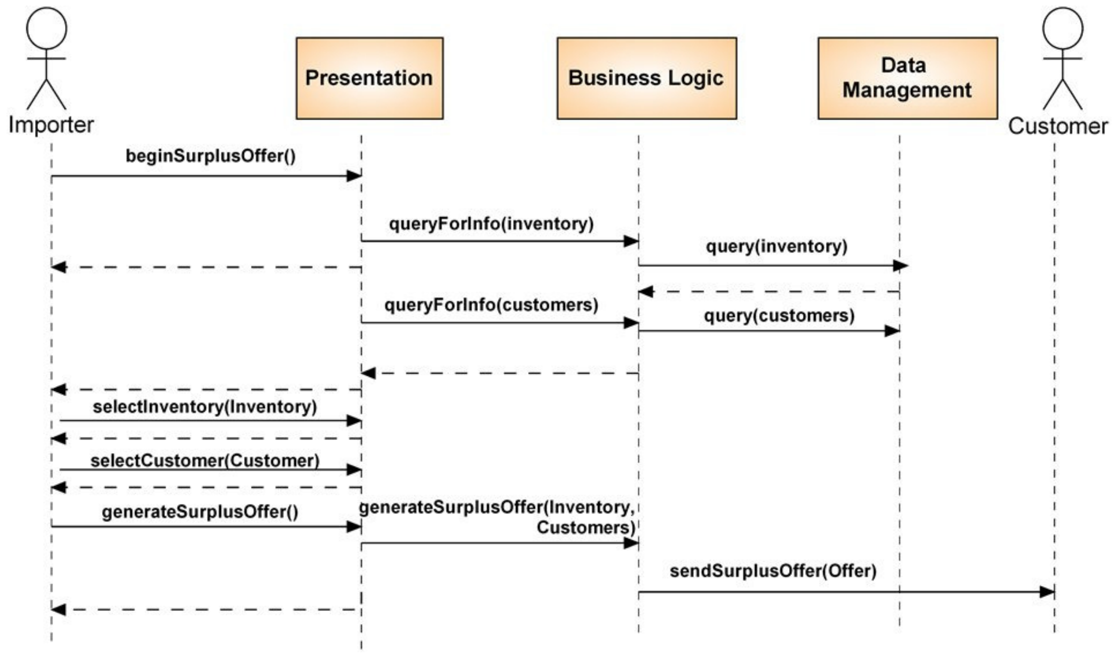


Figure 25 Expanded Generate Surplus Offer Sequence Diagram

5.2.1.1 Presentation Layer

Specific class and interaction details in a presentation view are highly dependant on the implementation decisions made after high level design is complete. One item in the presentation layer which can be designed independent of implementation is the user interface command hierarchy. This "defines major subsystem menu categories and all subfunctions that are available within the context of a major subsystem menu category" [Pressman 2001]. The command hierarchy is refined iteratively until every use-case can be implemented by navigating the hierarchy of functions.

The command hierarchy is created by revisiting the use cases and sequence diagrams again to understand the operations and flow required. For this system, a command hierarchy was created which allows for each use case to be completed in a minimum number of steps while still being simplistic enough to be user friendly. The shortest path would be to have every command accessible from the root, but this would result in a cluttered and unorganized interface. A more intuitive flow is accomplished by organizing the commands into major subsystems.

For this system, four major groups where identified: Information, Offers, Inventory, and Orders. The command tree was broken up based on root nodes for presentation purposes. Each use case identified for the system can be accomplished by traversing the nodes.

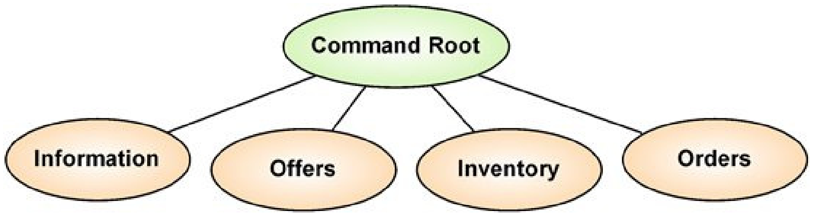


Figure 26 Command Root Heirarchy

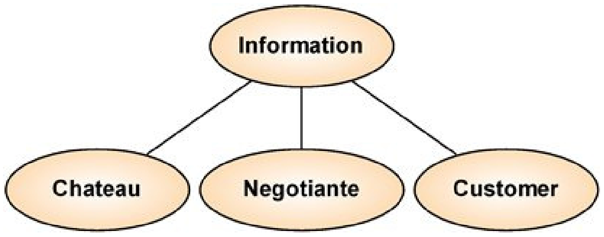


Figure 27 Information Command Hierarchy

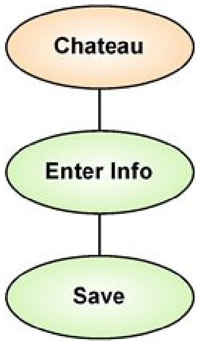


Figure 28 Chateau Command Hierarchy

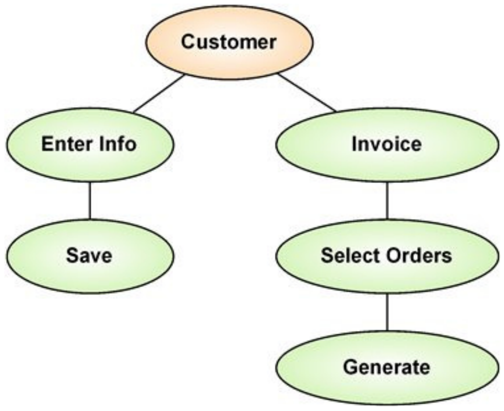


Figure 29 Customer Command Hierarchy

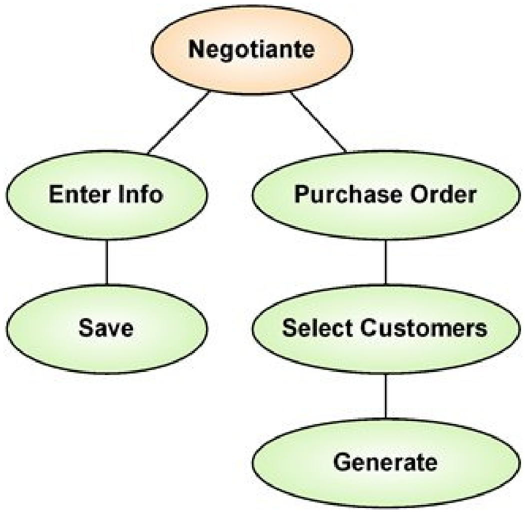


Figure 30 Negotiante Command Hierarchy

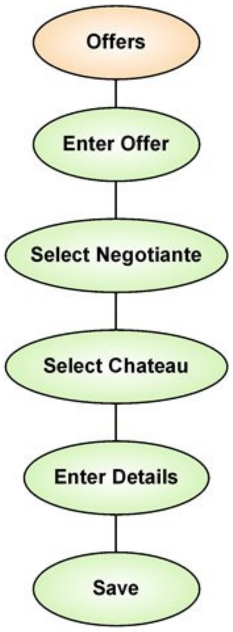


Figure 31 Offer Command Heirarchy

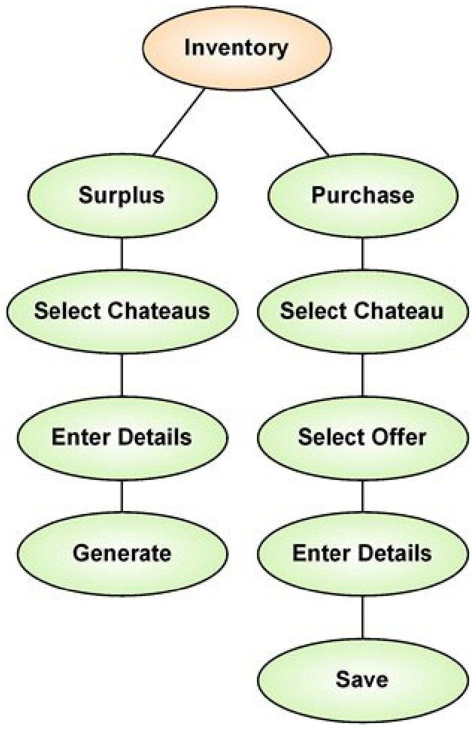


Figure 32 Inventory Command Hierarchy

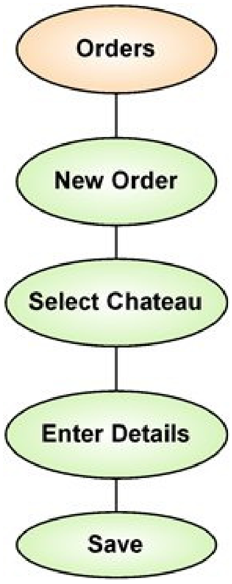


Figure 33 Order Command Hierarchy

5.2.1.2 Business Logic Layer

As described earlier, the business logic layer performs operations in response to the presentation layer's requests. Working with the expanded system sequence diagrams, an interface is defined for the business logic layer so that any external components (such as the presentation layer) can access the information contained within the system. This abstraction provides two important functions. First, if the implementation of the data management layer changes only the business logic layer has to be updated. Second, if additional components are added to the system and require access to the data store, they can easily "plug in" to the business logic interface. Decisions regarding how the presentation layer and business logic layer will physically interact should be delayed until implementation, but with an understanding of the messages the layer will receive and send, the internal design can be accomplished.

Design of the business logic layer can be done through sequence and class diagrams. Previously, sequence diagrams have focused on the system as a whole, but for this level of design, the diagrams will show the interactions between the classes themselves. Class diagrams show details of the classes themselves and how they relate to others. Class diagrams are an excellent way to visualize object-oriented relationships and principles the classes will be built with. These concepts include, but are not limited to, encapsulation, inheritance, and polymorphism. While interesting and vitally important, a discussion of these is beyond the scope of this paper.

Design of object-oriented systems has been done countless times, and often similar problems are faced by designers which can be solved by the same ideal solution. Designers realized this trend and captured these common solutions as "patterns." Patterns exist for all types of programming situation and a designer need only be aware that the solution exists to implement it. For this system, a handful of patterns have been leveraged during design of the business logic layer. The two primary patterns used are the Decorator and Visitor patterns. [Gamma 1995] Both of these patterns allow operations to be performed with a class without having to alter the internals of a class. Again, a detailed discussion of these is beyond this paper, but using these does allow for simple classes to represent information without having to know how the rest of the system will use them.

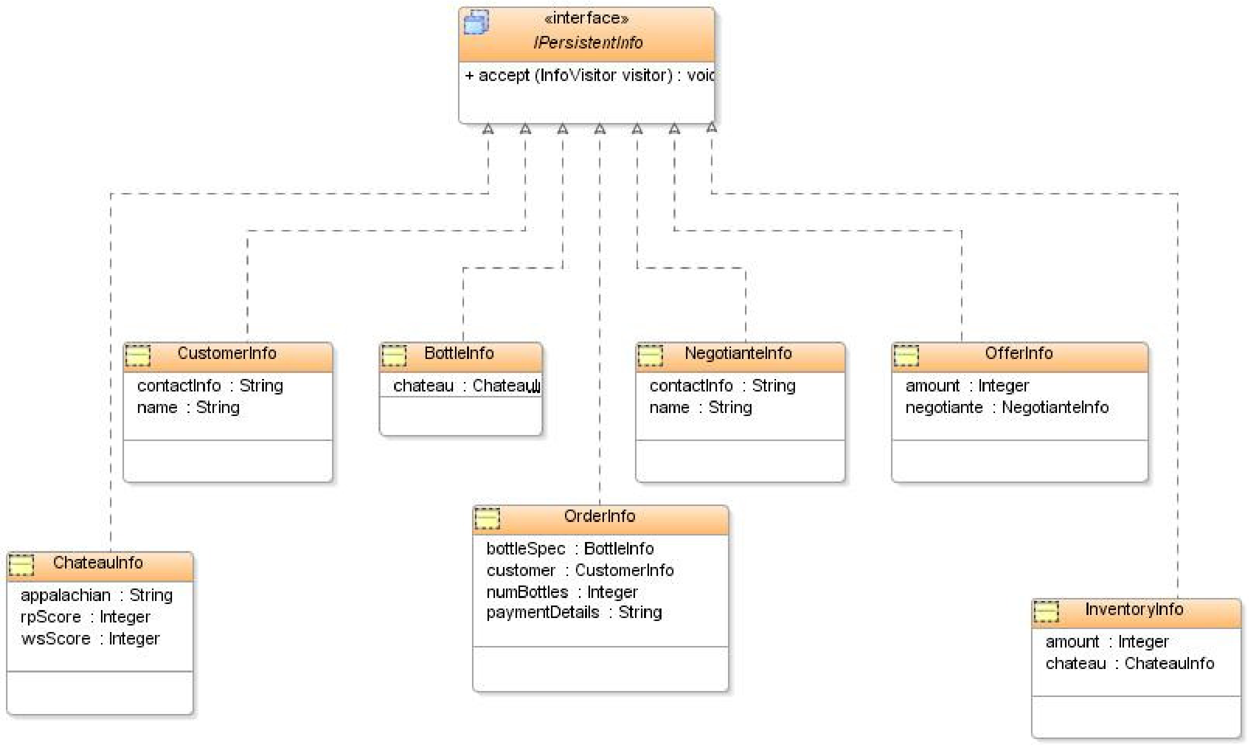


Figure 34 Persistent Info Class Diagram

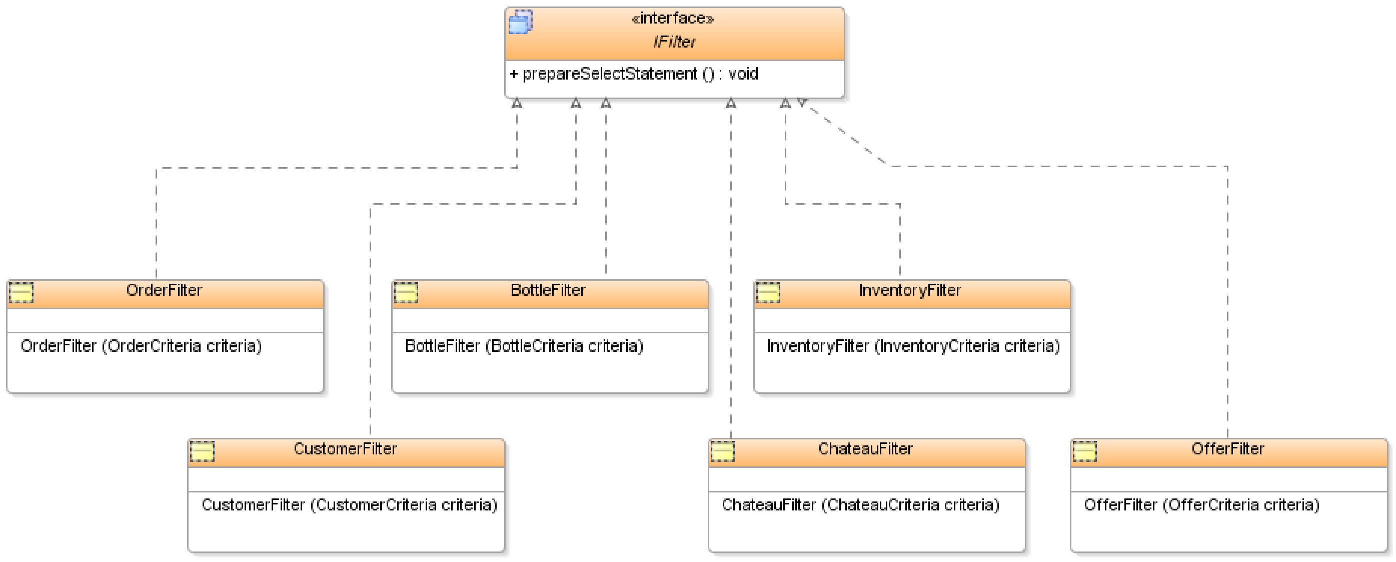

«interface»
IFilter
+ prepareSelectStatement () : void
OrderFilter
OrderFilter (OrderCriteria criteria)
BottleFilter
BottleFilter (BottleCriteria criteria)
InventoryFilter
InventoryFilter (InventoryCriteria criteria)
CustomerFilter
CustomerFilter (CustomerCriteria criteria)
ChateauFilter
ChateauFilter (ChateauCriteria criteria)
OfferFilter
OfferFilter (OfferCriteria criteria)

Figure 35 Filter Class Diagram

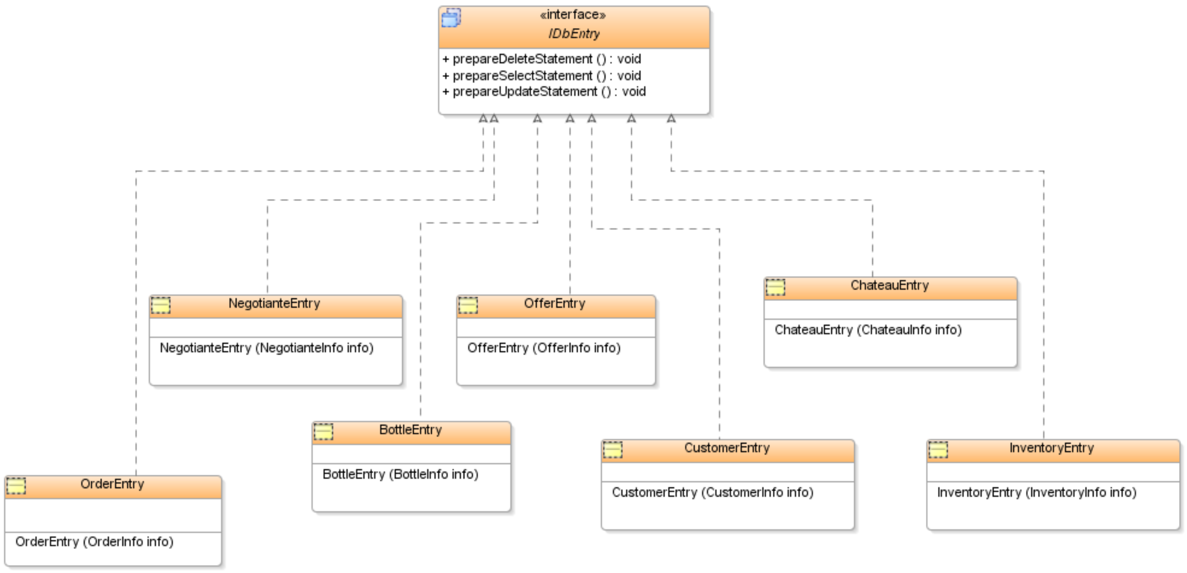

«interface»
IDbEntry
+ prepareDeleteStatement () : void
+ prepareSelectStatement () : void
+ prepareUpdateStatement () : void
NegotianteEntry
NegotianteEntry (NegotianteInfo info)
OfferEntry
OfferEntry (OfferInfo info)
ChateauEntry
ChateauEntry (ChateauInfo info)
BottleEntry
BottleEntry (BottleInfo info)
CustomerEntry
CustomerEntry (CustomerInfo info)
InventoryEntry
InventoryEntry (InventoryInfo info)
OrderEntry
OrderEntry (OrderInfo info)

Figure 36 Database Entry Class Diagram

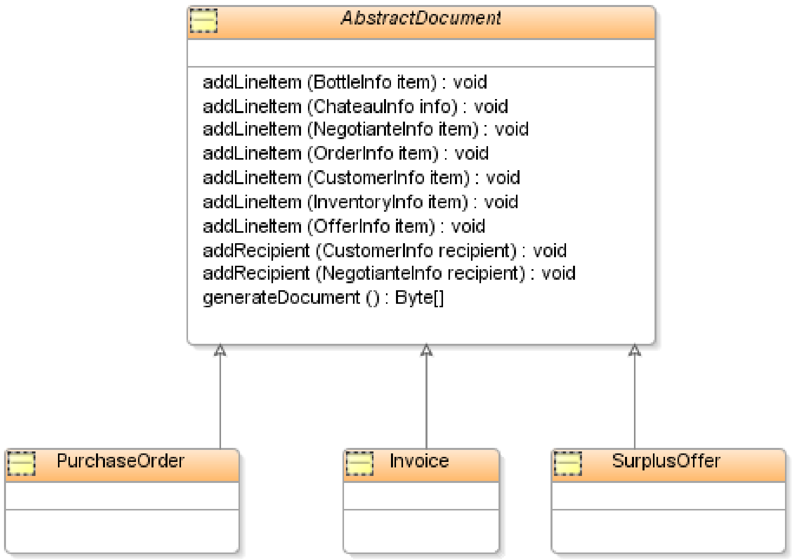

AbstractDocument
addLineItem (BottleInfo item) : void
addLineItem (ChateauInfo info) : void
addLineItem (NegotianteInfo item) : void
addLineItem (OrderInfo item) : void
addLineItem (CustomerInfo item) : void
addLineItem (InventoryInfo item) : void
addLineItem (OfferInfo item) : void
addRecipient (CustomerInfo recipient) : void
addRecipient (NegotianteInfo recipient) : void
generateDocument () : Byte[]
PurchaseOrder
Invoice
SurplusOffer

Figure 37 Document Class Diagram

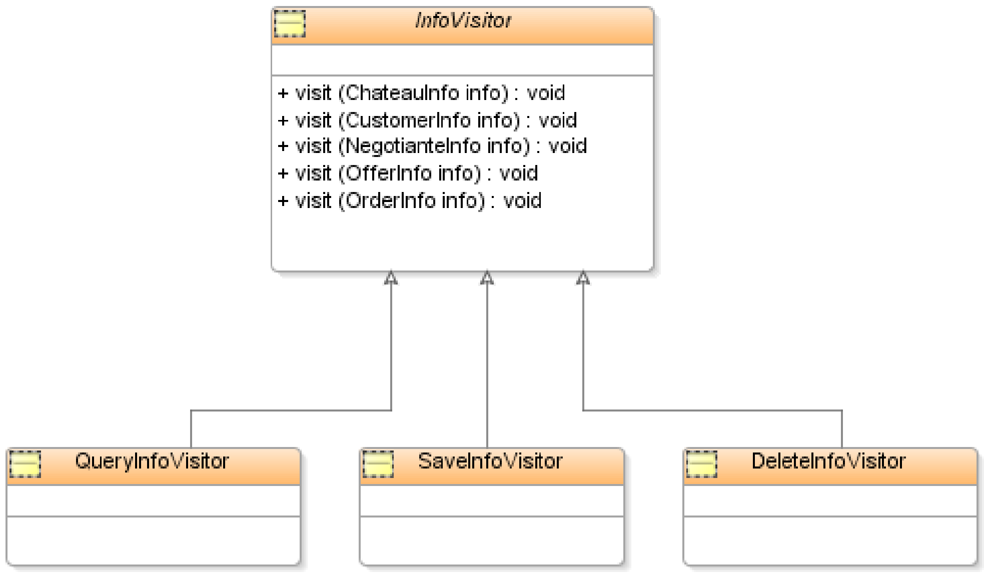

InfoVisitor
+ visit (ChateauInfo info) : void
+ visit (CustomerInfo info) : void
+ visit (NegotianteInfo info) : void
+ visit (OfferInfo info) : void
+ visit (OrderInfo info) : void
QueryInfoVisitor
SaveInfoVisitor
DeleteInfoVisitor

Figure 38 Info Visitor Class Diagram

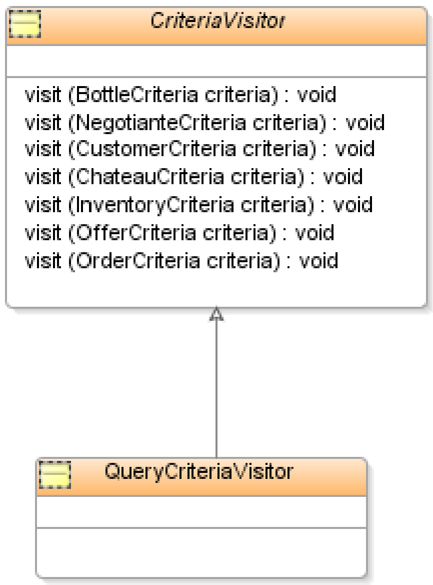

CriteriaVisitor
visit (BottleCriteria criteria) : void
visit (NegotianteCriteria criteria) : void
visit (CustomerCriteria criteria) : void
visit (ChateauCriteria criteria) : void
visit (InventoryCriteria criteria) : void
visit (OfferCriteria criteria) : void
visit (OrderCriteria criteria) : void
QueryCriteriaVisitor

Figure 39 Criteria Visitor Class Diagram

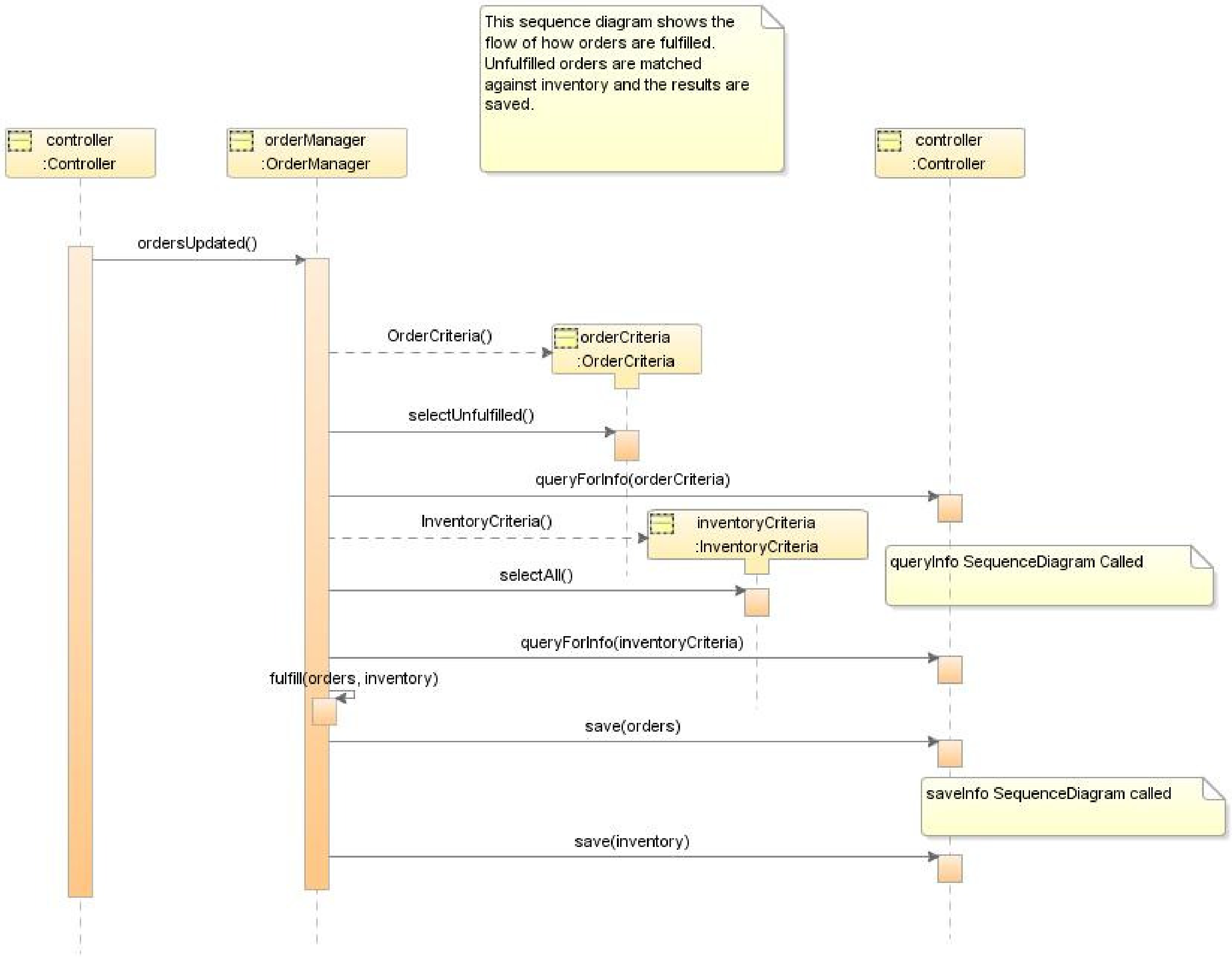
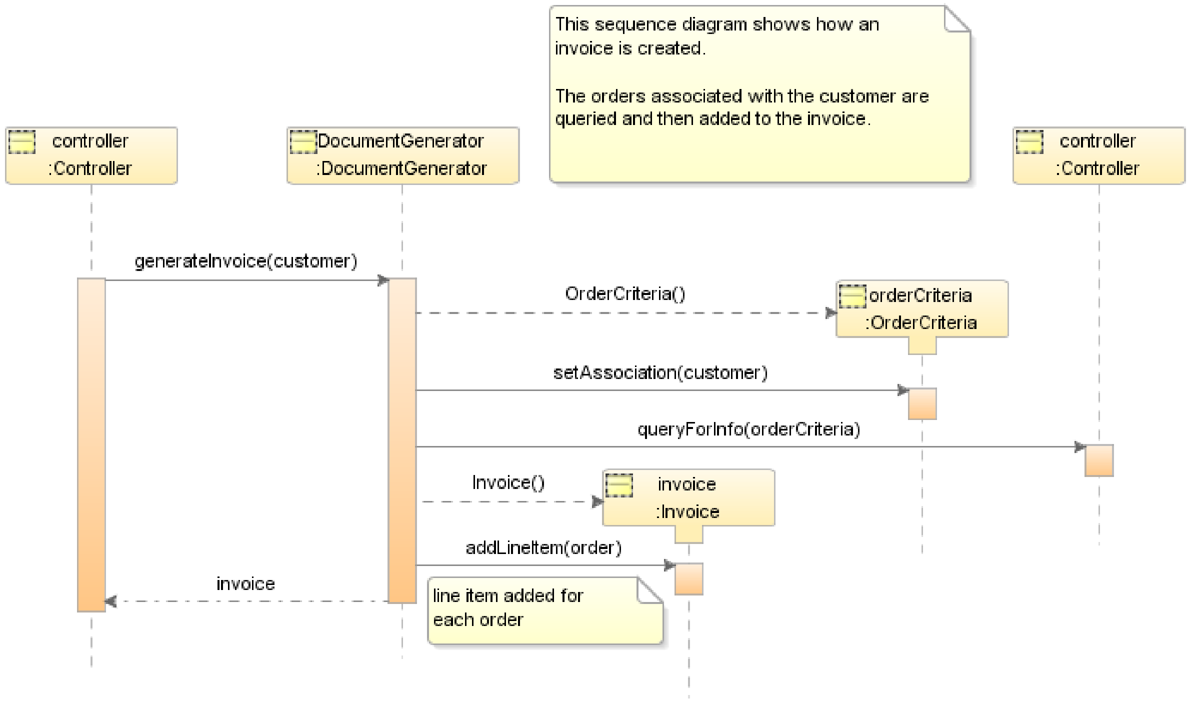

This sequence diagram shows the flow of how orders are fulfilled. Unfulfilled orders are matched against inventory and the results are saved.
controller :Controller
orderManager :OrderManager
controller :Controller
ordersUpdated()
OrderCriteria()
orderCriteria :OrderCriteria
selectUnfulfilled()
queryForInfo(orderCriteria)
InventoryCriteria()
inventoryCriteria :InventoryCriteria
queryInfo SequenceDiagram Called
selectAll()
queryForInfo(inventoryCriteria)
fulfill(orders, inventory)
save(orders)
saveInfo SequenceDiagram called
save(inventory)

Figure 40 Fulfill Orders Business Logic Sequence Diagram


This sequence diagram shows how an invoice is created.

The orders associated with the customer are queried and then added to the invoice.
controller :Controller
DocumentGenerator :DocumentGenerator
controller :Controller
generateInvoice(customer)
OrderCriteria()
orderCriteria :OrderCriteria
setAssociation(customer)
queryForInfo(orderCriteria)
Invoice()
invoice :Invoice
addLineItem(order)
invoice
line item added for each order

Figure 41 Generate Invoice Business Logic Sequence Diagram

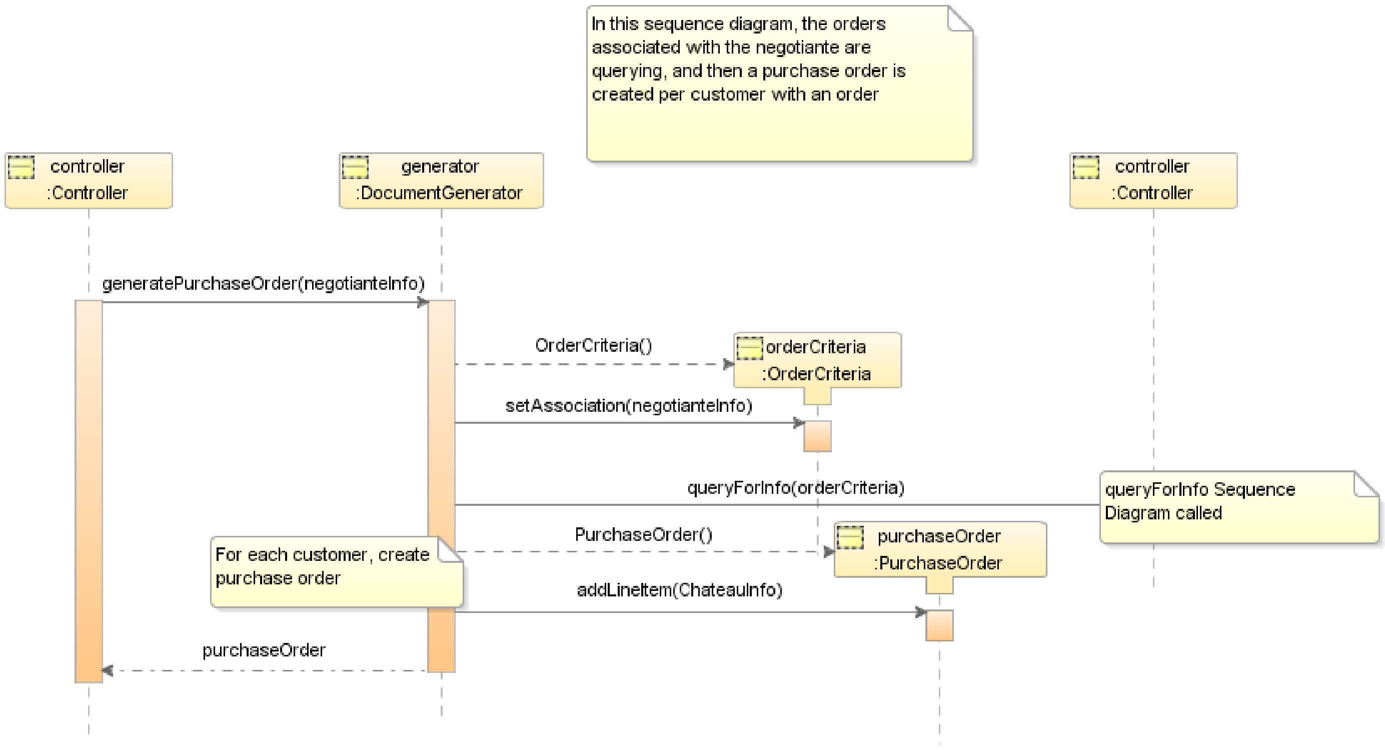
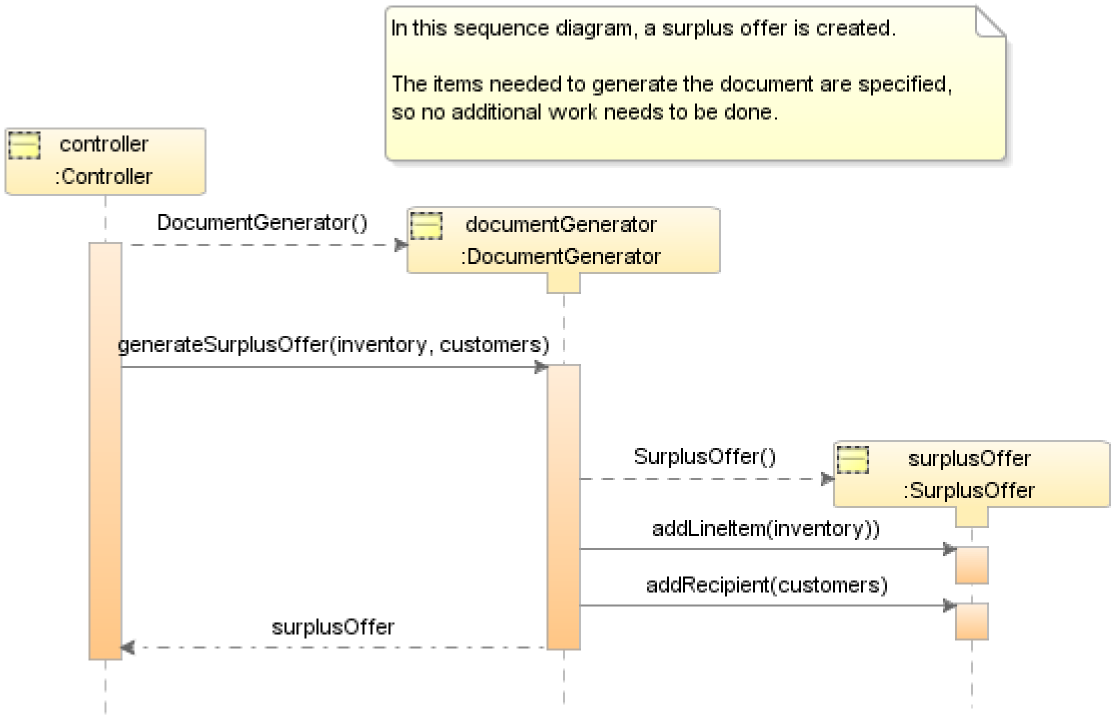

In this sequence diagram, the orders associated with the negotiante are querying, and then a purchase order is created per customer with an order
controller :Controller
generator :DocumentGenerator
controller :Controller
generatePurchaseOrder(negotianteInfo)
OrderCriteria()
orderCriteria :OrderCriteria
setAssociation(negotianteInfo)
queryForInfo(orderCriteria)
queryForInfo Sequence Diagram called
PurchaseOrder()
purchaseOrder :PurchaseOrder
For each customer, create purchase order
addLineItem(ChateauInfo)
purchaseOrder

Figure 42 Generate Purchase Order Business Logic Sequence Diagram


In this sequence diagram, a surplus offer is created.

The items needed to generate the document are specified, so no additional work needs to be done.
controller :Controller
DocumentGenerator()
documentGenerator :DocumentGenerator
generateSurplusOffer(inventory, customers)
SurplusOffer()
surplusOffer :SurplusOffer
addLineItem(inventory))
addRecipient(customers)
surplusOffer

Figure 43 Generate Surplus Offer Business Logic Sequence Diagram

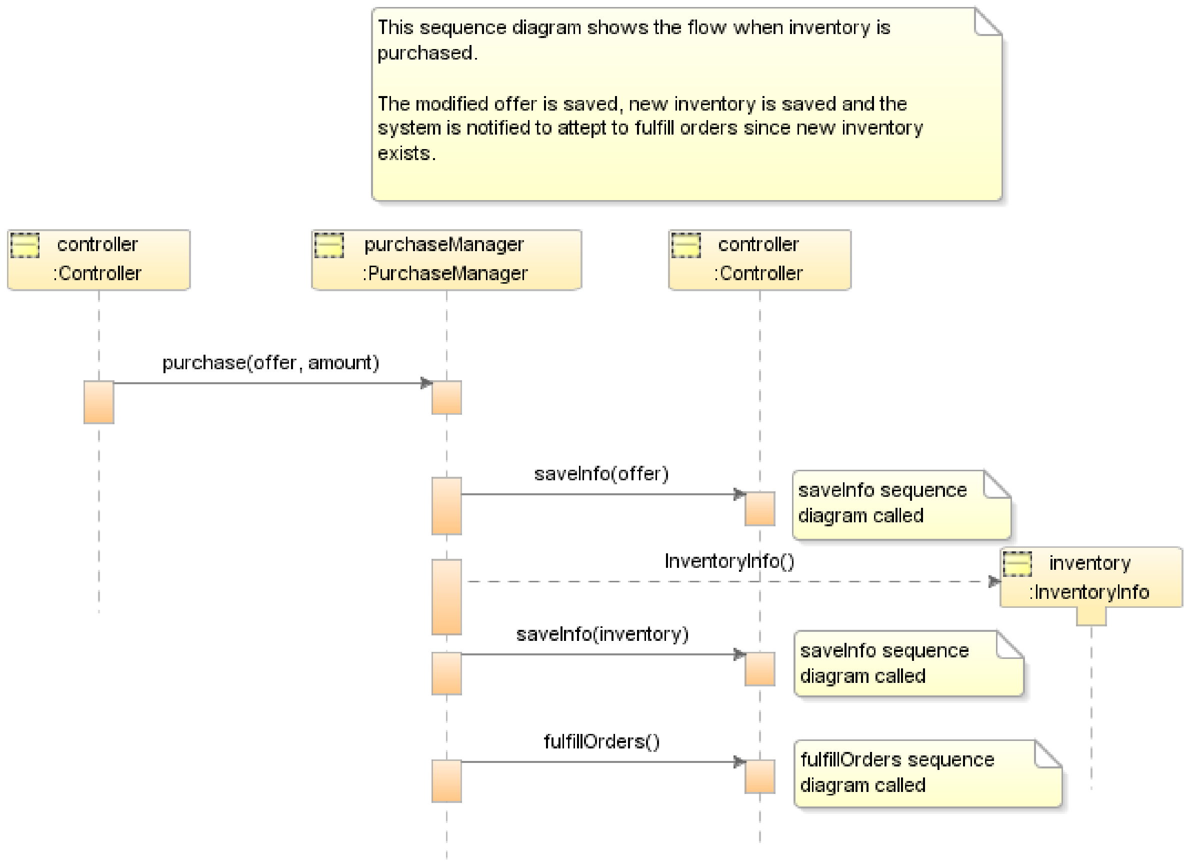

This sequence diagram shows the flow when inventory is purchased.
The modified offer is saved, new inventory is saved and the system is notified to attept to fulfill orders since new inventory exists.
controller :Controller
purchaseManager :PurchaseManager
controller :Controller
purchase(offer, amount)
saveInfo(offer)
saveInfo sequence diagram called
InventoryInfo()
inventory :InventoryInfo
saveInfo(inventory)
saveInfo sequence diagram called
fulfillOrders()
fulfillOrders sequence diagram called

Figure 44 Purchase Inventory Business Logic Sequence Diagram

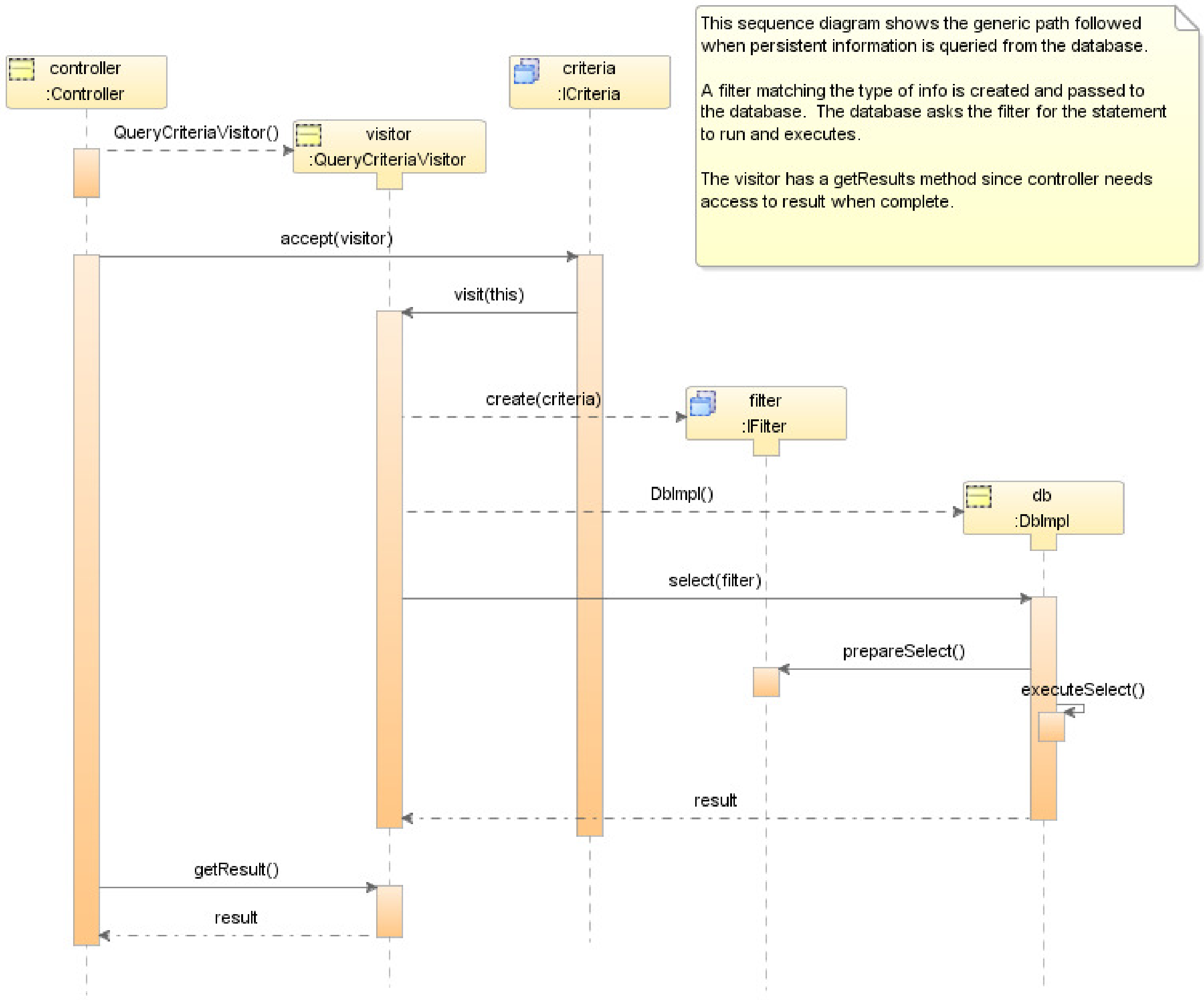

This sequence diagram shows the generic path followed when persistent information is queried from the database.
A filter matching the type of info is created and passed to the database. The database asks the filter for the statement to run and executes.
The visitor has a getResults method since controller needs access to result when complete.
controller :Controller
criteria :ICriteria
QueryCriteriaVisitor()
visitor :QueryCriteriaVisitor
accept(visitor)
visit(this)
create(criteria)
filter :IFilter
DbImpl()
db :DbImpl
select(filter)
prepareSelect()
executeSelect()
result
getResult()
result

Figure 45 Query For Info Business Logic Sequence Diagram

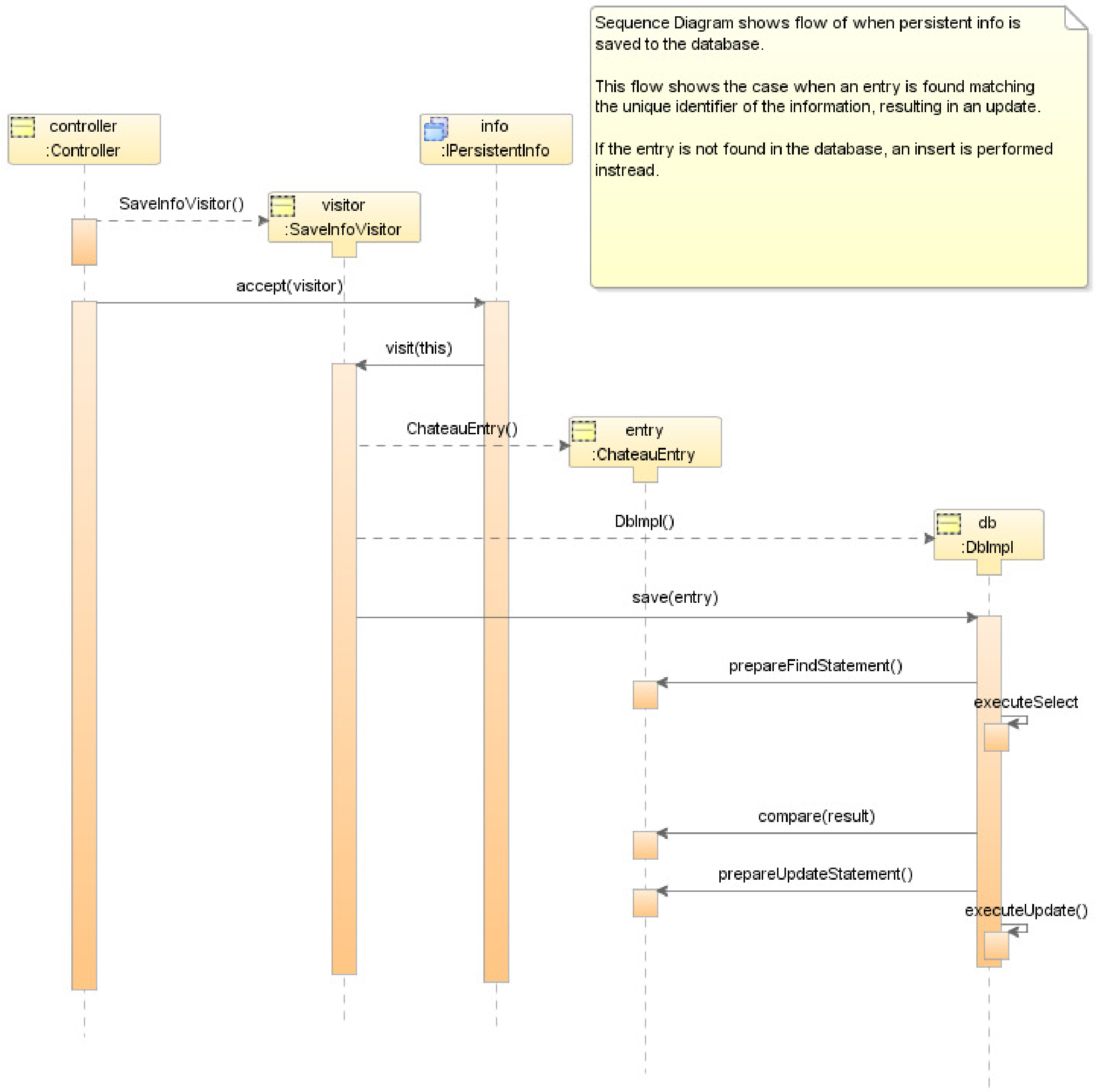

controller
:Controller
info
:IPersistentInfo
Sequence Diagram shows flow of when persistent info is saved to the database.
This flow shows the case when an entry is found matching the unique identifier of the information, resulting in an update.
If the entry is not found in the database, an insert is performed instread.
SaveInfoVisitor()
visitor
:SaveInfoVisitor
accept(visitor)
visit(this)
ChateauEntry()
entry
:ChateauEntry
DbImpl()
db
:DbImpl
save(entry)
prepareFindStatement()
executeSelect
compare(result)
prepareUpdateStatement()
executeUpdate()

Figure 46 Save Info Business Logic Sequence Diagram

5.2.1.3 Data Management Layer

Many frameworks and commercial off-the-shelf (COTS) products exist which provide data management functions. The design considerations for the data management include created models of how the persistent information of the system will be organized. Similar to class diagrams, entity relationship diagrams are commonly used to model data tables. Each table defined has a primary key which is unique within that table. This allows tables to only store a unique reference to information in a row, called a foreign key. The database diagram designed for this system represents the persistent information the system will store.

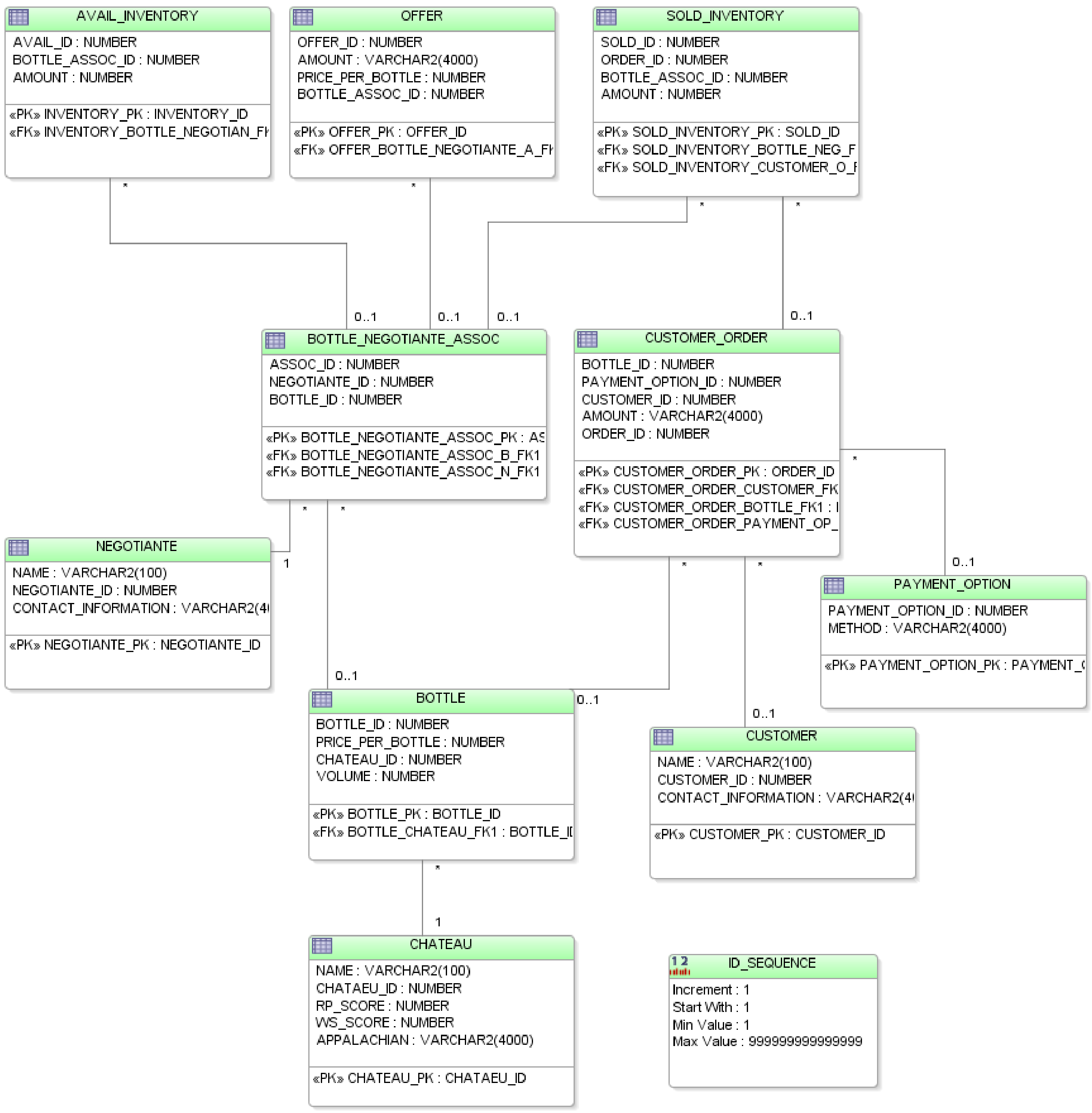

AVAIL_INVENTORY
AVAIL_ID : NUMBER
BOTTLE_ASSOC_ID : NUMBER
AMOUNT : NUMBER
«PK» INVENTORY_PK : INVENTORY_ID
«FK» INVENTORY_BOTTLE_NEGOTIAN_FI
OFFER
OFFER_ID : NUMBER
AMOUNT : VARCHAR2(4000)
PRICE_PER_BOTTLE : NUMBER
BOTTLE_ASSOC_ID : NUMBER
«PK» OFFER_PK : OFFER_ID
«FK» OFFER_BOTTLE_NEGOTIANTE_A_FI
SOLD_INVENTORY
SOLD_ID : NUMBER
ORDER_ID : NUMBER
BOTTLE_ASSOC_ID : NUMBER
AMOUNT : NUMBER
«PK» SOLD_INVENTORY_PK : SOLD_ID
«FK» SOLD_INVENTORY_BOTTLE_NEG_F
«FK» SOLD_INVENTORY_CUSTOMER_O_I
BOTTLE_NEGOTIANTE_ASSOC
ASSOC_ID : NUMBER
NEGOTIANTE_ID : NUMBER
BOTTLE_ID : NUMBER
«PK» BOTTLE_NEGOTIANTE_ASSOC_PK : AS
«FK» BOTTLE_NEGOTIANTE_ASSOC_B_FK1
«FK» BOTTLE_NEGOTIANTE_ASSOC_N_FK1
CUSTOMER_ORDER
BOTTLE_ID : NUMBER
PAYMENT_OPTION_ID : NUMBER
CUSTOMER_ID : NUMBER
AMOUNT : VARCHAR2(4000)
ORDER_ID : NUMBER
«PK» CUSTOMER_ORDER_PK : ORDER_ID
«FK» CUSTOMER_ORDER_CUSTOMER_FK
«FK» CUSTOMER_ORDER_BOTTLE_FK1 : I
«FK» CUSTOMER_ORDER_PAYMENT_OP_
NEGOTIANTE
NAME : VARCHAR2(100)
NEGOTIANTE_ID : NUMBER
CONTACT_INFORMATION : VARCHAR2(4
«PK» NEGOTIANTE_PK : NEGOTIANTE_ID
PAYMENT_OPTION
PAYMENT_OPTION_ID : NUMBER
METHOD : VARCHAR2(4000)
«PK» PAYMENT_OPTION_PK : PAYMENT_(
BOTTLE
BOTTLE_ID : NUMBER
PRICE_PER_BOTTLE : NUMBER
CHATEAU_ID : NUMBER
VOLUME : NUMBER
«PK» BOTTLE_PK : BOTTLE_ID
«FK» BOTTLE_CHATEAU_FK1 : BOTTLE_II
CUSTOMER
NAME : VARCHAR2(100)
CUSTOMER_ID : NUMBER
CONTACT_INFORMATION : VARCHAR2(4
«PK» CUSTOMER_PK : CUSTOMER_ID
CHATEAU
NAME : VARCHAR2(100)
CHATAEU_ID : NUMBER
RP_SCORE : NUMBER
WS_SCORE : NUMBER
APPALACHIAN : VARCHAR2(4000)
«PK» CHATEAU_PK : CHATAEU_ID
ID_SEQUENCE
Increment : 1
Start With : 1
Min Value : 1
Max Value : 999999999999999
*
0..1
1

Figure 47 Database Diagram

CHAPTER VI
IMPLEMENTATION

After object-oriented analysis and design is complete, the next stage is implementation where the design models are translated into actual software applications. This labor intensive process includes activities such as selecting commercial off-the-shelf (COTS) products, programming new components, and integrating everything together. Fully implementing the enterprise application is outside the scope of this paper, but an interesting discussion can be had regarding implementation decisions which need to be made. Each layer identified during design requires its own separate selections, and this chapter will select potential technologies which could be used to successfully build the solution. These choices are often heavily influenced by environment variables such as previous experience and business partnerships. For example, in this paper Java has been selected as the primary programming language. While other languages exist which could implement the solution, Java is the best choice because it can satisfy the design and personal experience heavily favors its use.

To effectively design a series of subsystems, a high level solution needs to be developed and technologies selected to build the application with. The producers of Java, Sun Microsystems, released an edition which was designed to meet the needs of web application programmers, Java 2 Enterprise Edition (J2EE). A typical J2EE application is multitiered as shown in Figure 48. The tiers shown in the figure correspond to the layers identified during design.

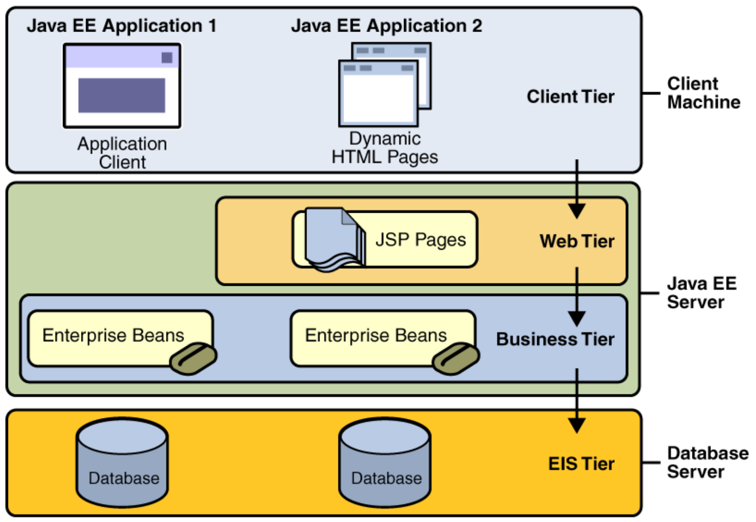


Figure 48 J2EE Multitiered Applications [Ball 2006]

To facilitate these applications, business functions are performed in the middle tier, which in this case would be an application running inside a Java EE server. J2EE offers a number of complimentary frameworks which can be leveraged to build this application.

6.1 Presentation Layer

6.1.1 Implementing the Client

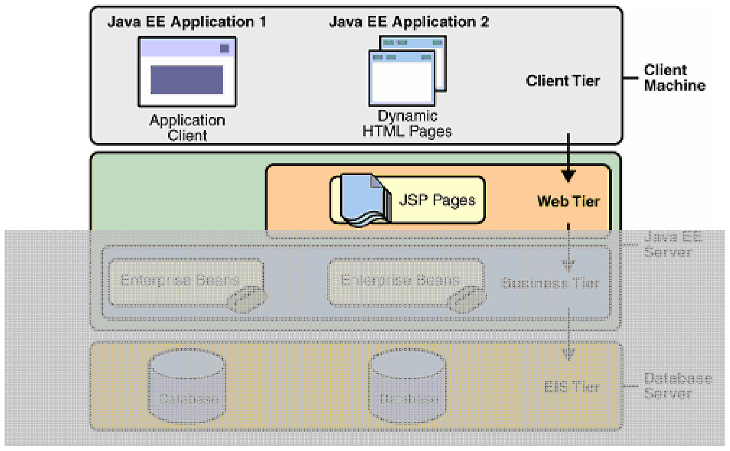


Figure 49 J2EE Presentation Layer

In J2EE, the most open ended implementation decision is how the client will access the system, and thus requires the largest amount of research regarding options. Figure 49 J2EE Presentation Layer shows two generic categories of client which access the Java EE Server, Application Client and Dynamic HTML Pages. These categories can even more broadly be referred to as thick client and thin client. To decide which client type would best fit this solution, an understanding of the pros and cons of each is necessary.

6.1.1.1 Thick-client versus Thin-client

In a client-server application, the system architect has to decide how much of the application should run on the client and how much on the server. This decision must be made early on since it will affect many aspects of the project including security, flexibility and hardware.

In the terms of this project, a thin client can be defined as a lightweight component running on the user's machine which provides a mechanism for input to the server and displays responses. The server is responsible for performing business logic and other intensive tasks. A typical client application which provides thin client operations is a web browser.

A thick client is also an application which runs on the client machine, but much more processing tasks are done locally instead of on the server. The server would be used for data access, client communication and possibly persisting state. Typically, a thick client is an executable program which is installed and runs on through the client's operating system.

Thick clients tend to be used when the client is doing very intensive tasks which consume a lot of resources. They are also used when the application demands a feature rich interface which historically could not be delivered by thin clients. Thin clients however, typically are used in fairly straight forward interfaces such as data entry and management. The benefits of this tradeoff in interface are that the application can be administered and controlled through the server. When a new version is released, only a single point needs to be updated for all the clients to have access to it. Also, when a thin client is served through web pages, the application can be accessed on any computer with a connection to the network. This is in contrast to a thick client which would need to be installed before it could be used.

The trend over the past few years has been towards thin clients. As network speeds increase and markup languages used in web browsers become more sophisticated, a case has to be made to justify building a thick client (typically this argument is regarding the user interface).

After reviewing the functional requirements and understanding the needs of the user, a thin client solution has been selected. This should result in a clean functional application which will have the potential to be expanded to larger tasks in the future.

6.1.1.2 Thin Client Implementation

To build the thin client, elements of Java Enterprise Edition (J2EE) will be used. Sun Microsystems, the developer of Java describes Java EE as being designed to:

"support applications that implement enterprise services for customers, employees, suppliers, partners, and others who make demands on or contributions to the enterprise. Such applications are inherently complex, potentially accessing data from a variety of sources and distributing applications to a variety of clients." [Ball 2006]

Deciding how to implement the thin client is a very complex task. A substantial amount of research was performed for this paper, and could be its own research project. A summary of the research has been included in Appendix I for reference.

6.2 The Business Tier

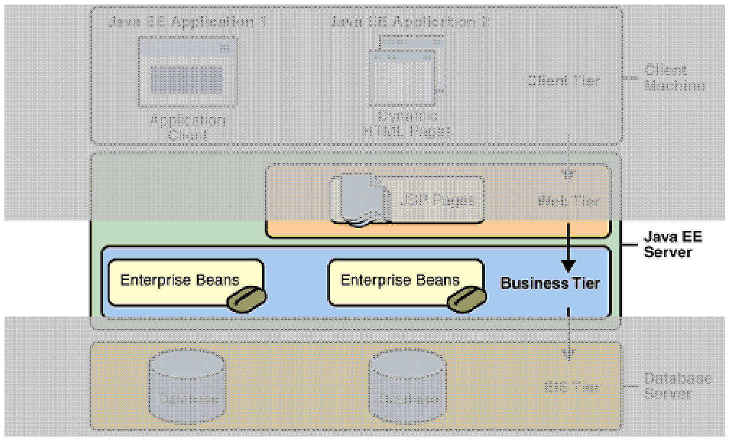


Figure 50 J2EE Business Tier

In J2EE, the business tier of an application is commonly implemented using Enterprise Java Beans (EJBs). An EJB is a "server-side component that encapsulates the business logic of an application" [Ball 2006] By invoking the interfaces defined in the EJB, remote clients can access the business functions it provides.

The application server provides an enterprise bean container which manages resources used by the beans. This allows developers to focus on functionality and not worry about resource management issues such as database connections and security authorization. There are two types of EJBs defined for business logic, session and message-driven.

Session beans exist for one client and allow that client to make calls for specific functionality. Similar to http sessions, when the client is done the session is finished. A developer can choose between stateful and stateless session beans. Stateful is a unique conversation between an itself and a client. Stateless contains no client specific information and can be used by any client once done with a request.

Message driven beans allow for messages to be processed asynchronously. These are commonly implemented as JMS listeners. Unlike session beans, clients do not connect to specific beans but instead send the message to a specific location to which the message beans listen.

Each type of bean has its own lifecycle. In the stateful session lifecycle, when the client makes a connection the bean is created or returned from a passive state. The server manages these beans and will move it between the three stages (does not exist, ready, and passive) depending on timeout settings and client use. Stateless session beans never go passive and thus move between not existing and ready. Message driven bean lifecycles are similar to stateless session beans. The bean is either ready or does not exist. [Ball 2006]

For the implementation of the solution presented in this paper, decisions will need to be made for when a stateful session bean is required versus a stateless one and when a message bean will be required to perform asynchronous work.

6.2 Data Management Tier

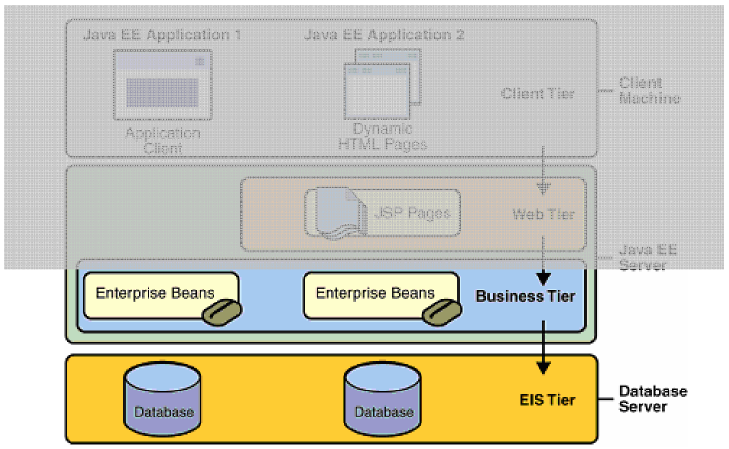


Figure 51 J2EE Data Management Tier

The implementation decisions which need to be made for the data management tier primarily have to do with how to persist information. The widely accepted storage mechanism for large amounts of information is through a relational database, but the decision which needs to be made is how much of the information should be accessible directly in the business tier.

J2EE offers a persistance mechanism called entities, which are a "lightweight persistence domain object." These special types of beans typically are built to mimic the underlying database tables, such that an individual instance represents one table row [Ball 2006]. The use of entity beans allows the developer to abstract the schema of the database from the business logic performed by sessions beans. Also, entity beans are persistant regardless of an individual session, so multiple clients access the same set of information. The primary disadvantage of using entity beans to model is performance hits when a large amount of information is being represented. Representing millions rows in memory can become quite cumbersome to an application server.

When making the decision to use session beans with database calls or entity beans, the following concerns should be weighed. [Gabhart 2003]

- **Read/write needs.** Session beans are the best choice for data that is read often but changed rarely. If frequent data manipulation occurs frequently, then entity beans provide the better solution.
- **Transactional support.** Entity beans shield the developer from having to manage details of transactions. How the container will manage them is predefined in the deployment descriptor of the beans. For maximum control of transaction handling, session beans can be used.
- **Time to market.** Entity beans easily represent the single fastest time to market of any J2EE persistence mechanism, but this must be weighed against the resource usage discussed next.
- **Resource usage.** As mentioned before, entity beans consume a large quantity of resources. In comparison, session beans with database connections are lightweight and require much less server resources.

CHAPTER VI
SUMMARY AND CONCLUSIONS

This paper has focused on object-oriented analysis and design principles recommended by educational text and are commonly used in commercial application development. These activities where leveraged to produce design artifacts which could be used to implement a functional software solution for use in the wine futures industry.

The first task was to collect requirements of the system to be built. While many types of requirements exist, only functional requirements were gathered for purposes of scope. This was done using the use-case analysis tool to identify actors to the system and operations they would perform. Each use-case included details of the purpose and flow of the action. From the use-cases, formal functional requirements were defined in a manner consistent with commercial requirement procedures.

Once requirements were defined, object-oriented analysis was performed to understand the system in object-oriented terms. The first activity of analysis was to create system sequence diagrams which translated each use-case into a diagram that showed specific system operations over the flow of time. Next, candidate classes were identified and used to create a domain model which showed associations of objects in the real world system. Thirdly, operation contracts were formed which specified in detail how the system was modified as a result of each operation performed on it.

Object-oriented analysis provided a definition of the system in the real world and operations which could be performed. These artifacts were then used to design the software system. First the tiers of the system were defined and each system sequence diagram was expanded to show how the internal tiers of the system would respond to system operations. Then, each tier was designed using object-oriented design techniques. The presentation layer was designed using a command heirarchy tree which allowed each use case to be satisfied through traversal. The business logic tier was designed using sequence and class diagrams which showed how operations would flow over time and the structure of objects in the tier. The data management tier was designed using entity-relationship diagrams to define data tables and the relationship between them.

Finally the topic of implementation was discussed for each tier in the system. Actual implementation would involve building the system designed, but that would be well outside the scope of the paper. Instead, for each tier implementation decisions which need to be made were discussed with multiple options presented.

While a large amount of work was done creating the artifacts presented in this paper, this is only a small amount of the work required to take a problem statement and turn it into a fully functional software solution. Process is the key to taking a problem through the necessary stages, and exactly which process is best is a topic frequently debated and documented. Regardless of how many cycles occur or how many stages are defined, the principles of object-oriented analysis and design remain a solid and productive means for designing high quality software solutions.

REFERENCES

1. [Alhir 2003] Learning UML. **Author:** Sinan Si Alhir. **Publisher:** O'Reilly Media, Inc. Sebastol, 2003.
2. [Asleson 2006] Foundations of Ajax. **Author:** Ryan Asleson and Nathaniel T. Schutta. **Publisher:** Apress. Berkley, 2006.
3. [Ball 2006] "The Java EE Tutorial." **Author:** Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, Kim Haase, and Eric Jendrock. **Website:** http://java.sun.com/javaee/5/docs/tutorial/doc/index.html.
4. [Bray 97] "Object-Oriented Analysis." Author: Mike Bray, Lockheed-Martin Ground Systems. Web**site:** http://www.sei.cmu.edu/str/descriptions/ooanalysis.html. 1997
5. [Champeon 2001] "Javascript: How Did We Get Here." **Author:** Steve Champeon. **Website:** http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html. 2001.
6. [Gabhart 2003] "J2EE pathfinder: Persistent data management, Part 1." **Author:** Kyle Gabhart. **Website:** http://www-128.ibm.com/developerworks/java/library/j-pj2ee3.html.
7. [Gamma 1995] Design Patterns Elements of Reusable Object-Oriented Software. **Author:** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. **Publisher**: Addison-Wesley, 1995.
8. [Model 2006] "Model View Controller." **Website:** http://patternshare.org/default.aspx/Home.PP.ModelViewController
9. [Perrone 2003] J2EE Developers Handbook. **Author:** Paul J. Perrone, Venka S. R. Chaganti, and Tom Schwenk. **Publisher**: Sams Publishing. Indianapolis, 2003.
10. [Pressman, 2001] Software Engineering A Practitioner's Approach. Fifth Edition. **Author:** Roger S. Pressman. **Publisher**: McGraw-Hill Higher Education. New York, 2001.
11. [Sommerville 2001] Software Engineering. Sixth Edition. **Author:** Ian Sommerville. **Publisher:** Pearson Education Limited. Essex, 2001.
12. [Valtech 2003] "Object-Oriented Analysis and Design using UML, Patterns, and the Unified Process." **Release:** OOAD-4.0.3-En. **Publisher:** Valtech Technologies, Inc.

APPENDIX A
THIN CLIENT RESEARCH

A.1 Java Applet

One option which has been around since Java was first introduced is Java applets. Java applets where introduced with the first release of Java in 1995 as an option to providing a rich user interface inside a browser. Web programmers could use the same user interface components that are available for desktop applications inside an applet, making it easy to embed an existing application into a webpage, or write a completely new applet without having to learn a different language.

While this is very convenient to programmers, applets have always had disadvantages which lead to many web developers not preferring to use them. For one, the user of the web browser must have the Java plugin installed on their computer to run the applet. If the browser does not have the plugin, they must download it off of the provider's website. With the pervasiveness of high speed internet connections over the last few years, download time is no longer the concern it used to be, but it is still an extra step the end user must perform to use the web application. This distracts from the concept of a web application being available to anyone with an internet connection and a web browser.

Another disadvantage is that in order for the Java plugin to use the applet, the applet must be downloaded from the website's server. The size of an applet can vary from in the low kilobytes to many megabytes, and this places a waiting time on the end user to begin interaction with the application. If the user's browser is not set up to cache information, the applet will be downloaded every visit to the page.

A third disadvantage is that applets run on the client's machine inside the Java virtual machine installed on the computer. This has a number of implications. The performance of the applet will depend entirely on the hardware specifications of the client machine. This means the user experience could very greatly between different users with different machines.

Also, by running on the client machine, the applet, although embedded inside the webpage, is running outside the web application. This may not be a major concern if the applet is the entire application, but it is common to see an applet performing specific tasks inside a larger web application. While the rest of the page is running on the server and performing business logic, the applet is running on the client machine, and this requires programmers to create inventive workarounds to send information back to the server on out into the page through a scripting language.

So why would a programmer choose an applet to produce a web application? There are a number of advantages to having a Java applet embedded in the web page. As previously mentioned, Java programmers do not have to learn a new language to develop for the web. Also, Java applets do provide a very rich user interface possibilities that until recent years was unmatched by other web technologies. For example, Java allows for very complex components (such as trees, tabs, scroll panes, drawing canvases, and many input options) to be integrated together. Each of these components can be interacted with without having the webpage refreshed. Other web technologies which render html code can provide similar controls, but in order for an action to be processed, the event must be sent back to the server and a new html page generated. This can be particularly distracting and not to mention time consuming for the end user to perform what is a trivial interaction in Java. Another advantage is that java applets can take some of the processing load off of the host server. As mentioned before, applets run on the client machine and this alleviates memory requirements and transaction processing from the server. This is particularly attractive if the end user has a slow internet connection or if the host has a relatively small server.

A.2 Client Side Scripting

Java applets are not the only technology that offers client side interaction without requiring communication with the server. Scripting languages run inside the browser as well and provide similar interface possibilities. The most widely used scripting languages support ECMAScript which is the standardized version of Javascript. Javascript was first introduced by Netscape in 1995 and was quickly accepted by the programming community. Due to Javascript's success, Microsoft released their own scripting language known as JScript, which was roughly compatible with Javascript. After Netscape submitted Javascript for standardization in 1996, the first edition of ECMA-262 was adopted in June 1997. New editions of Javascript and JScript aim to be standard with ECMA, while providing their own unique functionality. [Champeon 2001]

Over the year's the possibilities of what can be done with scripting have increased dramatically. While scripting is commonly used to do client side validation and dynamic effects in the page, true interface components have begun to emerge as user's expectations of what a web application can be have risen and developers become more creative to accommodate the demand.

Scripting languages typically are not used for business logic operations, but are instead used to provide a rich user experience on top of another web technology such as servlets or portlets.

Scripting languages do have their disadvantages. The primary one is that different browsers support different versions of these languages, and a user on Internet Explorer may be able to do many things that a Mozilla or Opera user cannot do. Also, support varies across different versions of the same browser. Many computers over five years old require updated version of browsers to be installed because what would not run on an old version will now run in the latest releases. This requirement of the user results in many developers only using basic scripting components that are supported across all browsers and this limits the possible interface opportunities offered by scripting languages. Also, users may have security permissions set up to prevent complex scripts from running inside their browsers since malicious code inside a page can damage computers.

Resurgence in interest in client side scripting has come about recently due to Ajax. Ajax is the newest buzzword in the web development community, but really is only an acronym for a group of technologies used in conjunction to do asynchronous communication between a web browser and a server. The acronym stands for Asynchronous Javascript + XML, but the term encompasses all the technology required to perform asynchronous transactions.

Ajax uses existing scripting technologies built into most modern browsers to send a request to the server and receive a response that allows for portions of the page to be updated. This rather simple concept has already taken the web developer community by storm. By having the ability to update only portions of a web page at a time, one of the major advantages of Java applets is eliminated. In addition, Ajax can be used with any server side technology which means it does not limit the solutions to choose from. It can just be used to improve whatever is chosen. [Asleson 2006]

A.3 Server Side Application

A server side application generates the user interface responses locally and then returns them to the client to view. Every major Java based framework which provides this functionality is based on the earlier web development technologies such as Servlets, JavaServer Pages (JSPs), and Java Beans. These building blocks have allowed more complex technologies to be built which organize application logic into design patterns such as model-view-controller.

A servlet is essentially a java program which runs on a web server. They allow developers to produce dynamic web pages and perform business logic on the server. Typically a servlet performs tasks such as reading and processing data from a request, generating results with the existing server side information and returning a response in a markup format.

JSP's are text documents which reside on the server containing a mixture of a markup language (such as HTML) and Java code. The java code is compiled by the server at runtime the resulting output is a dynamic response containing values returned by the java code.

A Java Bean is simply a Java class which conforms to naming conventions defined by Sun (the developer of Java). A typical Java Bean is used to store values and provides methods for accessing and altering them (getters and setters). [Ball 2006]

Developers integrate these three building blocks to provide web applications where servlets handle control of the pages and JSP's handled the presentation. There are many groups which have developed frameworks to simplify common tasks all web developers must perform when working with serlets and JSPs. Most revolve around the Model-View Controller pattern (MVC). This pattern is commonly used whenever designing an application with a user interface. This pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate categories [Model 2006]:

1. A model that represents the data for the application.
2. The view that is the visual representation of that data.
3. A controller that takes user input on the view and translates that to changes in the model.

A.3.1 Apache Struts

Apache Struts was founded to provide a standard Model-View-Controller framework from these building blocks. It is one of many frameworks independently developed to solve the same problem. The struts framework uses java beans and servlets to provide access to the model, jsps for the view, and the controller was written by the group to handle requests from the view to the model. The controller is itself a servlet that uses mappings defined in an xml document to know how to forward specific

While Struts provides a clean distinction between the model, view, and controller some criticisms come from developers. Some say that struts is too “code heavy” and does not provide rapid application development opportunities. Others feel that the framework is out of date and works at too low a level inside the JSP’s. Newer frameworks offer tags and other mechanisms to abstract out low level coding inside the page. Also, while Struts is one of the more popular web development frameworks available, it is not standardized and competes with a large number of other similar projects.

A.3.2 JavaServer Faces

First introduced in 2002, JavaServer Faces is a web application framework which aims to ease some of the headaches of Java web development by providing a controller and request lifecycle similar to Java Swing. In addition to this control architecture, JSF is built with components allowing developers to reuse presentation code and extend them to provide customized controls. Each of these components is mapped to a developer defined backing bean which serves as the entry point from the web browser to the server side application. JSF is a standardized through the Java Community Process which encourages vendor support in servers and IDEs. [Ball 2006]

JavaServer Faces is still a relatively new technology which results in the typical disadvantages of more well established frameworks such as documentation availability.

Also, as the product matures new releases come out providing bug fixes and extended functionality. This can be hard on developers who already have a learning curve to overcome.

A.4 Choosing a Presentation Framework

Choosing between a client side web application and server side application is difficult. The newest web technologies remove many of Java applet's traditional advantages, meaning that deciding to use an applet would be one purely of comfort with the older technology. Client side scripting is very appealing because of the promise of responsive controls. Ajax is especially exciting because having to refresh web pages to process a request is very frustrating for users.

On the server side, the two frameworks discussed both show promise. JavaServer Faces is the most appealing because it is a standardized framework and is on the cutting edge of web development. Struts is a reliable framework, but is viewed as an aging dinosaur by many. Also, JSF has predefined html components which will save time on the presentation layer of the application.

A.5 JavaServer Faces In Depth

This section summarizes the JSF introduction from [Ball 2006]

The life cycle of a JavaServer Faces page is more complicated than a simple JSP page. A multi-stage lifecycle occurs to support the UI component model. These stages validate the user input, handle events, and propogate date to beans. A JavaServer Faces page is represented by a tree of UI components, called a view. During the life cycle, the JavaServer Faces implementation must build the view while considering state saved from a previous submission of the page. Following is a diagram of the JavaServer Faces lifecycle.

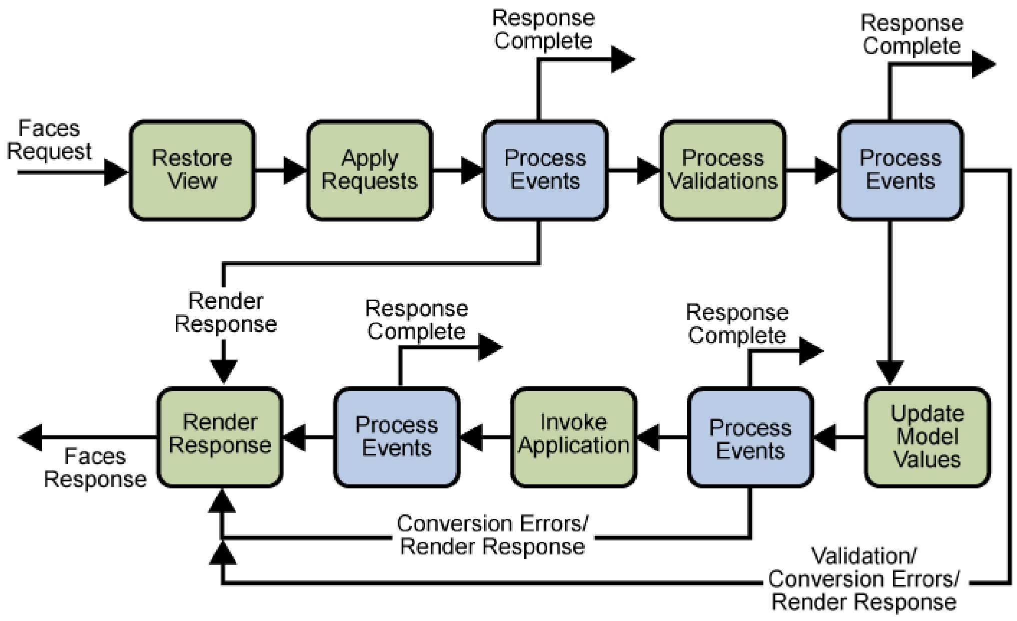


Figure 52 JavaServer Faces Lifecycle [Ball 2006]

There are two types of requests that the browser must handle, initial requests and postbacks. Initial requests occur the first time a user calls a page and only the restore view and render response phases since no inputs where provided. When a postback occurs, the entire lifcycle takes place.

A.5.1 Restore View Phase

The restore view phase begins when a request for a JavaServer Faces page is made. During this phase, the FacesContext instance is created which contains all the information needed to process the request. This includes a view of the page, event handlers and validators. The view is either created empty (initial request) or restored from state information sent in.

A.5.2 Apply Request Values Phase

In this phase, each component extracts its new value from the component tree. During this time event listeners can be notified. If the value for the component is invalid, an error message associated with the component is placed in the FacesContext to be displayed later. During this phase a component or event listener can force the lifcycle to skip directly to the render response phase.

A.5.3 Process Validations Phase

During this phase, the JavaServer Faces implementation processes all validators registered on the components in the tree. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. If validation fails, an error message is captured and returned to the user.

A.5.4 Update Model Values Phase

After the JavaServer Faces implementation determines that the data is valid, it can walk the component tree and set the corresponding server-side object properties to the components' local values.

A.5.5 Invoke Application Phase

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

A.5.6 Render Response Phase

During this phase, the JSP container is given control of rendering the page. If this was an initial request, the components are added to the component tree, otherwise they are already there. After the content of the view is rendered, the state of the response is saved so that future requests can access it and it is available to the restore view phase.

A.5.7 JSF Components

JavaServer Faces provides a rich, flexible architecture for UI components. These components can be as simple as a button or compound such as components nested inside a table. To facilitate this JavaServer Faces technology provides the following component functions:

- A set of UIComponent classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in various ways
- An event and listener model that defines how to handle component events
- A conversion model that defines how to register data converters onto a component
- A validation model that defines how to register validators onto a component
- This section briefly describes each of these pieces of the component architecture.

A.6 Ajax

With an understanding of the lifecycle and models provided with JSF, lets take a look at how Ajax works. The javascript object which allows asynchronous communication between the browser and server is XMLHttpRequest. Although not a standard, XMLHttpRequest is supported by the major internet browsers.

XMLHttpRequest is used by creating an instance of the object, registering a listener funtion to it, and sending a request to the server. This request is very similar to a normal HTTP request and can include parameters containing information required to perform the operation. When the server is done with the request and the response is received, the function registered as a listener is called and the response can be worked with. The response from the server is typically xml, but can be plain text. The scripts in the page can then use the reponse to customize the page for the user. All of this happens without the page refreshing in the browser. [Asleson 2006]

A.7 Putting JSF and Ajax together

While JavaServer Faces is designed very similar to Java Swing in that it has individual components for different pieces of the interface, the resulting page suffers from the normal pitfall of web applications, page refreshes. As mentioned before, Ajax works independent of the server side implementation of the web application. This gives us an opportunity to leverage Ajax's asynchronous qualities to provide component updates in JavaServer Faces.

The web development community has seen this opportunuty and already begun designing components that achieve the goal of asynchronous components. Some of these are being developed through open-source communities and other are commercial, but the resources are available to design web applications effectively with these tools.