



Games Digital Distribution Platform

SYSTEMS ANALYSIS AND DESIGN

CS4125

April 21st, 2011

xxx	0000
xxx	0000
xxxx	0000

TABLE OF CONTENTS

1. Project Description.....	5
1.1 Narrative	5
1.1.1 Introduction to Digital Distribution.....	5
1.1.2 Gamer	5
2. Requirements Summary.....	7
2.1 To record information about users and their games.....	7
2.2 To have a store of purchase-able games	7
2.3 To have a wide range of social network functionality	7
2.4 Non Functional Requirements	7
3. Software Life cycle model	8
4. Project Plan and allocation of roles	10
4.1 Timeline & Deliverables	10
4.2 Roles.....	12
5. System Architecture.....	13
5.1 UML Workbench	13
5.2 Architectural Decisions Taken.....	13
5.3 Package Diagram.....	14
6. Requirements.....	15
6.1 Use Case Diagrams.....	15
6.1.1 Login.....	15
6.1.2 Logout	16
6.1.3 Browse Store.....	17
6.1.4 My Games	18
6.1.5 Create Account	19
6.2 Structured Use Case Descriptions.....	20
6.2.1 General Use Case Descriptions	22
6.2.2 Store Use Cases Descriptions.....	23
6.2.3 My Games Use Case Descriptions.....	24
6.2.4 My Community Use Case Descriptions	25

6.3 Detailed Use Case Descriptions	26
6.4 Non-Functional Requirements.....	31
6.4.1 The 'ilities'	32
6.5 Screenshots / Mock-ups / report formats	35
6.5.1 Login View.....	35
6.5.2 Register View	35
6.5.3 Main Application View	36
7. Analysis Diagrams	37
7.1 Identifying the candidate classes.....	37
7.1.1 Noun Identification Technique	37
7.1.2 Sample Nouns Identified.....	37
7.1.3 Evaluation Heuristics.....	38
7.2 System class diagram	39
7.3 Communication Diagrams.....	40
7.3.1 Purchase Communication Diagram	40
7.3.2 Login Communication Diagram.....	41
7.4 State Charts.....	42
7.5 Entity Relationship Diagram.....	43
7.6 Sequence Diagram	44
8. Design.....	45
8.1 System Class Diagram	45
8.2 Account Package	46
8.3 Cart Package.....	47
8.4 Database Package	48
8.5 Payment Package	49
8.6 Item Package	50
8.7 Session Package	51
8.8 Item Management Package	52
8.9 Window Management Package	53
8.10 Network Package	54

9. Design Discussion.....	55
9.1 Design Patterns used	55
9.1.1 Singleton Pattern	55
9.1.2 Model View Controller.....	56
9.1.3 Observer Pattern.....	58
9.1.4 State Pattern	59
9.2 Design Patterns Considered	60
9.2.1 Memento Pattern	60
10. Data Dictionary.....	61
11. Critique.....	66
12. References.....	69

1. PROJECT DESCRIPTION

1.1 NARRATIVE

1.1.1 INTRODUCTION TO DIGITAL DISTRIBUTION

With the proliferation of high-speed broadband and larger download capacity per user, digital downloads of software, movies, music and books have become a big industry. The gap in the market of selling game software that appeals to the affluent age group of 16 to 30 is one that we can fulfil.

The games industry is one of the fastest growing entertainment industries in the modern digital age. They took in about \$9.5 billion in the US in 2007, and 11.7 billion in 2008, showing a 25% per annum. Given these statistics, there is plenty of profit to be made in an ever growing market with only a few big competitors, customers growing daily and a digital pc gaming market of over \$6 billion.

1.1.2 GAMER

This enterprise application will be built by the new company, Gamer, which was setup by David O'Neill, Gavin Fitzgerald and Yang Zhang in 2011. Our company plan to offer a digital distribution service unlike any other currently available.

A key advantage of our system is its' support for multi-platforms, allowing it to run on Windows, Macintosh, Touch Tablets and some Linux distributions, which our competitors strategically fail to, offer e.g. a customer, can easily back-up their game files and migrate it from one OS to another.

We identified a key benefit in using Peer-to-Peer distribution of the game software bundles thereby cutting costs and actually increasing download speed. We intend our system to have a high quality ease of use, as we wish the system to be used by more than just the "hardcore" technically informed customers. We wish to reach a wide-age bracket, of 10 to 40. Therefore a core tenant of the system will be to have a modern level of human computer interaction heuristics that will incorporate age-based specific themes that appeal to different age groups.

The advent of social networking and 'smart' devices is something our system can benefit from, as customers wish to add their friends, see what they're playing, join their game and even do all this on the go with their smart device. Similarly we can offer a Touch-based selection of games as a competitor to the iOS AppStore and Android Marketplace.

In terms of security and secure payments we're partnering with 3 competing companies, PayPal, iDeal and WebMoney as we wish to offer our customers a wide range of payment types with the flexibility of adding new secure payment providers as the online payment industry changes. Our

payment system must be flexible allowing users to add and remove items before completing the purchase as empowering the customer is clearly beneficial.

With the system we intend it to be language-independent, where we can quickly localise it for different countries and languages around the world, being able to account for local rating laws and taxes.

As a product differentiator, we will re-make available many titles that have been out of print for some time, attracting consumers who have fond memories of titles that they can no-longer buy elsewhere. Similarly, the system will have the functionality to have special offers that have been a proven success at increasing sales when sales for a title drop off.

2. REQUIREMENTS SUMMARY

2.1 TO RECORD INFORMATION ABOUT USERS AND THEIR GAMES

- To record new user name, address, and other relevant details.
- To update user details.
- To delete a user account.
- To browse for user details and related game playing history
- To record details of each game for each user including length of time played
- To record desired games and owned games

2.2 TO HAVE A STORE OF PURCHASE-ABLE GAMES

- To have a browse-able store of games
- To have multiple digital payment sources
- To install and uninstall games from the user's library
- To delete a game from the store

2.3 TO HAVE A WIDE RANGE OF SOCIAL NETWORK FUNCTIONALITY

- To add friends to an account
- To see a friends status
- To message a friend
- To rate purchased games

2.4 NON FUNCTIONAL REQUIREMENTS

- To backup data every day.
- To allow the system to be deployed on the existing affordable and household computer hardware
- To ensure customer personal and payment details are private and secure
- To ensure the security of the game software
- To allow new item types to be added easily, if we expanded beyond games purely

3. SOFTWARE LIFE CYCLE MODEL

For our project we've decided to use an adapted Agile Software Development Model. We felt an iterative and incremental development would result in an application ripe for further growth. Continuous improvements are a great way of holding onto users, when they see that the functionality of the application is increasing regularly. The 12 principles of the Agile Manifesto paint an ideal picture that we feel will help make the project a competitive one, maximising our team's skills.

In Agile the team will break tasks into small iterations, preferably 2 or 3 weeks per iteration, depending on the size of the team at the time (flexibility/adaptability is a key component of Agile). These increments will have minimum pre-planning, as at the start of each iteration, the planning requirements and analysis design will be reviewed and refactored. In "pure" Agile, little to no long-term planning is involved, including a document as specific and detailed as this.

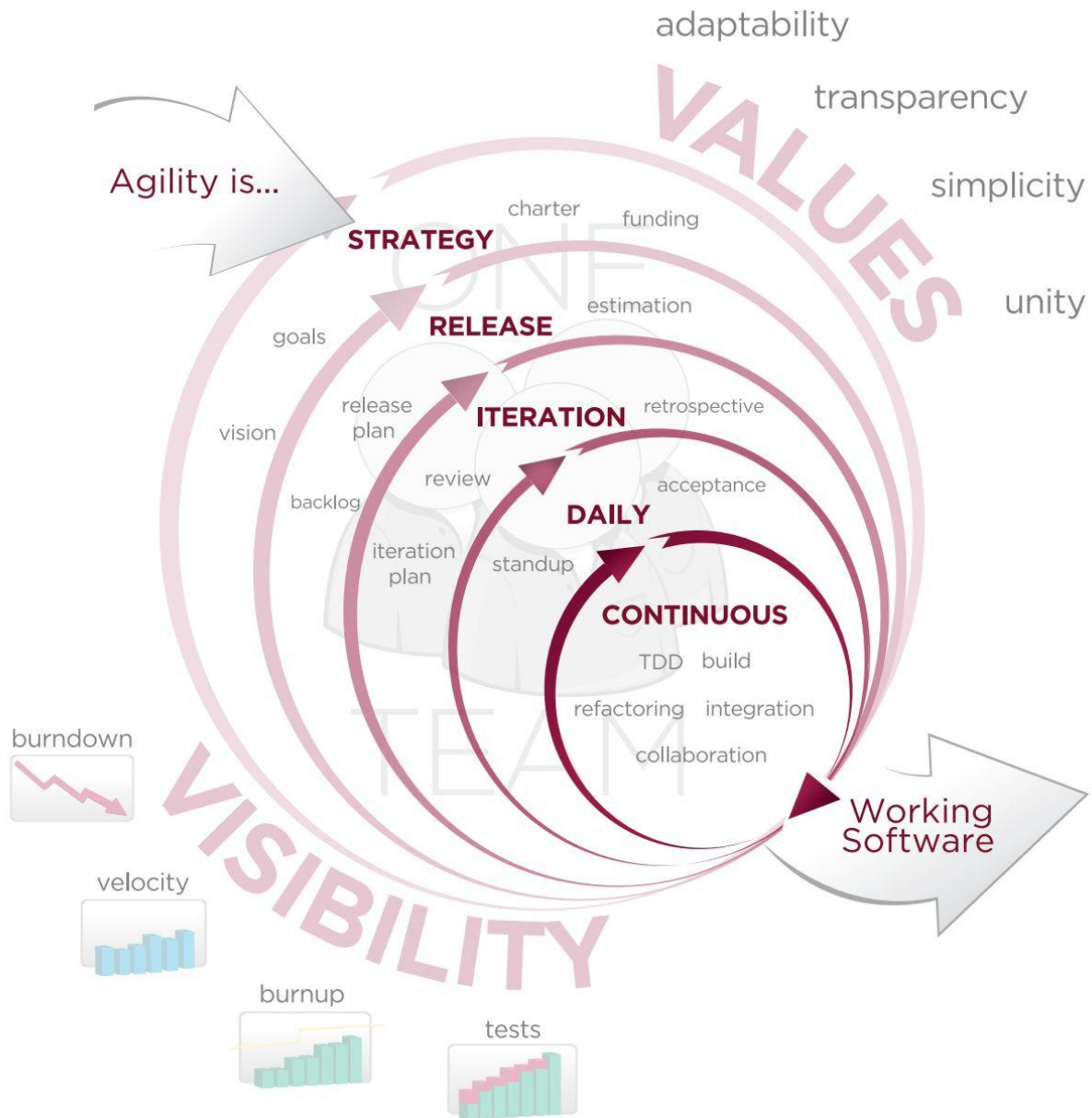
We feel a marriage of traditional development (Waterfall) and Agile will have the best outcome for our product as our stakeholders have a definite vision for the product but wish us to be ready to adapt in the rapidly growing and changing digital distribution industry. We realise that pure Agile is not suited to our team as we have little previous Agile development experience and would like to use this project as our first big foray into such an environment.

We've rejected a complete Waterfall development methodology as we feel the sequential nature of development means that many mistakes in the requirements, design, analysis or implementation will appear too late in the project in comparison to Agile. In Agile, we see minimal risk, as "it is a low over-head method that emphasizes values and principles rather than processes".

We recommend a change in office environment as Agile methods emphasize face-to-face communication over written documents and often use a technique called Pair Programming in which two programmers work together at one work station. One types in code while the other reviews each line of code as it is typed in. The person typing is called the driver. The person reviewing the code is called the observer (or navigator). The two programmers switch roles frequently. This should bring improved discipline and time management while promoting team friendliness and code confidence for the programmers involved.

We've included an explanatory/demonstrative diagram on the next page as we feel it is a simple and approachable explanation of what Agile Development really is.

AGILE DEVELOPMENT



ACCELERATE DELIVERY

4. PROJECT PLAN AND ALLOCATION OF ROLES

4.1 TIMELINE & DELIVERABLES

Week	Deliverables	Description	Allocated to
04	Narrative Description	Description of the project and overview of the system	xxx
04	Requirements Summary	Early desired functionality	xxx
09	Software Life Cycle Model	Discussion of the software model used during this project	xxx
11	System Architecture	Designing for Interoperability	xxx
Requirements			
05	Use Case Diagrams	Realisation of Client Interactions	xxx xxx
06	Use Case Descriptions	Breakdown of Client Interactions	xxx xxx
06	Non-Functional Requirements & Quality Attribute Tactics	Discussion of quality attributes	xxx
12	Screenshots / Mock-ups	Visualisation of Interaction	xxx
Analysis			
08	Candidate Classes	List candidates using noun identification technique	xxx xxx
09	System Class Diagram	Class diagrams showing inheritance/aggregation/compositio n...	xxx xxx

10	Communication Diagrams	Charted Method Calls	xxx
09	Sequence Diagram	Order of Interaction between Objects	xxx xxx
10	State Charts	State of Objects during run time	xxx
11	Entity relationship Diagram	Show relationships between classes	xxx
Design			
11	Overview	Covered in Design Discussion	xxx
11	System Class Diagram	Complete design class diagram	xxx xxx
12	MVC	Refactor to MVC	xxx
12	Other patterns	Refactor to patterns, investigate others	xxx
Design Discussion			
12	Patterns Used	Singleton, State, Observer	xxx
13	Patterns Considered	Memento	xxx
13	Data Dictionary	Depict notations in the UML Diagrams	xxx
Close			
13	Critique	Analysis and review of shortcomings of the project	xxx xxx
13	References	Sources used for learning/information	xxx xxx

4.2 ROLES

Each member of the team shared the Requirements, Analysis & Design roles with specific areas assigned to different team members as outlined in the project plan above.

5. SYSTEM ARCHITECTURE

5.1 UML WORKBENCH

For our project we chose Visual Paradigm as our preferred UML workbench. It provides support for UML 2, a rich diagramming toolset and both commercial and personal editions, which meant we were able to work off campus, unlike using Select. It provides the ability to interface with a subversion repository, offering us the ability to collaborate in our individual homes. It provides report generation and code engineering capabilities including code generation. Similarly it provides Business Process Modelling and Database Modelling Support. It offers interoperability functionality e.g. XML, .csv

5.2 ARCHITECTURAL DECISIONS TAKEN

The architecture of this project is sub-divided into 9 packages separating the data, the business logic and the UI. The data itself is split into local to the client and external data in databases. We built a rich user interface which allows our users to configure their own UI graphics which allows the application to appeal to all ages groups and not just target a certain age group.

The system provides support for new types of items to be sold through the generalisation of the domain concern game into a broader entity, Item. This will allow us to expand to selling other types of content e.g. Movies, Music, E-books etc.

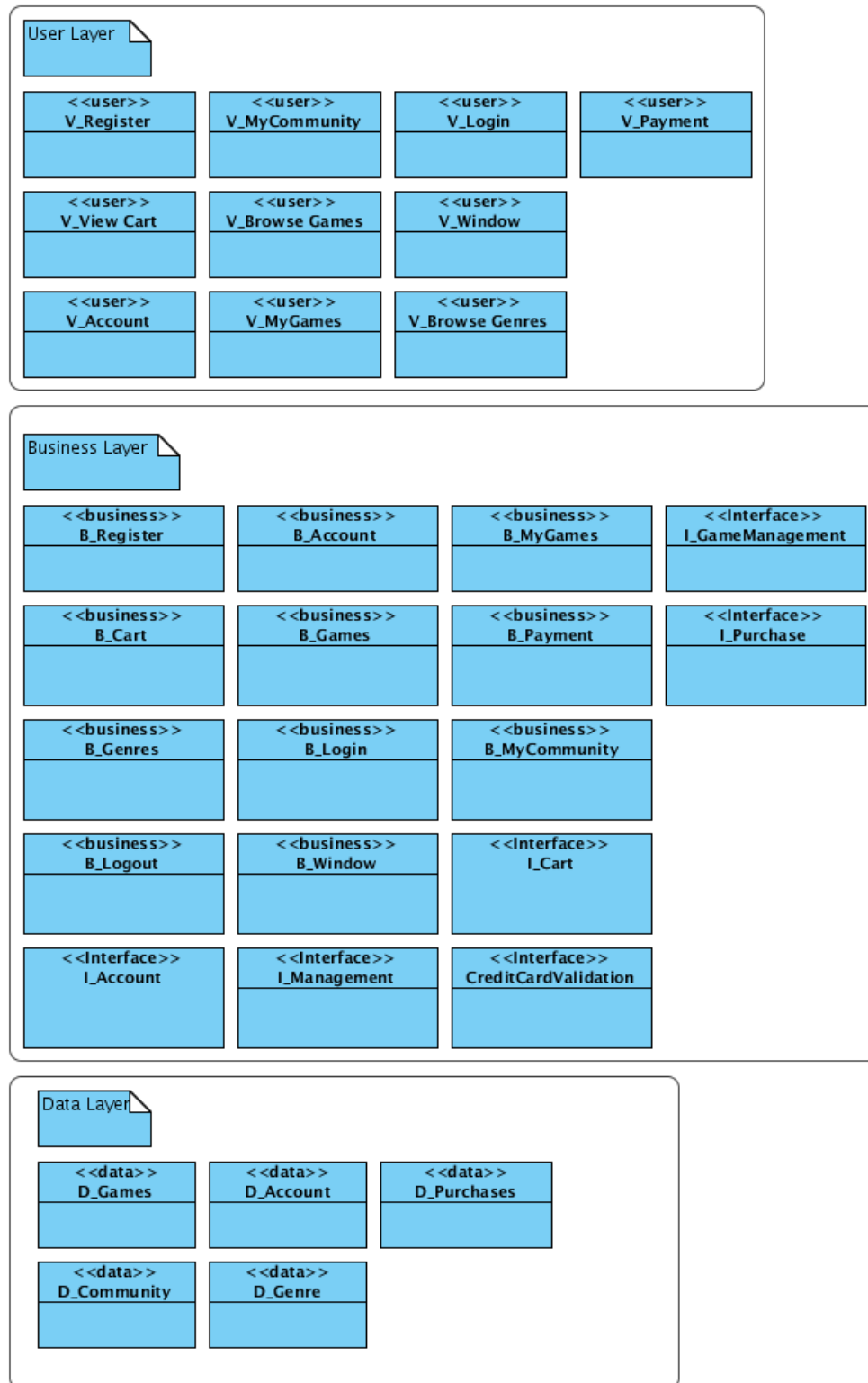
Due to the Component based design, combined with MVC, the system could be extended or modified to work in alternate domains or even domain-applications e.g. Document Viewers

During our design phase we didn't focus on any domain specific language, therefore implementation of the application could be delivered on many different hardware architectures or operating systems.

We generalised the 'Window' management in our design however further refactoring would allow for the use of the abstract factory pattern for the creation of native UI controls on targeting platforms.

Our application will allow a web interface for users to replicate browsing the store without the ability to install games i.e. functionality that will be client exclusive. It will also allow us to create a mobile widget for Smart Phones that will enable users to have access to the social functionality of the application.

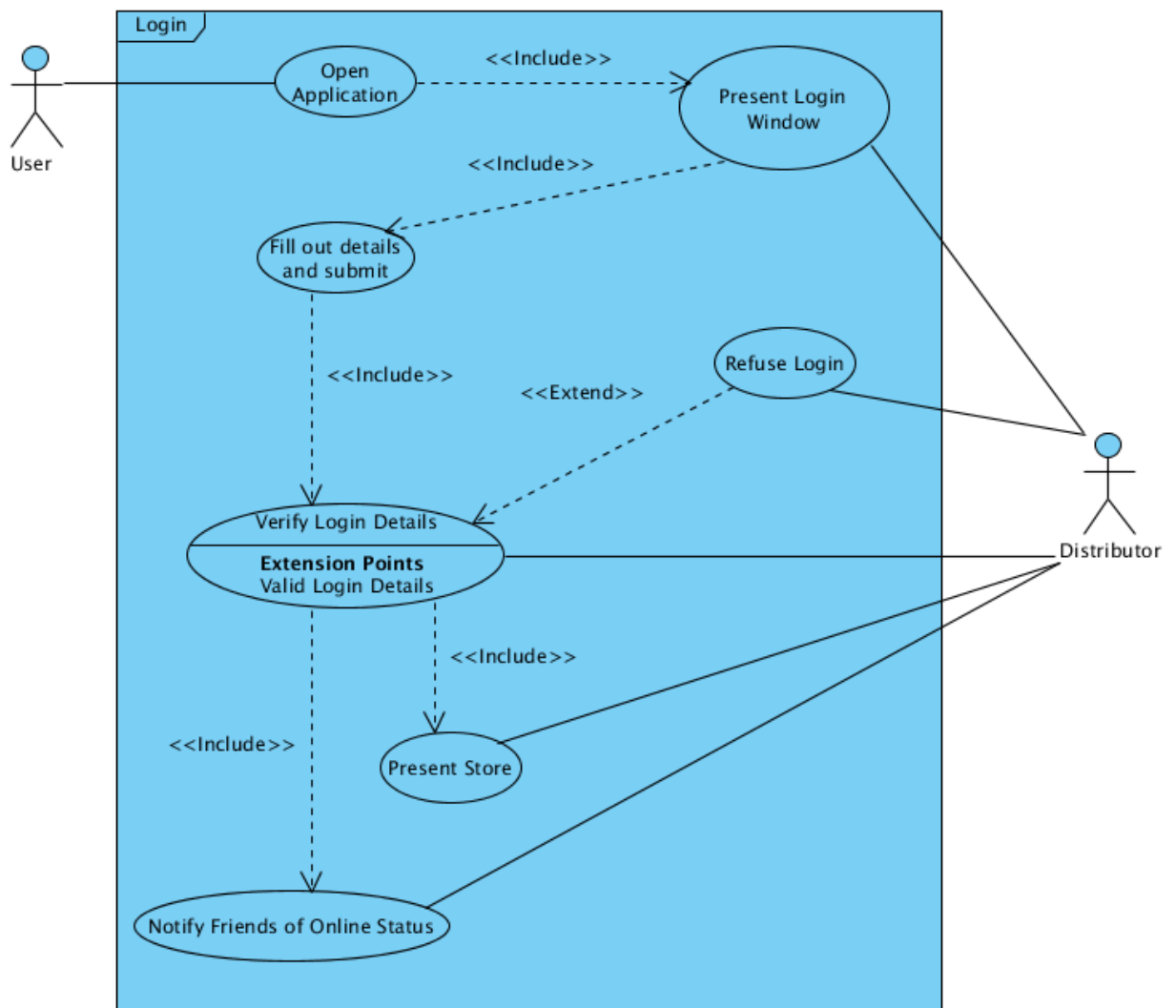
5.3 PACKAGE DIAGRAM



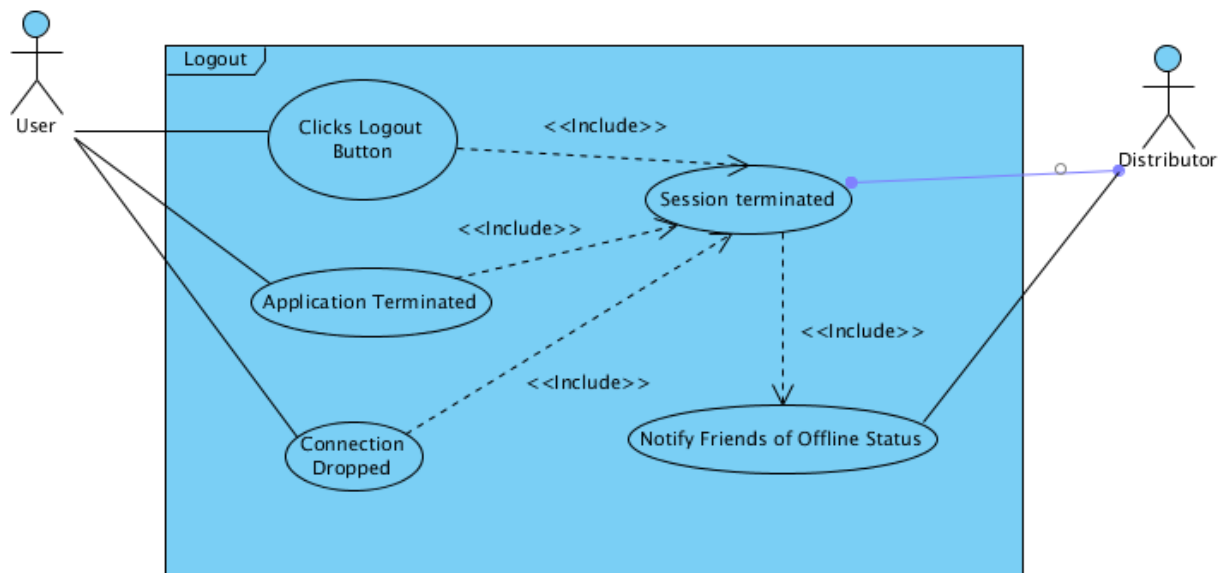
6. REQUIREMENTS

6.1 USE CASE DIAGRAMS

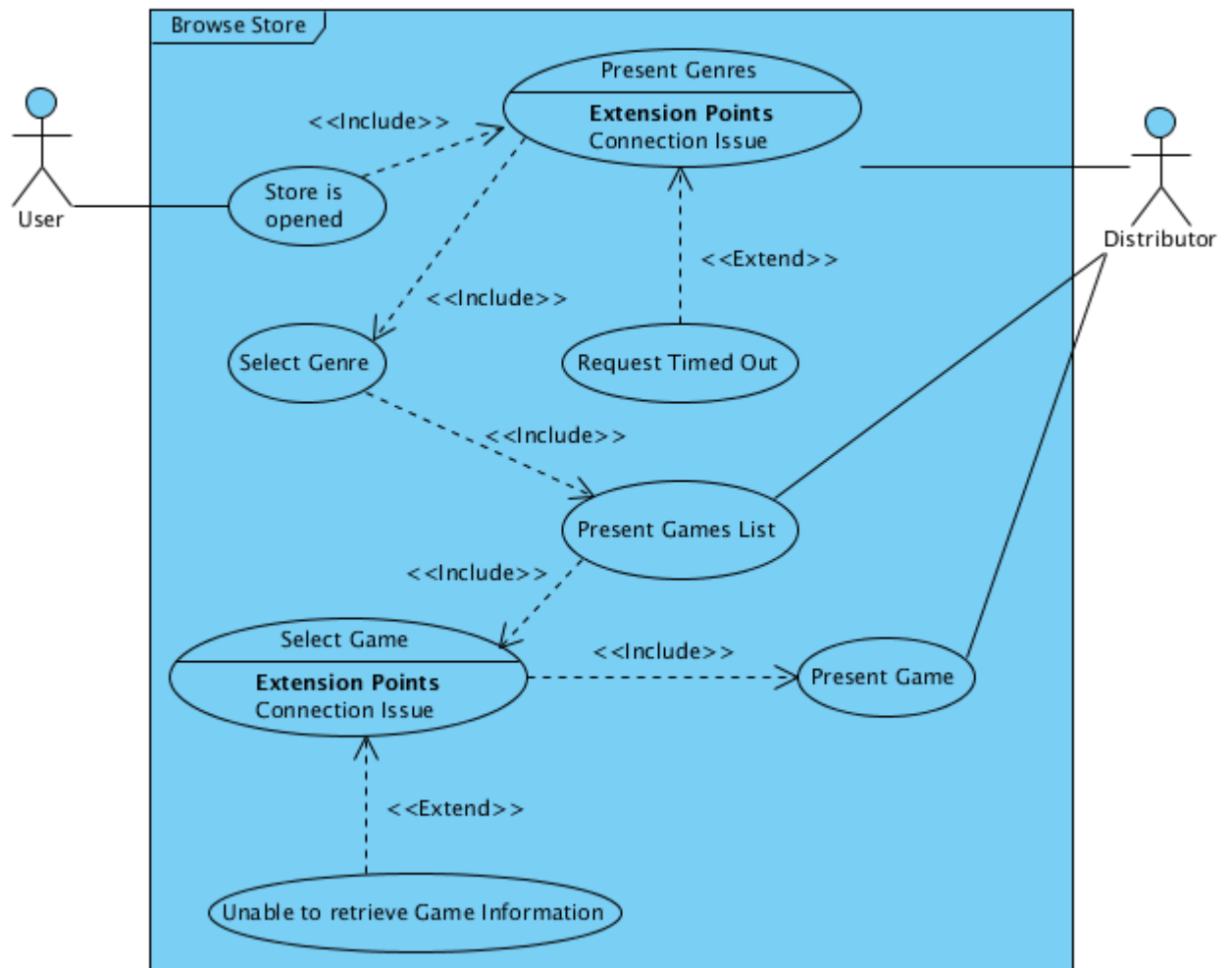
6.1.1 LOGIN



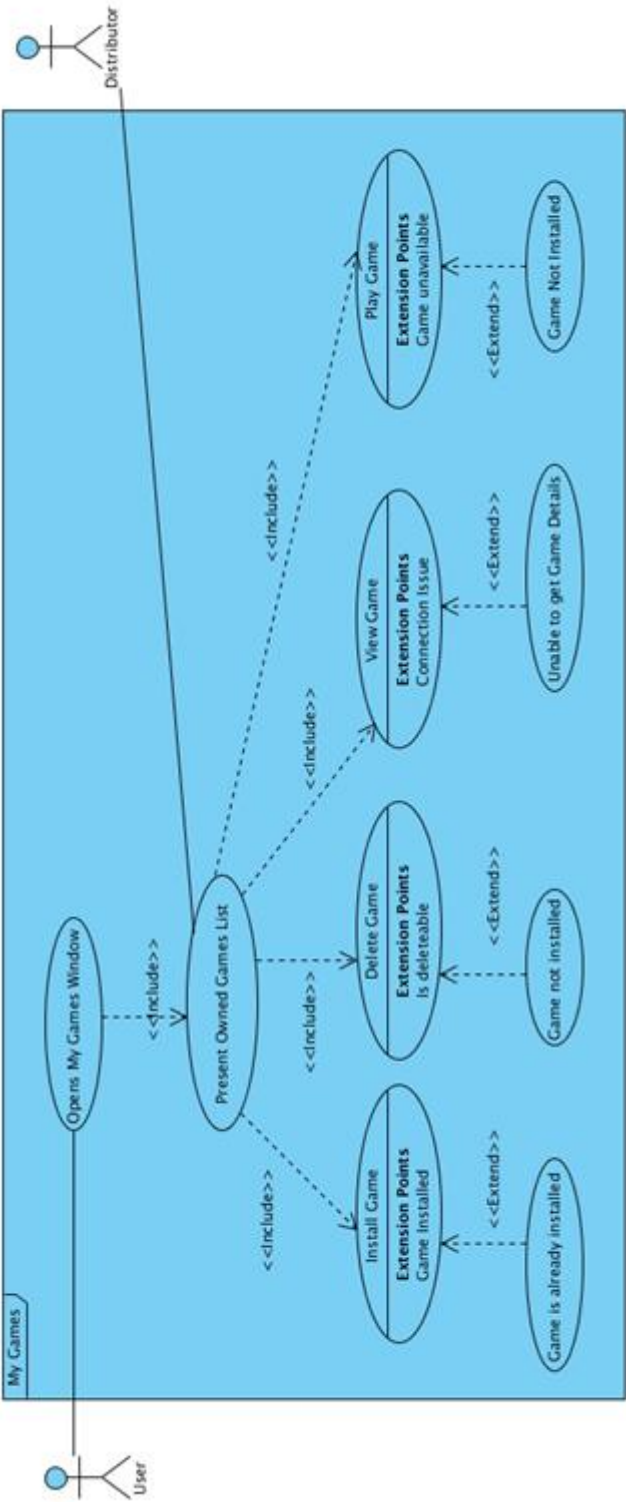
6.1.2 LOGOUT



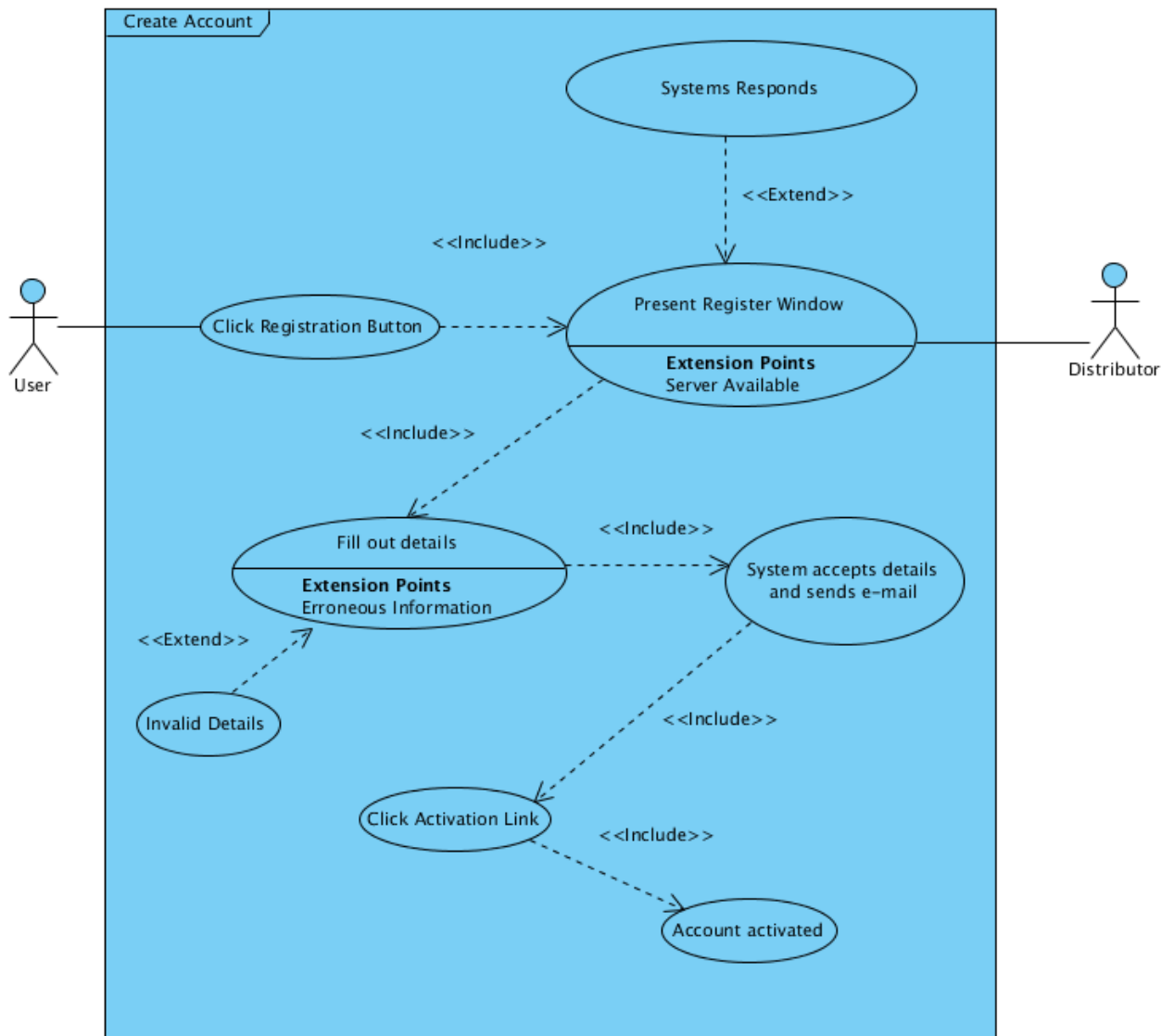
6.1.3 BROWSE STORE



6.1.4 MY GAMES



6.1.5 CREATE ACCOUNT



6.2 STRUCTURED USE CASE DESCRIPTIONS

Use Case	Create Account	
Goal	Customer will enter personal details Expects account created	
Scope and level		
Pre-conditions	Client is installed User has email address Internet access	
Success End Condition	Account created. Success dialog	
Failed End Condition	Account not created	
Primary Actor	Customer	
Secondary Actor	Email provider	
Trigger	User wants account	
Description	Step	Action
	1	User opens registration dialog
	2	User fills out details+ email
	3	Distributor accepts details and sends activation email

	4	User clicks activation link
	5	Distributor marks account active
Extensions	Step	Result
	2a	Invalid details
	2a1	Highlight erroneous details
	3a1	Email not received
Performance	5 minutes registration 24 hour activation	
Frequency	10000 / day	
Channel	Network Helper	
Open issues	Impersonation	
Due-Date	Initial release	
Super-Ordinate	Create customer relationship	

6.2.1 GENERAL USE CASE DESCRIPTIONS

#	Use Case	Description
1	Login	User completes login form. Details sent to the server for authorisation. System responds on successful login. Store homepage page loaded.
2	Logout	User closes application or clicks logout. Message sent to server. Servers invalidates session, server informs friends (clients) of offline status
3	Browse Store	User opens store Distributor sends games genres Users selects genre selection sent to the server Server receives and sends list of games in that genre
4	My Games	User opens my games tab. Requests sent to distributor for bought games. Games sent to client. Games shown in window
5	My Community	User opens community tab. Request sent to server for profile details and friend activity details. Response sent to the client and displayed in the window.

6.2.2 STORE USE CASES DESCRIPTIONS

#	Use Case	Description
3.1	Select Genre	User will select genre from list. Request sent to server for recent or popular games in that genre. First 10 Games shown, with pagination.
3.2	Browse Pages	User can go back and forward through 10 games per paginated games list. Each back and forth communication requests the system for the next/previous 10.
3.3	Select Game	User clicks on a game. Request sent to server for extended details. Server replies with full game details displayed in window.
3.4	Add to Cart	User selects “add to cart” option on individual game page. System saves cart state. System notifies user of add to cart success. User continues browsing.
3.5	Checkout	User clicks checkout. Checkout page shown, populated by server cart response.
3.6	Purchase	User inputs credit card information. User clicks confirm. Information sent to server. Server authorises with external verification system. Response sent to client.
3.7	Search	User inputs game name. Request sent to server. Paginated results shown with closest title first.

6.2.3 MY GAMES USE CASE DESCRIPTIONS

#	Use Case	Description
4.1	Browse Owned Games	User shown list of owned games (paginated). Installed games are bolded.
4.2	Install Game	User clicks install button next to game title. Request sent to server for confirmation of ownership. Server sends response and location of external game files. Installation status shown to user.
4.3	Delete Game	User selects delete game option on installed game. Confirmation dialog shown. Game deleted from hard-drive.
4.4	View Game Info	User clicks game title. Request sent to server for extended details. Details shown.
4.4	Start Game	User clicks launch Game. Game executable executed.

6.2.4 MY COMMUNITY USE CASE DESCRIPTIONS

#	Use Case	Description
5.1	Edit Profile	User clicks edit profile. User can change username, avatar, password and personal details.
5.2	Look at friend profile	User clicks on friend. Request sent to server for friend's profile. Public profile page sent to and displayed on client.
5.3	Add friend	User clicks "add friend". Presented with form for friend's username. User enters name, request sent to server. Server finds friend and notifies them of request.
5.4	Accept friend	Request appears on Community page. User clicks accept or deny.

6.3 DETAILED USE CASE DESCRIPTIONS

Use Case Description - Login

Actor Action	System Response
1. Selects "Login"	2. Login form presented.
3. User enters user name and password. Selects "confirm".	4. Verifies validity of user, sends confirmation. Re-directs to store page, updates user status, and notifies friends.

Use Case Description - Logout

Actor Action	System Response
1. Selects "Logout" or client termination event occurs.	2. Session invalidated. Updates user status, notifies friends. Presents login window

Use Case Description - Browse Games

Actor Action	System Response
1. Selects "Next"/"Previous" Page.	2. Displays the desired page depending on the genre previously selected & current page.

Use Case Description - View Owned Games

Actor Action	System Response
1. Selects "My Games"	2. Collects user's bought games from database & Displays list of games in window.

Use Case Description - Select Genre

Actor Action	System Response
1. Selects "Genre"	2. Displays genre list
3. Selects genre on the list	4. Displays list of games related to the genre

Use Case Description - Checkout

Actor Action	System Response
1. Selects "Checkout"	2. Add all price of games in cart & Display purchase page in window

Use Case Description - Purchase

Actor Action	System Response
1. Selects type of payment	2. Displays relevant payment page
3. Fills in payment information, submits request	4. Verifies validity of payment information, updates new purchase to database, displays bought games

Use Case Description - Search

Actor Action	System Response
1. Types in key words in searching box	2. Searches relevant information in database, Displays paginated results in window

Use Case Description - Install Game

Actor Action	System Response
1. Select "Install game"	2. Displays confirmation message in window
3. Confirms the confirmation message	4. Sends response & location of external game files, Displays installation status
5. Select "Finish" to finish installation	

Use Case Description - Delete Game

Actor Action	System Response
1. Select "Delete"	2. Displays confirmation dialog
3. Select "Yes"	4. Delete game from hard drive & display completion message
5. Select "OK"	

Use Case Description - View Game info

Actor Action	System Response
1. Clicks game title	2. Search data in database, displays game information in window

Use Case Description - Star Game

Actor Action	System Response
1. Selects "Launch"	2. Execute game

Use Case Description - Edit Profile

Actor Action	System Response
1. Clicks "Edit"	2. Searches user's information from database & Displays profile page
3. Fills in new username, avatar, password, personal details & Select "Submit"	4. Verifies validity of new data & Updates in database

Use Case Description - Add Friend

Actor Action	System Response
1. Select "Add Friend"	2. Displays form for friend's username
3. Enters desired username	4. Verifies validity of username, notifies user

Use Case Description - Accept Friend

Actor Action	System Response
1. None	2. Displays request adding friend
3. Selects "Accept"	4. Records new friend in database, Add in new friend in Friend List, Notifies requester

Use Case Description - View Friend Profile

Actor Action	System Response
1. Select "Profile"	2. Search friend profile in database, Displays profile page

Use Case Description - Deny Friend Request

Actor Action	System Response
1. None	2. Displays request adding friend
3. User selects "Deny".	4. Message of rejected request sent to requester. Requester is not added to user's friend list.

Use Case Description - Delete Friend

Actor Action	System Response
1. Select "Delete" tag around friend name	2. Displays confirmation dialog
3. Select "Yes"	4. Updates friend list in database, Displays completion message

Use Case Description - Add To Cart

Actor Action	System Response
1. Select "Add To Cart" on individual game page	2. Updates cart state, Displays notification to add to cart successful

Use Case Description - Launch Application

Actor Action	System Response
1. User runs client executable.	2. Client launches, user is presented with login window.

6.4 NON-FUNCTIONAL REQUIREMENTS

As part of the requirements analysis and the support of quality attributes such as extensibility, modifiability and reusability; otherwise known as the 'ilities', we identified the following .

Identification of quality attributes in the requirements phase

Quality covers a broad range of characteristics, which is one of the first challenges to overcome. Usability, performance and security are common examples of capabilities that are difficult to express as functionality, and there are many more areas to consider. We need a taxonomy that we can expect to cover the breadth of quality issues we may run into. We identified the following taxonomies from Managing Software Requirements: A Use Case Approach (2nd Edition), Addison Wesley where he compares three different points of view.

Comparison between several broad Quality Taxonomies

McCall et. al.	RADC	Wieggers
Correctness	Correctness	
Reliability	Reliability	Reliability
Usability	Usability	Usability
Integrity	Integrity	Integrity
Efficiency	Efficiency	Efficiency
Portability	Portability	Portability
Reusability	Reusability	Reusability
Interoperability	Interoperability	Interoperability
Maintainability	Maintainability	Maintainability
Flexibility	Flexibility	Flexibility
Testability	Verifiability	Testability
	Survivability	Robustness
	Expandability	
		Installability
		Safety
		Availability

Given these taxonomies we the forth coming list of quality attributes for our application.

6.4.1 THE 'ILITIES'

Performance Criteria

- (Distributor) Download speeds should match popular download systems e.g. iTunes, XBL
- (Client) Community page should be dynamically loaded, allowing for continuously fresh info
- (Client) Client should load and login in under 15 seconds
- (Client) Client should be reasonably bug free and with minimal crashes with quality error handling.

Anticipated Volume

- (Distributor) System should be able to handle transfer of 10s of terabytes of data per day with provisions for expansion, catering to special occasions/sudden rises.
- (Distributor) System should be able to handle 10,000+ concurrent users, registrations and their "web presence" i.e. achievements

Security Considerations

- (Distributor) System should be able to offer an effective security code at the end of registration, when purchasing new games. (captcha)
- (Distributor) Bank details should be secure and verifiable.
- (Client) "Remember me" functionality should be reasonably secure.
- (Distributor) A Game unlock key should not be vulnerable/exploitable.
- (Client) Network communication should be encrypted using SSL technology.

Usability Requirements

- Must be approachable for a demographic of 15 to 50, "If you can buy from Amazon you can buy from us".
- Multi-lingual, no hard-coded text
- Support text-to-speech software
- Simplistic, minimalist look and feel.

Communication Requirements

- Interoperable communication language to enable core fundamental support for multiple systems deployment. Such as RPC, XML, Serialized object notation
- Network communication should be encrypted using SSL technology.
- Support for 802.3 & 802.11 standards.

Audit Requirements

- Distributor should be able to track purchases
- Distributor should in act tax by region
- Distributor should in act age gating by region

Availability Requirements

- Distributor should guarantee 99% Uptime
- Recovery from unplanned downtime should be minimal

Backup Requirements

- Client should be able to backup games
- Distributor should have database backup solution.

Extensibility Requirements

- Ability to add features, and carry-forward of customizations at next major version upgrade, maintaining quality parity with similar systems.

Interoperability Requirements

- User should be able to logon independent of system

Maintainability Requirements

- System should not need fundamental code change in less than 5 years.

Modifiability Requirements

- Ability to expand to sell different types of content in the future

Platform Requirements

- Client should be installable on many platforms : Windows, Mac, Linux

Portability Requirements

- User should be easily able to migrate from one pc to another.

Quality Requirements

- Distributor should provide weekly updates to the client to address major faults, bugs and usability issues
- Major functionality improvements at least bi-annually.

Reliability Requirements

- Mean time between failures of one month

Resource Constraint Requirements

- Client should be designed to work on resource limited machines in respect to processor, ram and disk space.

Scalability Requirements

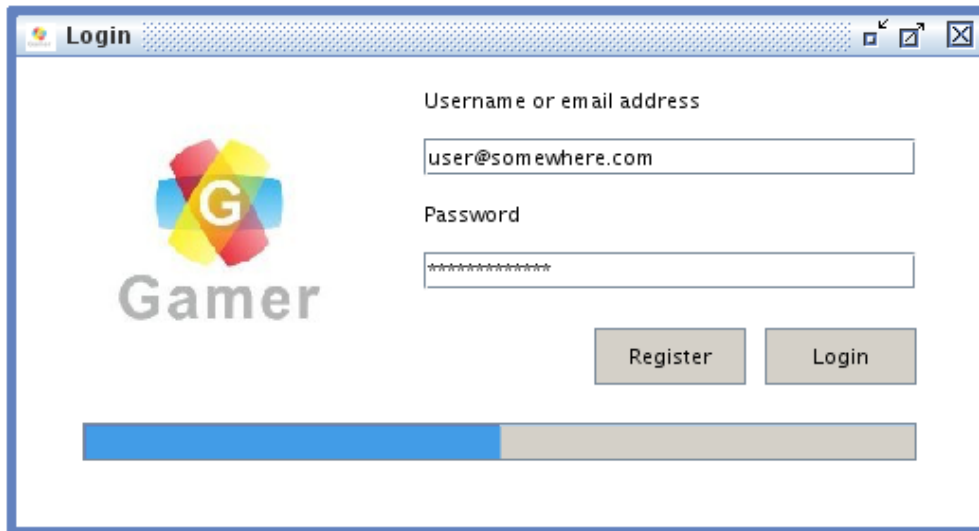
- System should be scalable to millions of concurrent users.

Supportability Requirements

- Client should offer a way to reach technical support and address problems such as installation and configuration.

6.5 SCREENSHOTS / MOCK-UPS / REPORT FORMATS

6.5.1 LOGIN VIEW



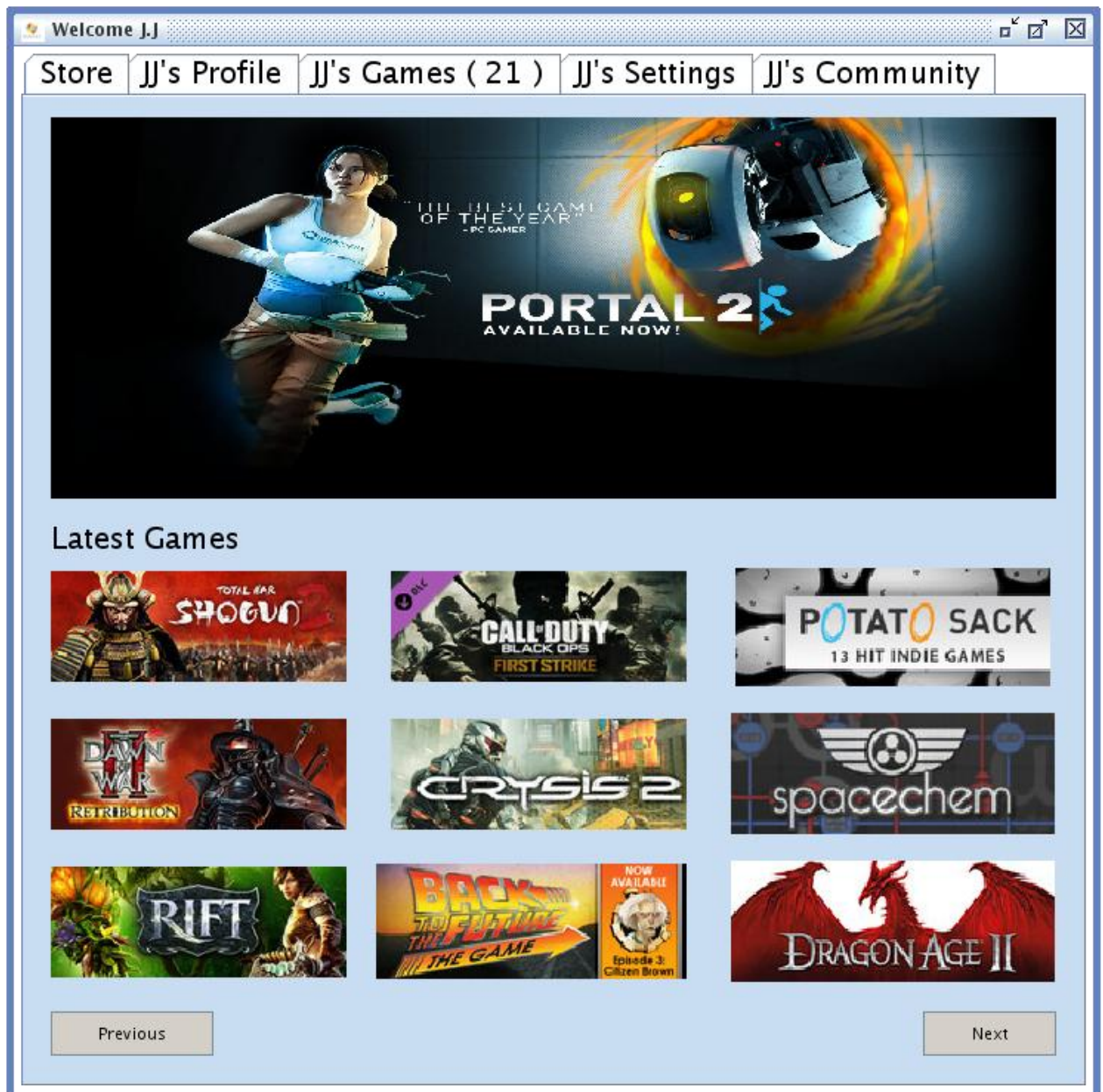
A screenshot of a web browser window titled "Login". The window has a blue border and standard window controls (minimize, maximize, close) in the top right corner. On the left side, there is a logo consisting of a stylized "G" made of four overlapping colored squares (red, yellow, blue, green) with the word "Gamer" in a grey sans-serif font below it. To the right of the logo, there are two text input fields. The first is labeled "Username or email address" and contains the text "user@somewhere.com". The second is labeled "Password" and contains a series of asterisks "*****". Below these fields are two buttons: "Register" and "Login". At the bottom of the window, there is a horizontal progress bar that is partially filled with blue.

6.5.2 REGISTER VIEW



A screenshot of a web browser window titled "Register". The window has a blue border and standard window controls in the top right corner. On the left side, there are several form fields with labels to their left: "Email address" (containing "user@somewhere.com"), "Password" (containing "*****"), "Confirm password" (containing "*****"), "First name" (containing "JJ"), "Surname" (containing "Collins"), "Date of birth" (three separate dropdown menus), and "Country" (a dropdown menu). To the right of these fields is a logo consisting of a stylized "G" made of four overlapping colored squares (red, yellow, blue, green) with the word "Gamer" in grey and "AVATAR" in red below it. At the bottom left, there is a text label "Email verification required". At the bottom right, there are two buttons: "Cancel" and "Register".

6.5.3 MAIN APPLICATION VIEW



7. ANALYSIS DIAGRAMS

7.1 IDENTIFYING THE CANDIDATE CLASSES

7.1.1 NOUN IDENTIFICATION TECHNIQUE

We identified the candidate classes through “Noun Identification” on the use case descriptions to obtain the key domain abstractions within the Games Digital Distribution system.

This was achieved by identifying and extracting the nouns found in our use cases, of which the results are below.

Example:

Actor Action	System Response
1. Selects “ <u>Login</u> ”	2. <u>Login form</u> presented.
3. <u>User</u> enters <u>user name</u> and <u>password</u> . Selects “ <u>confirm</u> ”.	4. Verifies validity of <u>user</u> , sends <u>confirmation</u> . Re-directs to <u>store page</u> , updates <u>user status</u> and notifies <u>friends</u> .

7.1.2 SAMPLE NOUNS IDENTIFIED

Customer	Game
Account	Collection
Dialog--abstract(success fill, complete)	Genre
Distributor	Window
Email	Cart
Network-Helper	Checkout-control class
Pages--abstract(store, my games, community, etc)	Authorisation-control class
Session	Search-control class
Account-control class	

7.1.3 *EVALUATION HEURISTICS*

Is the noun:

- Ambiguous
- Redundant/ duplication of behaviour
- An event
- An attribute
- An operation
- An association

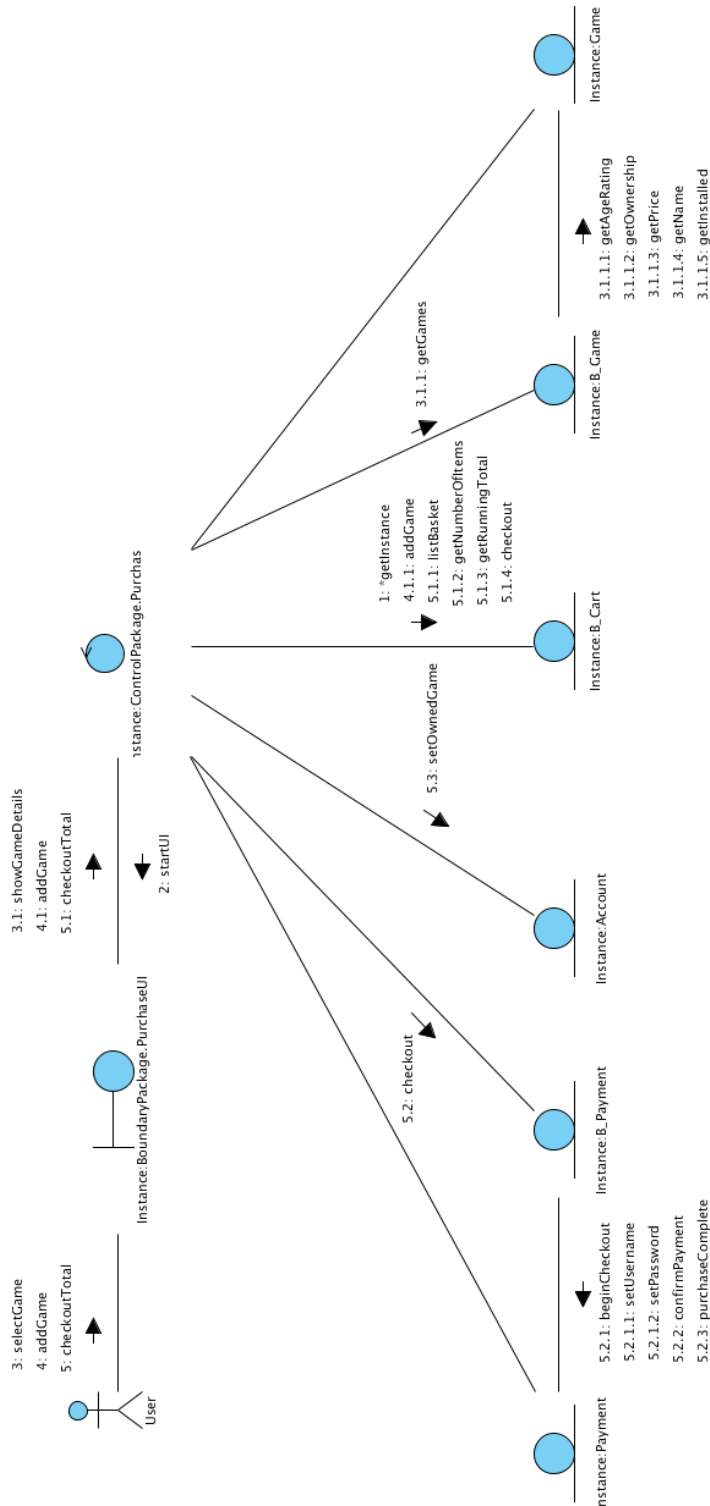
We both intelligently eliminated poor candidate classes by using these heuristics during the noun identification and after, reviewing the nouns we had identified.

7.2 SYSTEM CLASS DIAGRAM

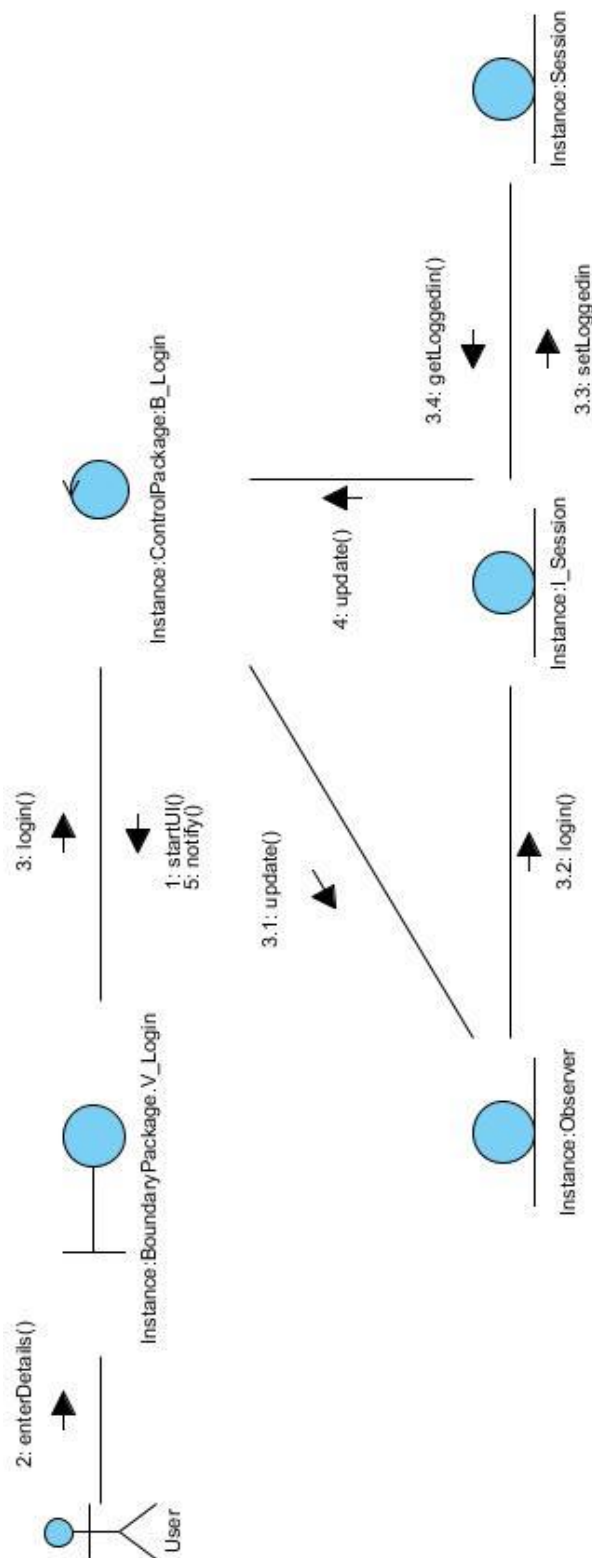


7.3 COMMUNICATION DIAGRAMS

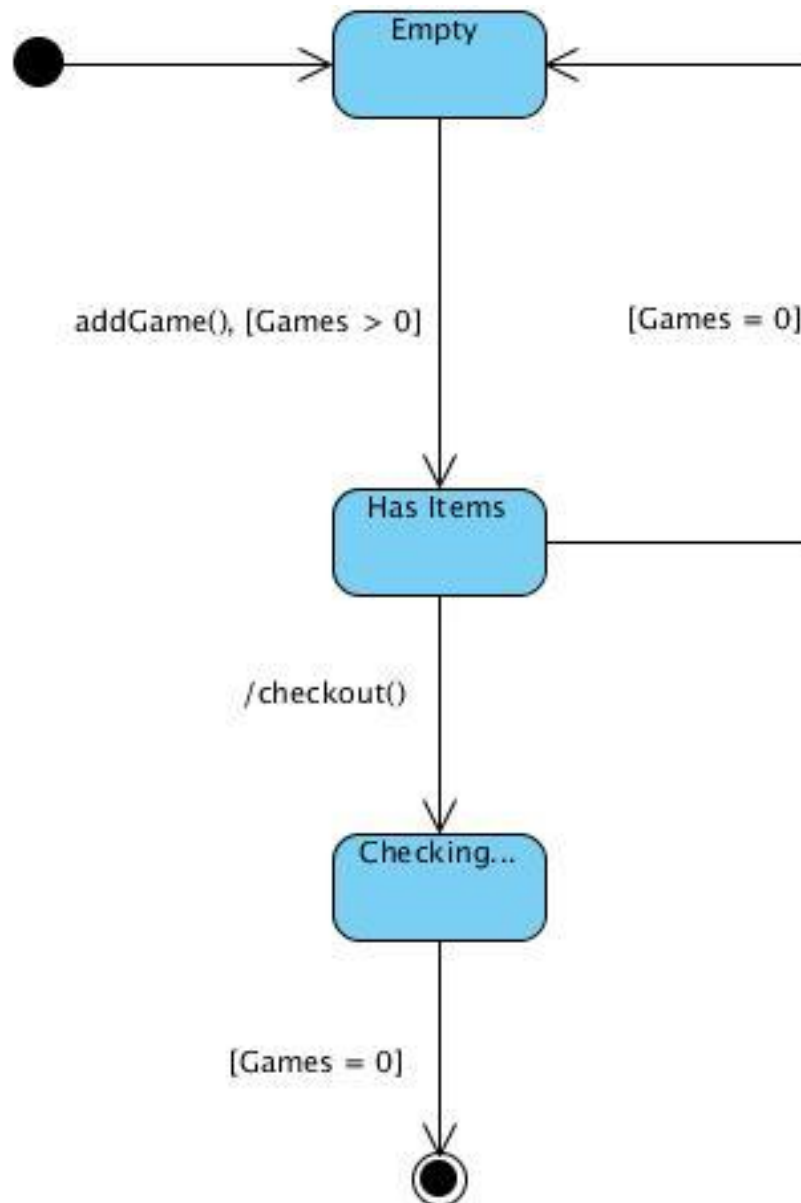
7.3.1 PURCHASE COMMUNICATION DIAGRAM



7.3.2 LOGIN COMMUNICATION DIAGRAM

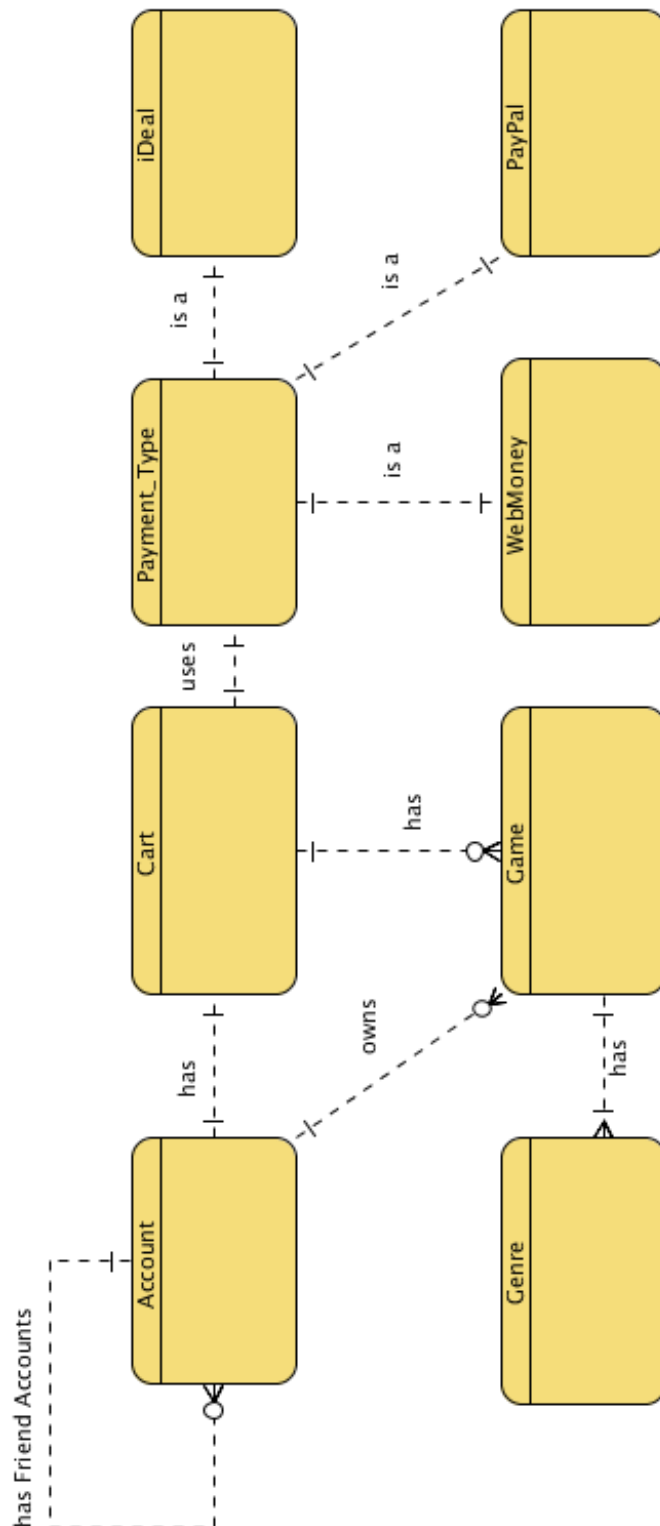


7.4 STATE CHARTS

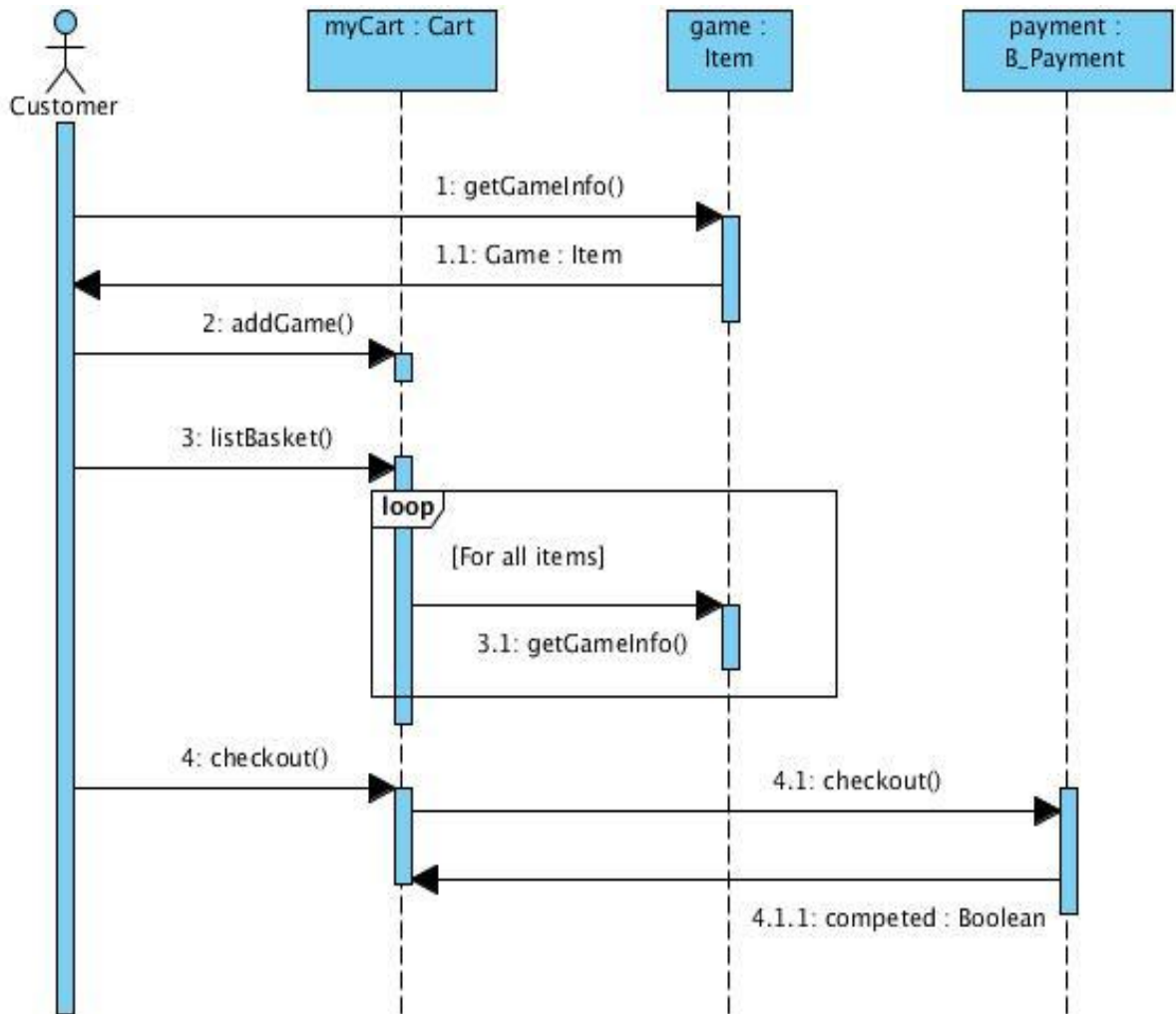


The Final State here does mean the object no longer exists. It always exists in the system, even after a user has finished their purchase.

7.5 ENTITY RELATIONSHIP DIAGRAM



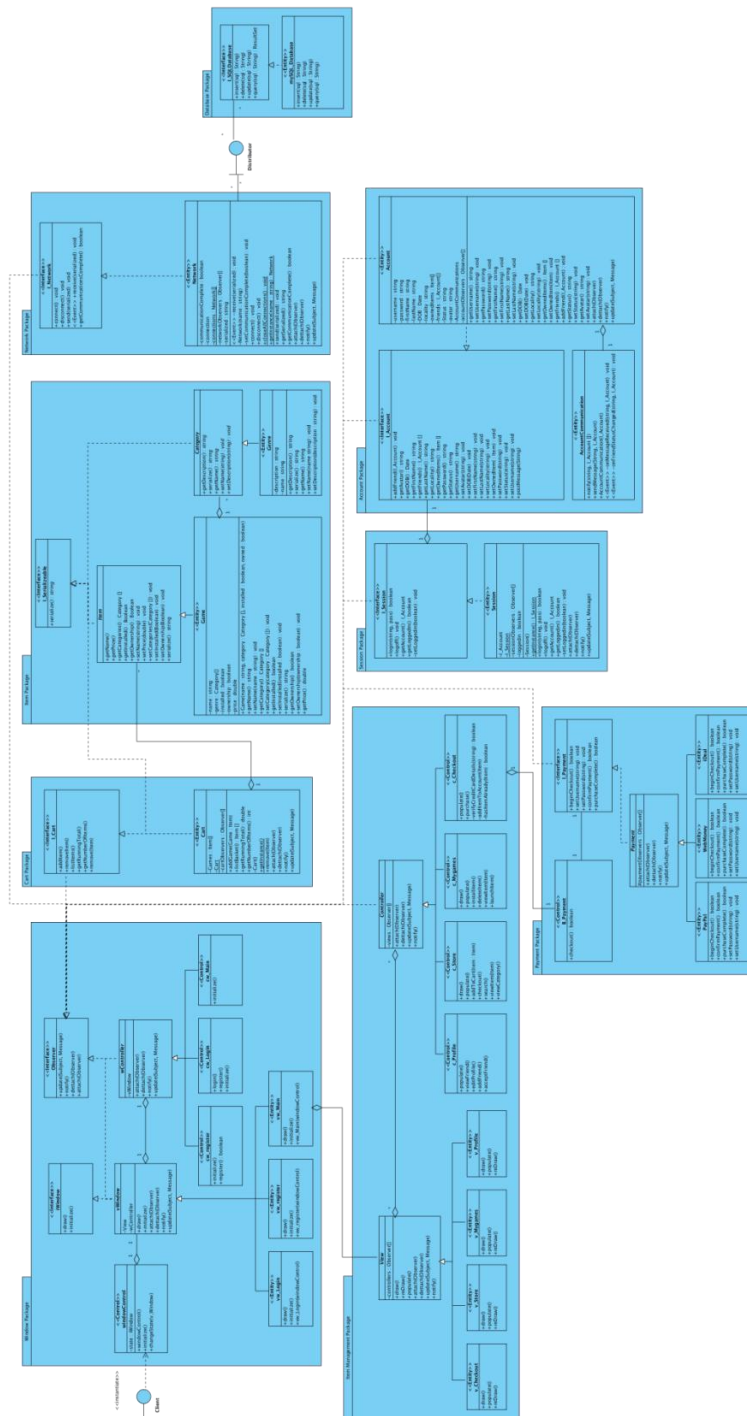
7.6 SEQUENCE DIAGRAM



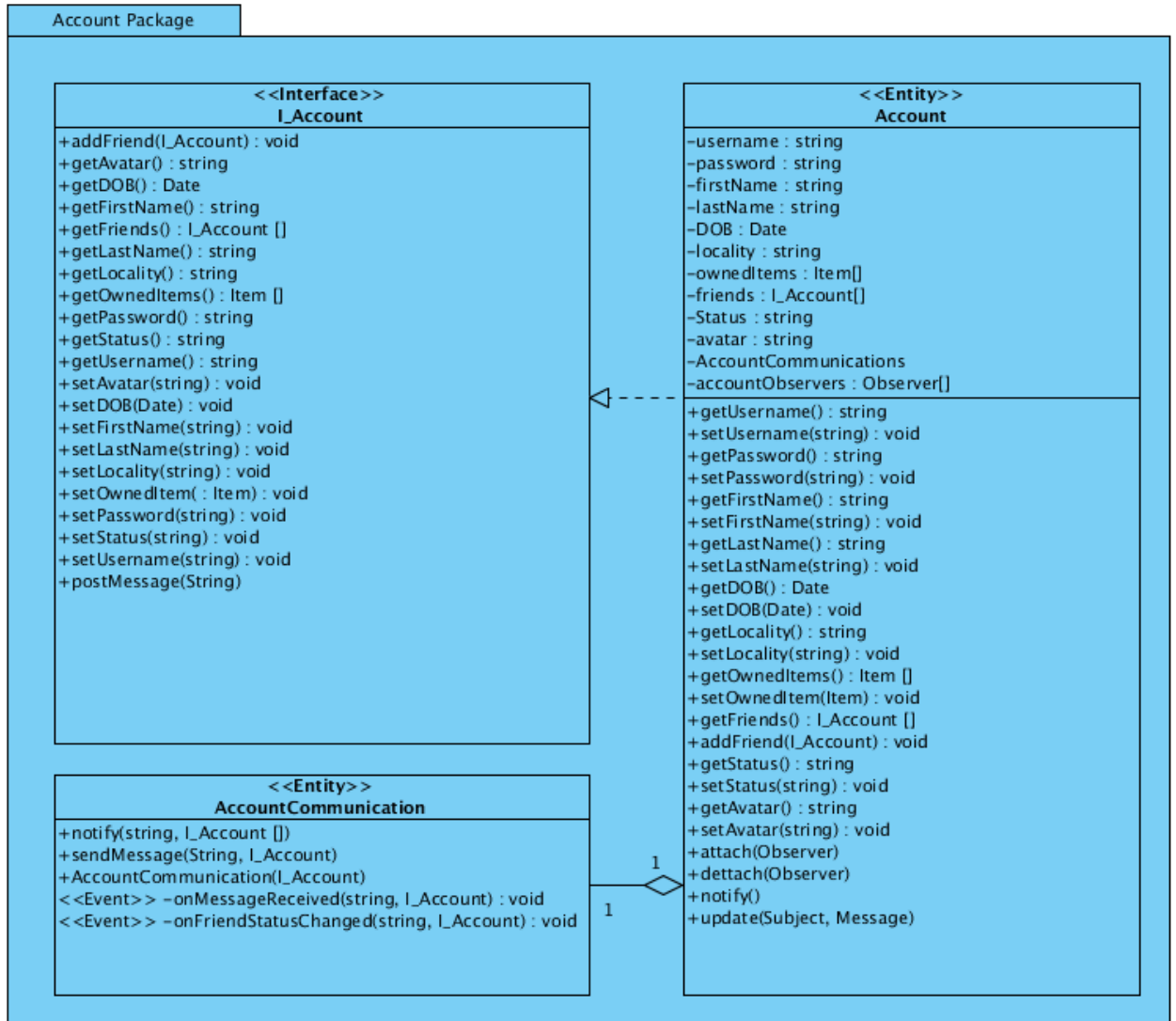
8. DESIGN

Design discussed below in Section 9.

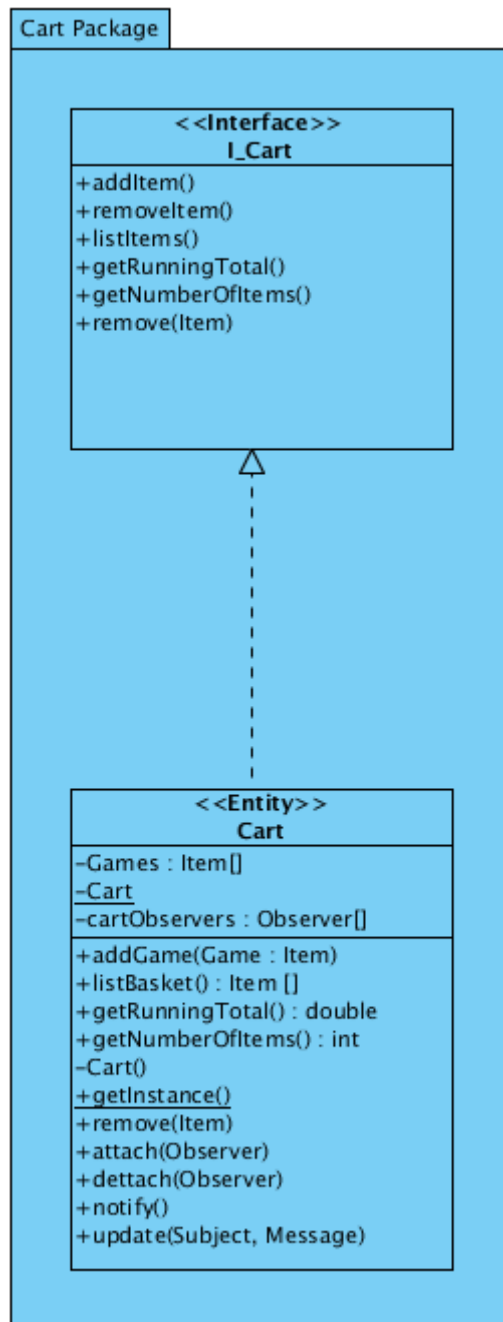
8.1 SYSTEM CLASS DIAGRAM



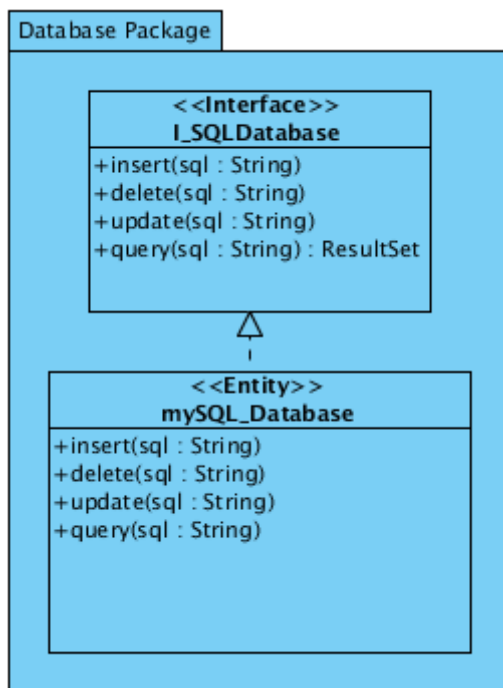
8.2 ACCOUNT PACKAGE



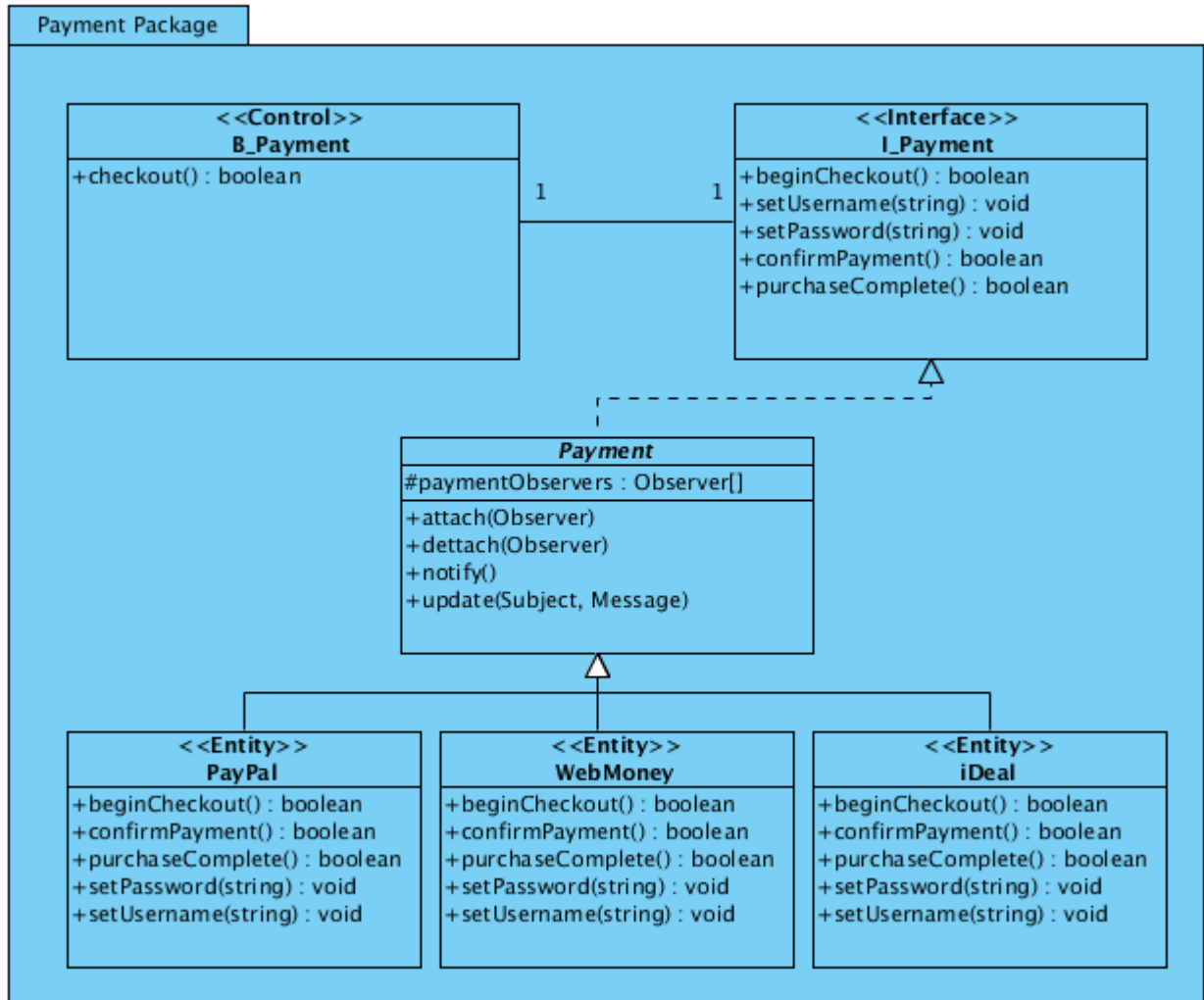
8.3 CART PACKAGE



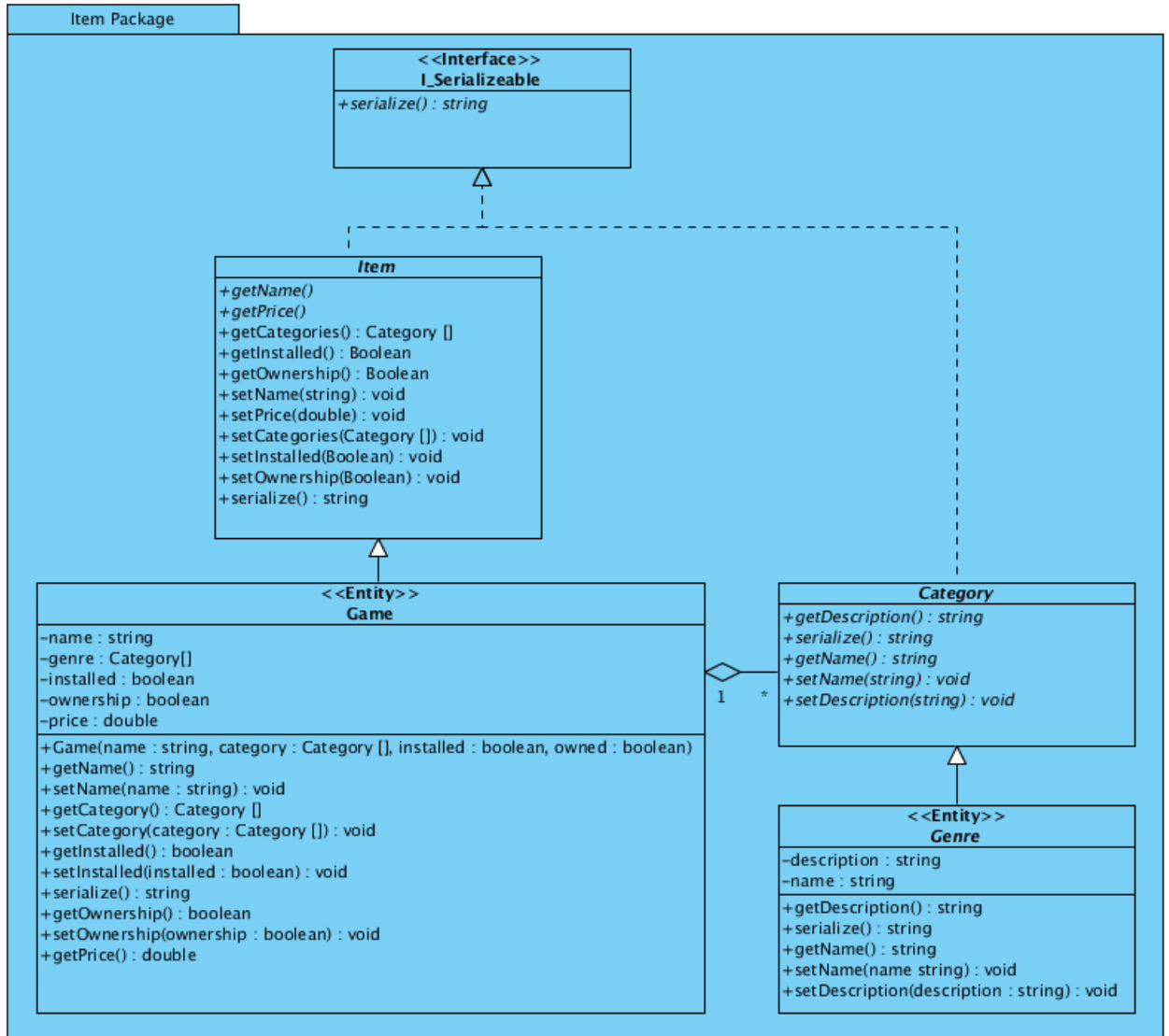
8.4 DATABASE PACKAGE



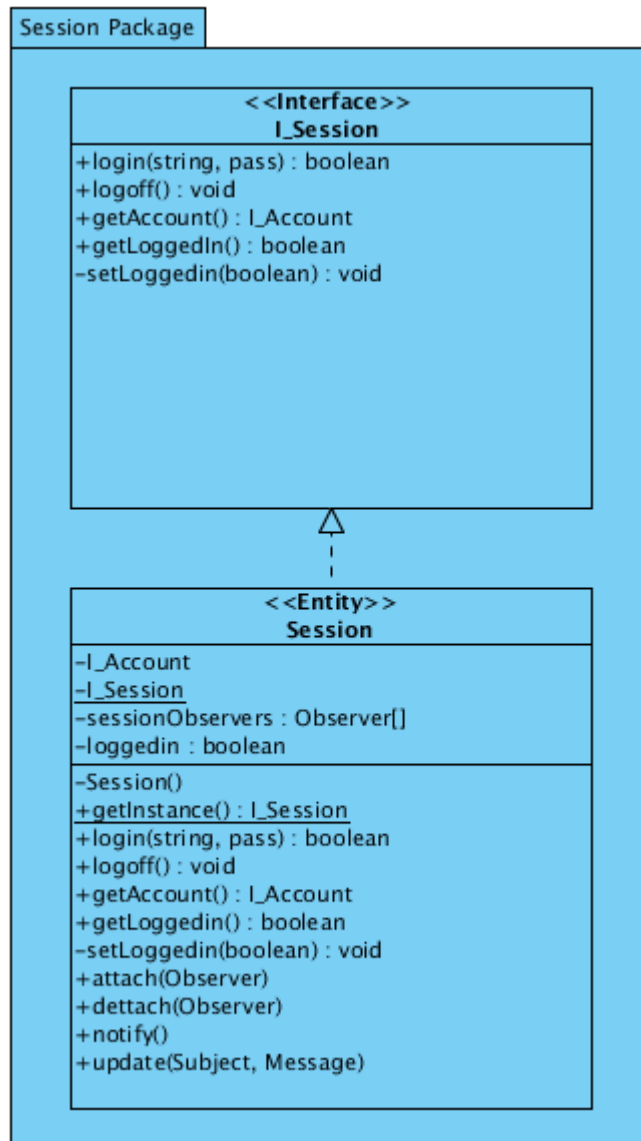
8.5 PAYMENT PACKAGE



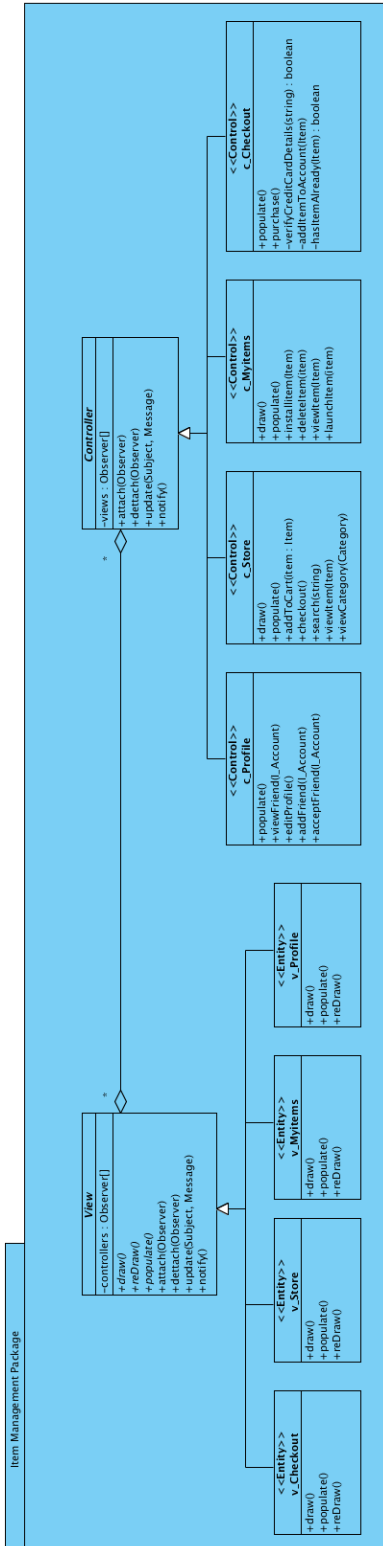
8.6 ITEM PACKAGE



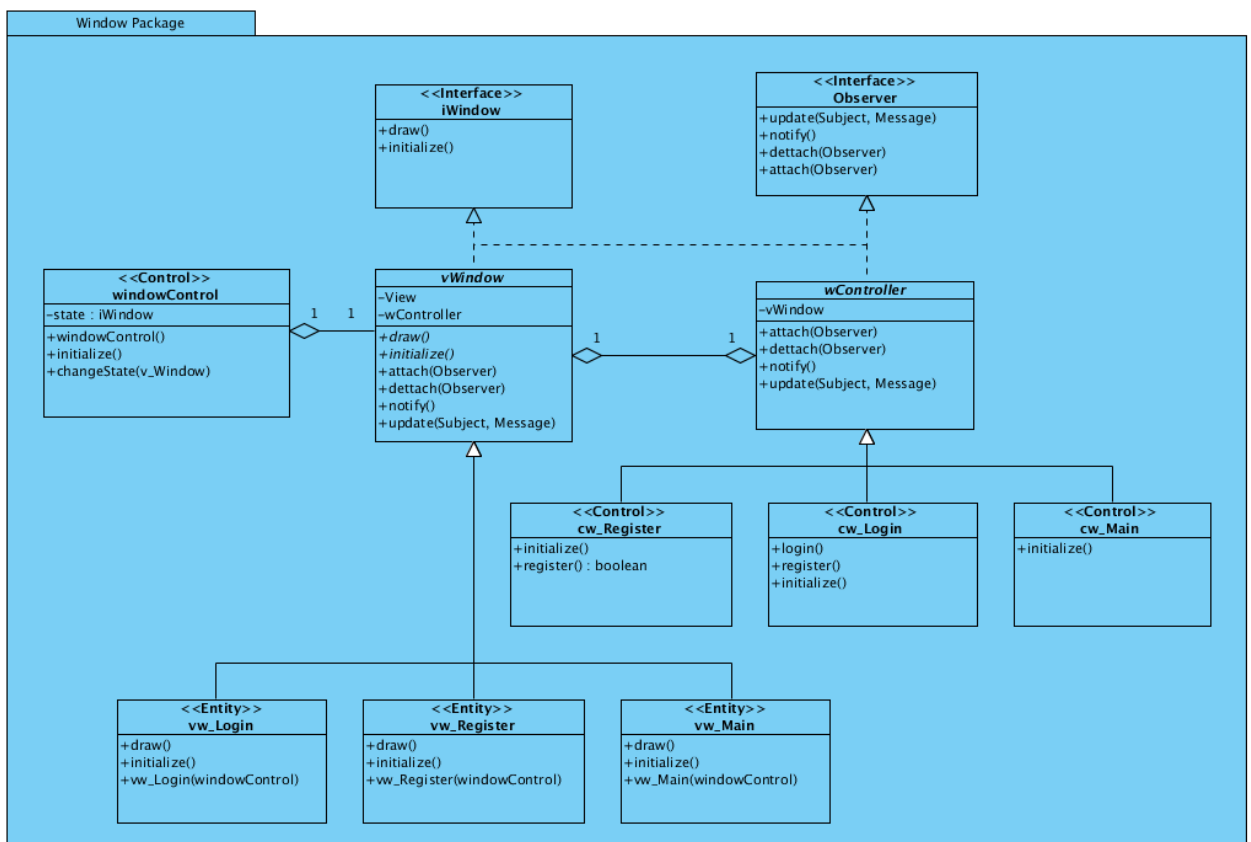
8.7 SESSION PACKAGE



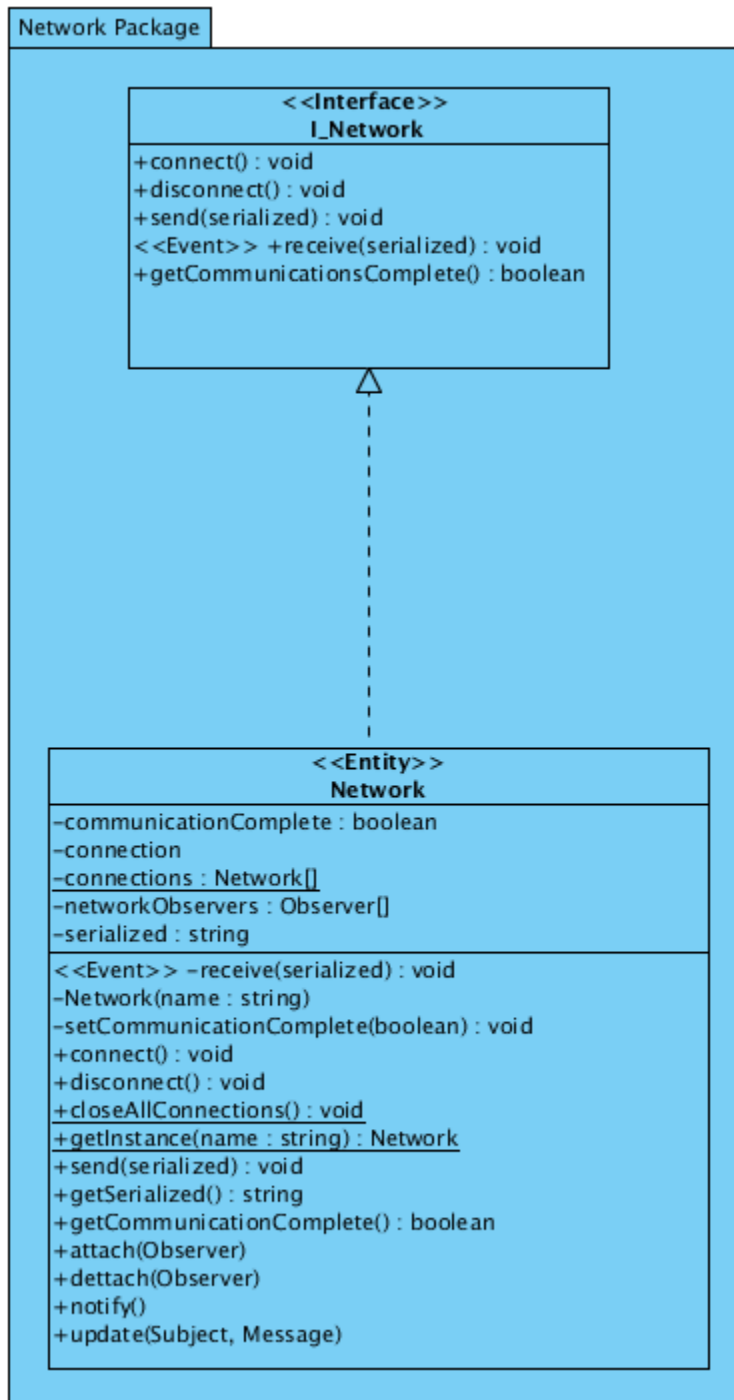
8.8 ITEM MANAGEMENT PACKAGE



8.9 WINDOW MANAGEMENT PACKAGE



8.10 NETWORK PACKAGE



9. DESIGN DISCUSSION

9.1 DESIGN PATTERNS USED

9.1.1 SINGLETON PATTERN

The singleton design pattern is used extensively in the architecture design to ensure more efficient use of objects within the system. Rather than recreating objects as needed in different scopes within the program or passing references to an object; this was replaced with the singleton design pattern.

The singleton design pattern ensures global state into the application and avoids state anomalies between objects instances. The singleton design pattern in its simplest form ensures a single instance of an object. In terms of our system where only a single user is logged into the client application at a time and the multitude of different objects scopes, it is important to have a single instance of the session object that can be used at any scope within the application.

Furthermore controlled access to an encapsulated sole instance restricts how and when the client accesses it, it avoids the need for global variables that pollute the name space. Modifying the singleton pattern and extending from it enforces a single instance to the derived class, and with some slight modification to the *getInstance* grant access operation, it can manage a collection of instances.

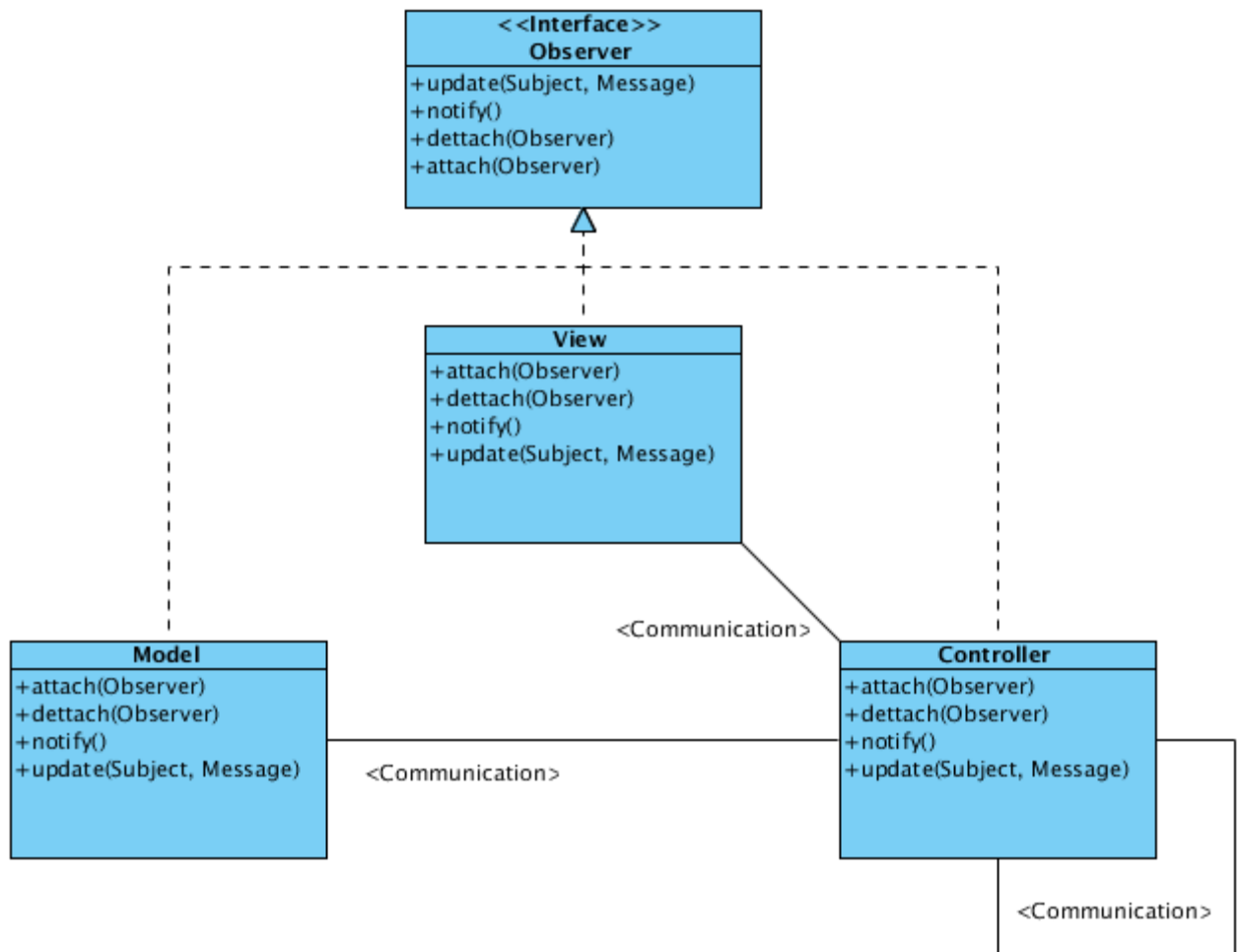
Singleton
<u>-singleton : Singleton</u>
<u>-Singleton()</u>
<u>+getInstance() : Singleton</u>

We decided to use the singleton pattern for our Session package, Network Package and the Cart package. As these three packages would be used extensively throughout the application; this was preferable to passing references to instances or polluting the global namespace. The initial instantiation of the cart, session and network instances would be domain at predefined logical points within the application and there on be used at any state within the application lifetime.

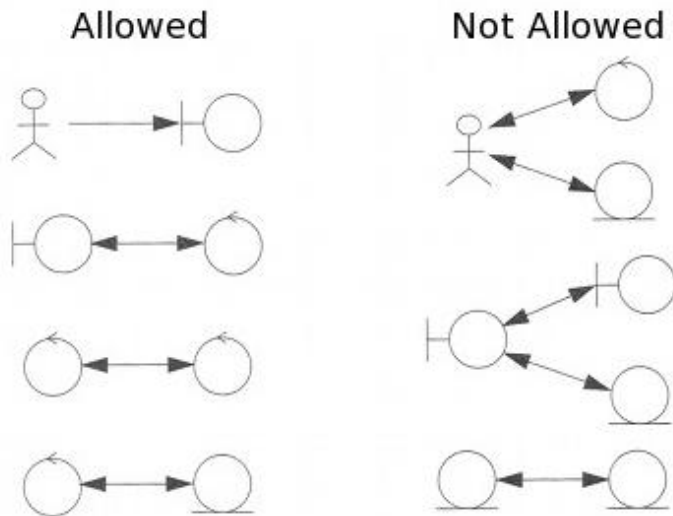
9.1.2 MODEL VIEW CONTROLLER

The Model View Controller pattern separates an application into 3 major parts.

- Model - Adapters that provides interface to the data, such as a database.
- Views - Show the data to the user
- Controllers - implement the business logic and modify the model through the model adapters.



In the original Smalltalk MVC pattern the following interactions were described as the legal communication between the entities.



However since its inception interactions between the model and the view have since been added. A change to the model can invoke a change on the view. Typically a change in the view however would invoke a change through the controller that would apply business rules and ensure the integrity of the data before updating the model, then the controller would invoke changes to any other views in the system.

In our system design class diagram we refactored our analysis diagram to include the MVC pattern extensively. The main window package has different business contexts, register login and the main window application. We decided to use the MVC pattern here with observer along with the window Controller which introduced the state pattern to invoke changes in state thereby allowing seamless changes between the states and providing the possibly for different views of the system interface for example applying themes to the application.

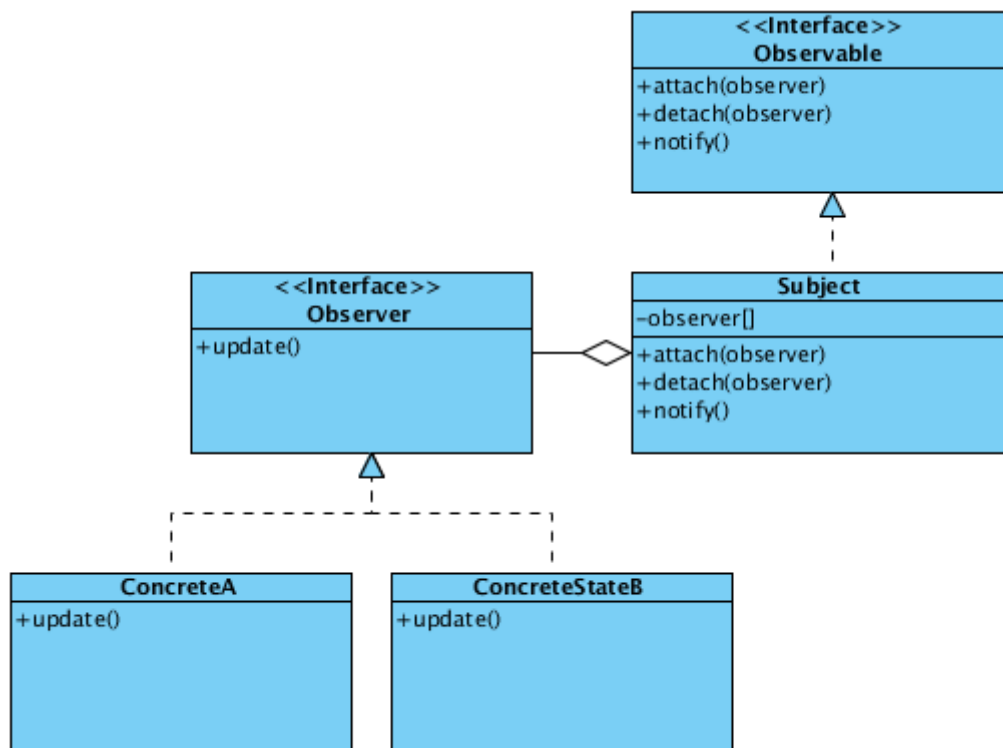
The second inception of the MVC pattern was to be used in the main window component based design. The main window is modelled as a container with sub pages or components. These sub pages were also modelled using MVC to allow message passing between these component views and the controllers thereby allowing full transparency without violating encapsulation, maintaining high cohesion while minimizing dependencies through the use of the observers.

9.1.3 OBSERVER PATTERN

The observer pattern defines a one-to-many dependency between objects so that when one object changes state all of its dependants are notified about the change in state and update automatically.

Creating highly coupled objects reduces their reusability, so observer endeavours to maintain the consistency among the collection of these cooperating classes.

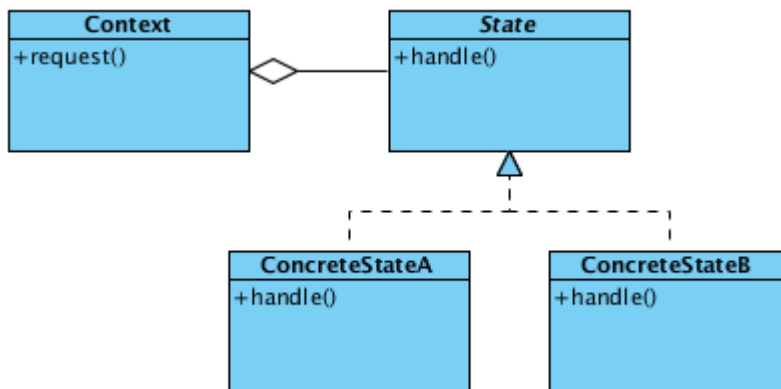
The subject or the object that has changed state invokes changes on all the observers attached to it using the update method realised by the observer interface. This is done through the notify method in the observable interface.



As part of the community design, Observer in relation to Model View Controller would be used as a way to notify and update each client's views. In this case the database on the distributors system. A model adapter bound to the data source would receive an update via a notification as a result to the change in the model. Model changes would be propagated via notifications to a change to a view and the update operation on the model.

9.1.4 STATE PATTERN

The state pattern allows an object to alter its behaviour when it's internal state changes. The aggregated type being the abstract type of the different states declares numerous abstract methods which must be implemented in each of the subclasses. These subclasses depending on their state carry out the desired operation depending on their internal state therein producing different behaviours at run time.

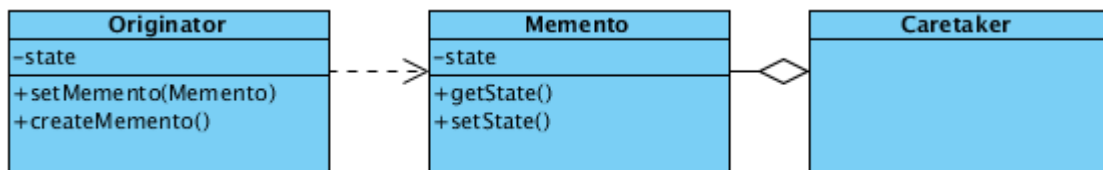


During our design we noticed that the window manager opened different windows depending on the certain conditions within the start-up sequence. Our initial design seemed to show allot of flow control depending on the users interaction with the application. We quickly saw that state pattern could solve this issue of flow control and the windows would invoke change in state thereby removing the need for flow control in the control class.

9.2 DESIGN PATTERNS CONSIDERED

9.2.1 MEMENTO PATTERN

Without violating encapsulation the memento pattern captures and restores an objects internal state. The memento pattern creates check points, that let user back out or undo tentative operations.



An obvious candidate for saving state and returning to a state is the memento pattern. The ability to undo or rollback and action is advantageous not only to the end user but can reduce the load on a system.

As part of the designing for concurrency and minimizing the network traffic, one of the design patterns considered was the memento pattern. By using the pattern the current state could be stored and restored later.

By example when the user is browsing the store, rather than re fetching the previous or next page the memento pattern can be used to capture and externalize the previous objects internal state thereby implementing an undo mechanism but also reducing network load and the interaction with the remote service.

The checkout system was also a candidate for this pattern as it would offer the user a rollback mechanism in the case of an error and the ability to recover from a mistake without having to redo the entire sequence of operation that led up to the undesired state.

10. DATA DICTIONARY

Name	User
Type	Actor
Description	A user can register to get an account for the application. User can view games, rate games, purchase games online by three payment ways and manage their owned games. Users can add other users into friend list to communicate and comment on games.

Name	Client
Type	Boundary class
Description	Launches the instance of the application

Name	iWindow [Window Package]
Type	Interface
Description	Provides the interface for the concrete Windows

Name	Observer [Window Package]
Type	Interface
Description	Provides the interface for MVC

Name	windowControl [Window Package]
Type	Control class
Description	Controls the state of the Client GUI Interface

Name	vWindow [Window Package]
Type	Abstract class
Description	Window can display Register window, login window or main window.

Name	wController [Window Package]
Type	Abstract class
Description	The domain logic for displaying the GUI views

Name	cw_Main
Type	Control Class
Description	Provides the container for the main child application components
Sister classes	cw_Register, cw_Login

Name	vw_Main [Window Package]
Type	Entity
Description	Provides the GUI for the cw_Main controller
Sister classes	vw_Register, vw_Login

Name	View [Item Management Package]
Type	Abstract Class
Description	Realises Observer, base class for component views

Name	Controller [Item Management Package]
Type	Abstract Class
Description	Realises Observer, provides domain logic for component views

Name	v_Checkout [Item Management Package]
Type	Entity
Description	Provides the GUI for the checkout entity
Sister class	v_Store, v_Myitems, v_Profile

Name	c_Checkout [Item Management Package]
Type	Control class
Description	Provides the domain logic for checkout entity, delegates to the payment package
Sister	c_Store, c_Myitems, c_Profile

Name	I_Cart [Cart Package]
Type	Interface
Description	Interface to the Cart

Name	Cart [Cart Package]
Type	Entity
Description	Realises Observer, stores list of items, used by Item Management, Payment & Account Packages

Name	I_Serializeable [Item Package]
Type	Interface
Description	Enforces interface for serialization in concrete components

Name	Item [Item Package]
Type	Abstract class
Description	Provides base structure for our primary domain business concern

Name	Category [Item Package]
Type	Abstract class
Description	Provides base structure for classifying our primary domain business concerns

Name	Game [Item Package]
Type	Entity
Description	Primary business domain concern

Name	Genre [Item Package]
Type	Entity
Description	Classifier of primary business domain concern

Name	I_Session [Session Package]
Type	Interface
Description	Provides the interface for the application sessions

Name	Session [Session Package]
Type	Entity
Description	Controls the authorisation and interactions with the external distributor system

Name	I_Account [Account Package]
Type	Interface
Description	Interface to the Account

Name	Account [Account Package]
Type	Entity
Description	An account holds attributes and behaviour to add friends, mange games, purchase games, change profile.

Name	AccountCommunication [Account Package]
Type	Entity
Description	Delegates the user notifications to & from the distributor system

Name	I_Network [Network Package]
Type	Interface
Description	Provides a common interface to the network throughout the system

Name	Network [Network Package]
Type	Entity
Description	Realises the iNetwork interface

Name	I_SQLDatabase [Database Package]
Type	Interface
Description	Provides interface to the external Database System

Name	mySQL_Database [Database Package]
Type	Entity
Description	Storage of Account and Game data which we interact with via an interface.

Name	B_Payment [Payment Package]
Type	Control class
Description	Controls the Payment System

Name	I_Payment [Payment Package]
Type	Interface
Description	Provides the contract for multiple payment systems

Name	Payment [Payment Package]
Type	Abstract class
Description	Implements the Observer, base type for different types of payment
Realisations	PayPal, WebMoney, iDeal

Name	Distributor
Type	Actor (external)
Description	Distributor manages the games uploaded, prices, etc through a distribution client (external application).

11. CRITIQUE

While happy with the project and the work we put into it, we feel it would be remiss not to discuss the mistakes and weaknesses that we found upon reviewing our project.

Project Scope

To jump back to Week 2, to the beginning of our project, we originally planned to build the client *and* the distributor application. We planned to use the interaction of these as a good point for potential concurrency e.g. the simultaneous downloading and installing of multiple games from multiple sources. However, once we began to properly consider the amount of work involved in this, we realised that this was far too ambitious at every level of the project. We should have chosen a product that would have been feasible in the time allocated but still featured a greater support for concurrency.

UML Workbench

When picking our choice of UML Workbench, we did not ensure Visual Paradigm had complete support for the diagrams necessary, nor a simple exportation of a Data Dictionary. In our construction of our ERD, we found out that Visual Paradigm treats all ERDs as diagrams that would only be used in database design, separate from the rest of the project. This meant entities that existed elsewhere in our project e.g. Account, Cart etc did not correlate with the entities one would create in the ERD diagram. This also became a problem when we found out that Visual Paradigm exports its' Data Dictionary based upon the Entity Relationship Diagram in a project.

UML

Related to this are the difficulties we had with UML as a Modeling Language. There were multiple instances when we felt classifications in UML documentation or guides were unclear e.g. When working on our Use Case Diagrams we asked our lab assistant, tutor and lecturer, and got different answers as to whether our diagrams were okay in regards to stereotypes and realisation of other UML techniques. We were then given a rule as how to treat UML specifications "You'll get three different answers from three different sources". This highlights if not a lack of standardisation, a lack of obeying the standardisation with different companies perceiving UML guidelines as it suits them.

Software Lifecycle

After much thought, we wonder as to whether RUP (Rational Unified Process) would have been a better choice for a software lifecycle. Agile is such a stark difference in environment from what we and most developers in industry are used to working in, as most in act a mix of Waterfall and Spiral due to the realities of deadlines and resources. With RUP, iterations progress sequentially with at least some level of development time spent on previous and future workflows i.e. portion of time spent working on design, implementation etc while the largest portion of the iteration is

spent on requirements. Similarly, the realities of having access to a stakeholder, while Agile requires, can be a difficult one to achieve for many companies.

Modifiability

We spotted some areas where a modifiable design was missed e.g. the Account entity. We only have a single type of account currently but we could easily have made it more extensible and useful by having the interface `I_Account` be the interface for a Child Account, Adult Account and even staff Account. This would have allowed us to have different store and community interaction features based on the type of account, including Parental Controls which have become a big part of "connected" modern technology to ensure child safety from online predators and other dangers.

Messaging passing

During our design phase we implemented heavy use of the observer design pattern for use in message passing. We realised that this many observes in the system may reduce system performance and increase the complexity of the system in regards to the decision on what observes should be attached to different views, controller, and adapter entities.

Event handling

We identified that the use of event handlers through the use of interfaces would have produced a more streamlined application and avoid the overhead of attaching observers and updating each of these. This would have been best realised in the network and session packages by providing event Interfaces to the other packages that depend on these services, thereby reducing the requirement for the observers. However since we did subscribe to a specific type of Objects Oriented approach, such as event driven programming, we believe that at the implementation stage further refactoring to a domain specific language would solve this issue.

“Event-driven programming is widely used in graphical user interfaces because it has been adopted by most commercial widget toolkits as the model for interaction. The design of those toolkits has been criticized for promoting an over-simplified model of event-action, leading programmers to create error prone, difficult to extend and excessively complex application code” – Wikipedia.

Program to interfaces not implementation – use patterns

In two instances we have designed to implementation and not the interfaces, primarily the Payment checkout package and the Item management package. The use of a factory pattern would have solved this problem of creating window sub components and each of these concrete component controllers / views would override the behaviour of the abstract component providing more flexibility in the creating of each view within the system.

The payment package system is highly coupled with the checkout control entity. Through the use of strategy pattern the family of different payments methods would allow the variation in behaviour while decoupling them from the checkout controller.

Performance - Support undo and redo / back and forward

During our design analysis we identified that the application could have also had its' performance increased by the use of the memento pattern. This would reduce the network load to the distributor while providing the user to undo and redo actions; whether in the form of history through pagination or payment back out or back up to correct errors.

Interface Package or code replication

The majority of the packages are dependent on the main package for the observer interface. This would have been solved by introducing the observer interface into each of the packages, yes it is code copying however, since there is n no real replication of implementation we believe that this is acceptable.

Conclusion

While happy with the effort and overall quality of our work, we feel we learned much that we will have to keep in mind when we begin our next software development project to prevent us making such errors in that and in other future projects.

12. REFERENCES

Object-Orientated Systems Analysis and Design – 4th Edition

[Bennett, McRobb & Farmer]

Using UML – Software Engineering with Objects and Components – 2nd edition

[Stevens & Pooley]

Design Patterns - Elements of Reusable Object Oriented Software

[Gamma, Helm, Johnson, Vlissides]

Patterns of Enterprise Application Architecture

[Martin Fowler]

Refactoring

[Martin Fowler]

Software Quality attributes: Following all the steps

[Clarrus Consulting Group Inc]