CPSC 462 - Software Design

Instructor: Christopher T. Ryu, Ph.D., Professor

NewGen POS
Project Report

Submission Date : 12.17.12

.

Team: NewGen

Name: Michael Robertson	**Email:** mirob2005@gmail.com	**Percentage:** 30%
Name: Xinchao Liu	**Email:** lookfor@188.com	**Percentage:** 20%
Name: Wee Siang Wong	**Email:** willydk@gmail.com	**Percentage:** 20%
Name: Bryan Tamada	**Email:** btamada@gmail.com	**Percentage:** 20%
Name: Wei Jen Lin	**Email:** littlemike44@hotmail.com	**Percentage:** 10%

Project Summary:
Our project focuses on creating a POS system that is easy to use and more efficient than existing systems. The goal is to simplify the checkout process and provide the user an interface that is easy to understand and expedites sales transactions. The development process model we used were the Scrum agile process and extreme programming (XP) methodology. The architecture used is a layered design that separates the user interface to gather user input, domain to satisfy the business requirements and technical services for testing and communicating with external services. Our product abides by the object-oriented design principles, applies the IS-A and 100% rules for class hierarchies and takes into consideration necessary quality attributes to increase software quality. In retrospective, we were able to successfully implement the Qt framework to work with our domain objects and SQL server to get our software in working order. However, since our classes were tied together with the Qt framework, slight modifications to the classes were necessary in order to apply the six design patterns to our application.

Table of Contents

1) Team Description

Team name: NewGen

Product Name: NewGen POS

Scrum Master: Michael Robertson (mirob2005@gmail.com)

Members:

- Name: Xinchao Liu
- Email: lookfor@188.com
- Background: Knowledge of Microsoft Office suite, C++ & Java Programming Languages, Web Programming (PHP & MySQL)

- Name: Wee Siang Wong
- Email: willydk@gmail.com
- Background: Knowledge of Microsoft Office Visio, Java, C++.

- Name: Bryan Tamada
- Email: btamada@gmail.com
- Background: Knowledge of MS Office, C++, Java, PHP, and MySQL.

- Name: Michael Robertson
- Email: mirob2005@gmail.com
- Background: Knowledge of MS Office, C++, Java, Scripting Languages (Python, Ruby/Rails, Perl), Database Programming (MySQL), Web and Graphics Programming

- Name: Wei Jen Lin
- Email: littlemike44@hotmail.com
- Background: Knowledge of MS Office, Web Programming (PHP & MySQL)

2) Team Management

- **Development Process**
  - Scrum with some of the UP practices
  - XP practices with additional practices as necessary.
- **Software Tools**
  - MS Office (Visio, Word, etc.)
  - Java IDE (Eclipse, NetBeans, XCode, etc.)
  - GUI Framework (Qt)
  - Version Control (GitHub)
- **Programming Languages**
  - Java
  - JUnit (Testing)
- **Operating System**
  - Windows
  - Unix
- **Database Management**
  - MySQL
- **Computing Environment**
  - PC
  - Mac

The team is able to provide for all software/hardware necessary to complete this project.

3) Product Description

- ## Product Vision

Our Point-Of-Sale (POS) system is intended for employees of a grocery chain who need a user-friendly POS system to record sales of products to customers. The Point-Of-Sale system is an information system that will allow cashiers to record sales of products to customers including the sale data such as date, product name, price, total sale price, method of payment, along with allowing management to keep track of all product sales. Unlike the current system, our product will be user-friendly as well as being simple and easy to operate.

- ## Product Scope

Our Point-Of-Sale (POS) system will provide the employees of the grocery chain to record sales of any products sold along with storing sale data such as the date of the sale, employee who made the sale, specific items sold, price of each item, total sale price, method of payment, customer information, etc. The system will support such payment methods such as credit, cash, and check along with each method's required information. The system will calculate the total price including tax for the selected items rung up for sale. The cashier will record the items selected to be purchased and the system will respond with a total sale price along with a request of payment method. The cashier will then select the payment method and the system will respond accordingly. If the payment method is cash, the cashier will be asked to input the amount of cash given by the customer and the system will display the amount of cash back. If the payment method is credit, the system will request the credit card type along with the card number and expiration date. If the payment method is check, the system will request the name and bank information. Once the cashier provides this information, the system will authorize the information and return with either an accepted or declined form of payment. If declined, a new payment method must be used. The system will also keep track of employees allowing different levels of access to regular employees such as cashiers and those with higher access such as management. Cashiers' access will be limited to the features stated above, while management will be able to track sales by date, cashier, or customer. The system will produce accurate calculations as well as being easy to use, fast. and reliable. The system will be designed as a complete replacement for the current Point-Of-Sale (POS) system of the grocery chain.

4) Use-Case

ID: 001

Name: Process Sale

Description:

The POS system will record purchased items, calculate a total along with product names, customer payment method, update store inventory and print out a summary of purchased items in the form of a receipt for the customer.

Actors: cashier

Pre-condition:

cashier ready

Basic steps:

1. customer checkout with goods
2. cashier start a new sale
3. cashier enter items
4. system present item name and price

cashier repeat step 3-4 until record all goods

5. system present total price with tax
6. cashier asks for payment
7. customer pays and system handles payment
8. system logs sales
9. system prints receipt
10. customer leaves with receipt and goods

Post-condition:

Amount compute correctly, payment authorized, sales saved, inventory updated, receipt generated.

Priority: high

Special requirements:

large font text to guarantee screen visibility
quick payment authorization response

Memo (open issues):

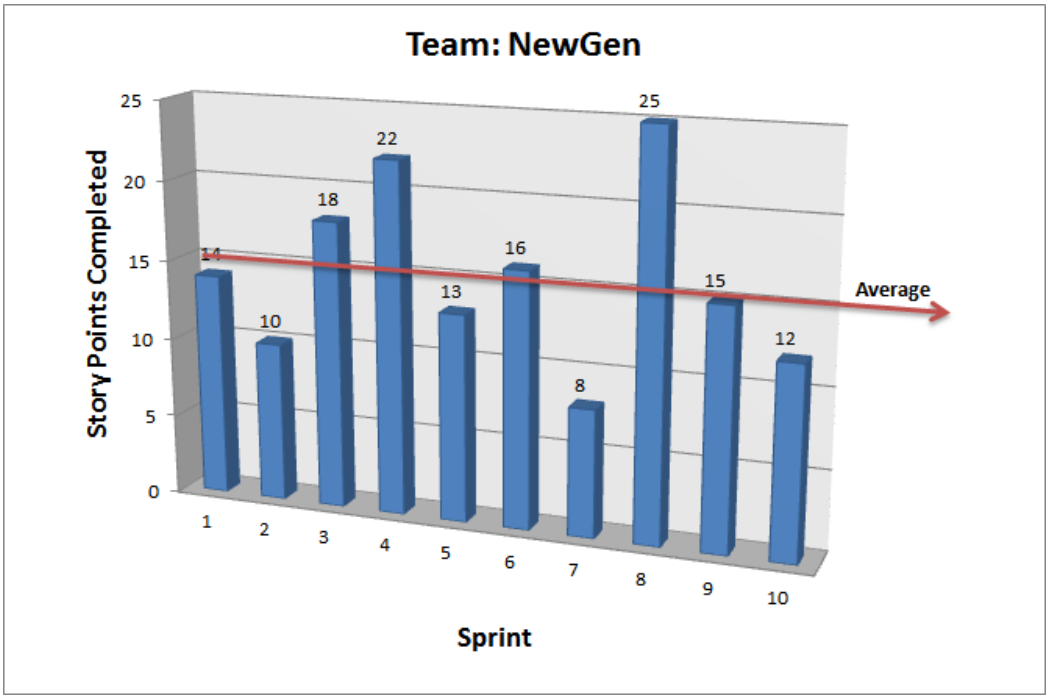
need to support manager’s override operation

5) Product Backlog

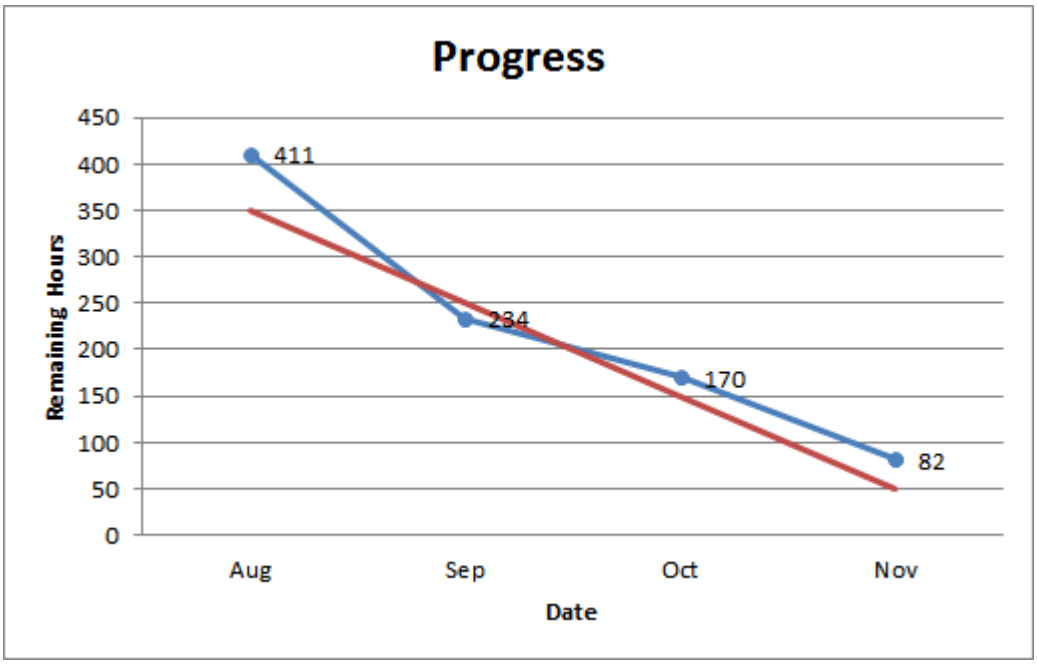
- **Sprint Backlog**

Who	Description				
Total Estimated Hours:		350	250	150	50
	Environment				
Michel	Install Git	4	2	2	
Michel	Setup GitHub	4	4	4	4
Will	Setup JDK	4	2	2	
Will	Install QT	4	2	2	
Lin	Setup MYSQL	4	4	4	4
	Database				
Bryan	Design Database	20	16	12	
Leo	Create MYSQL tables	15	12	8	
Michel	Connect POS system to database	8	4	4	4
Will	Add sales records in database	8	2		
	Design Model				
Leo	Domain Model	24	12	8	
Bryan	Sequence Diagram	24	12	8	
Lin	CRC models	24	16	12	
Will	Design System based on GRASP RDD pattern	16	8	8	
Michel	Check OO Design Principles	16	8	8	8
	Implementation				
Michel	Polymorphic Operation Example	12	8		
Will	Add test template	12	8	4	2
Leo	Add input testing for productID and qty	12	4	4	
Bryan	Move main method	12	4	4	2
Leo	Make class variables private	12	4	4	2
Bryan	Incorporating product catalog, register, store objects	12	8		
All	Build Objects	40	40	40	24
Will	Receipt Class overhaul	8	8		
Michel	Make payment class working	16	8	8	8
	Test				
Leo	Test using test template	24	12	8	8
Bryan	Test all classes seprately	24	12	8	8
Lin	Test objects functionality	24	8	2	2
Will	Report test results	12	2	2	2
Michel	Remove extra methods	8	2	2	2
Leo	Clean up all codes	8	2	2	2

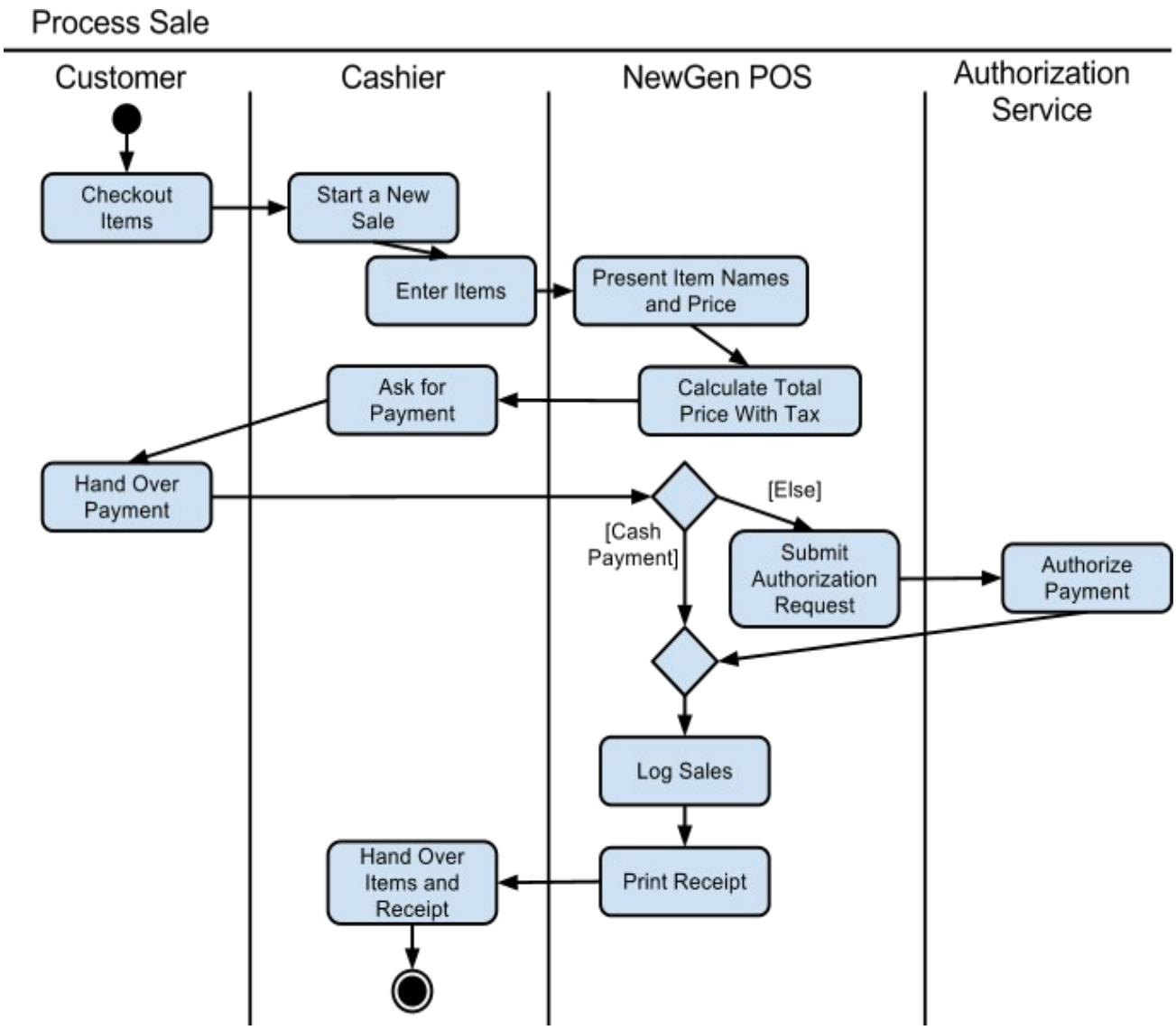
- **Sprint Velocity Chart**



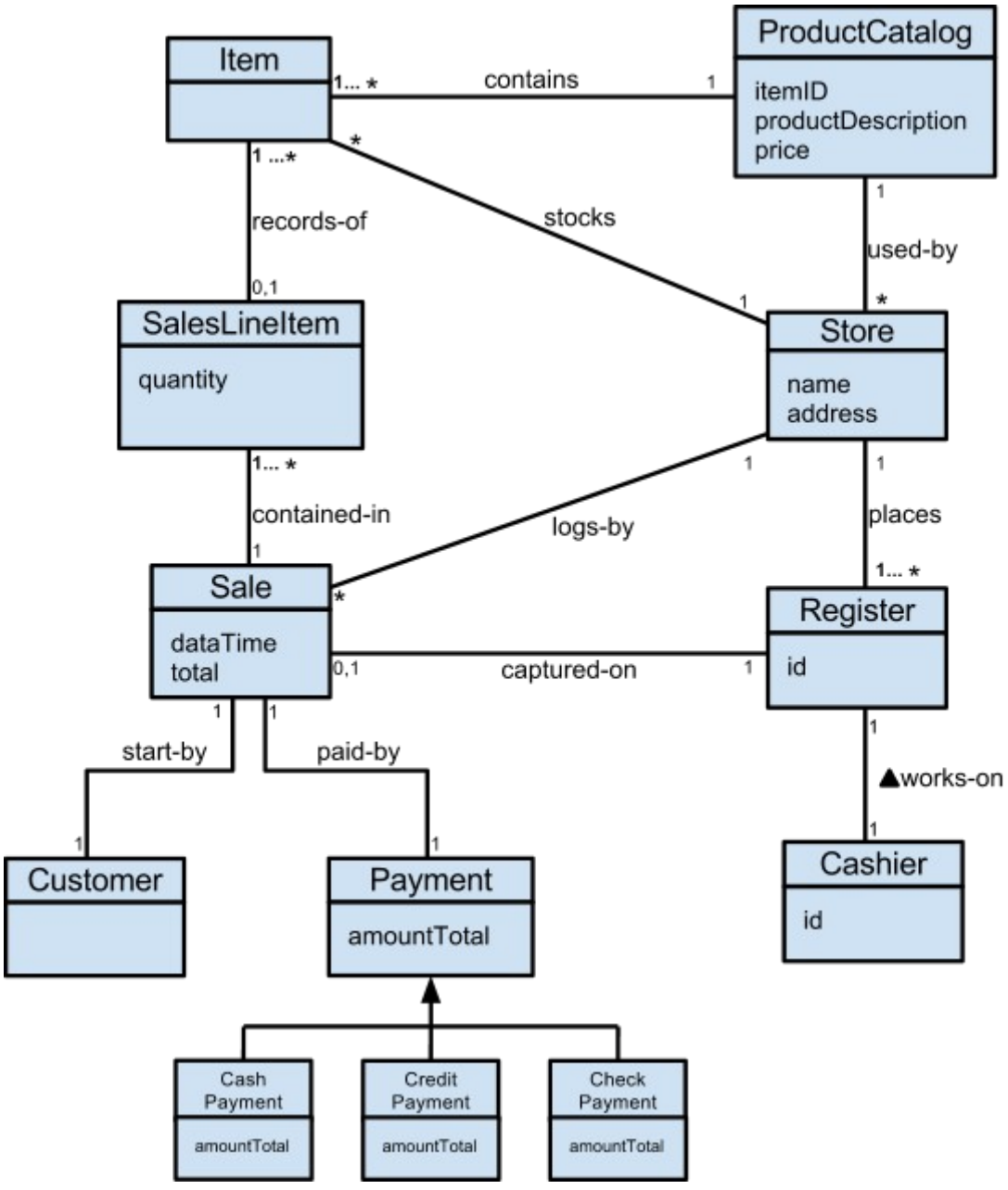
- **Burn-down Chart**



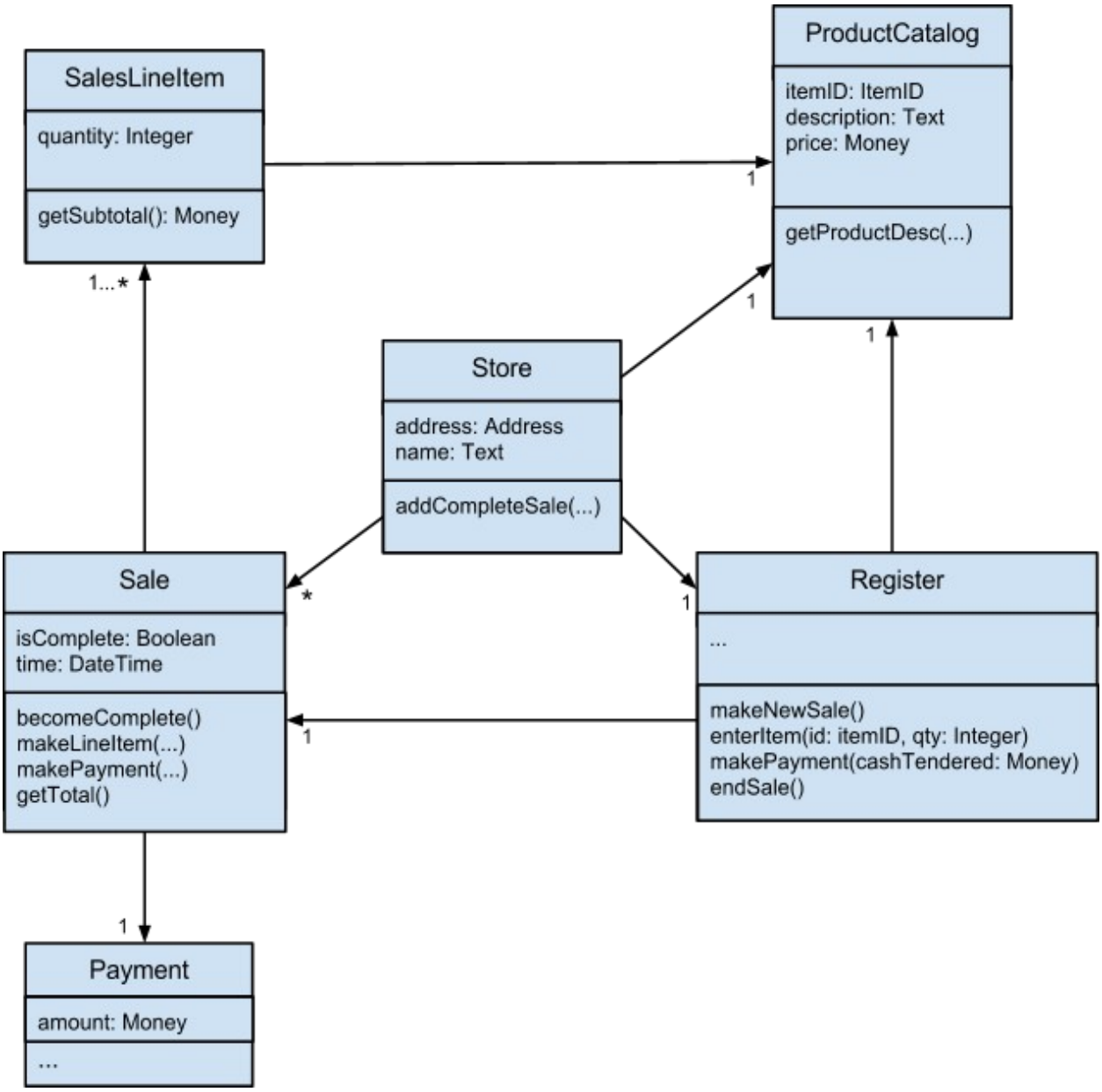
6) Activity Diagram



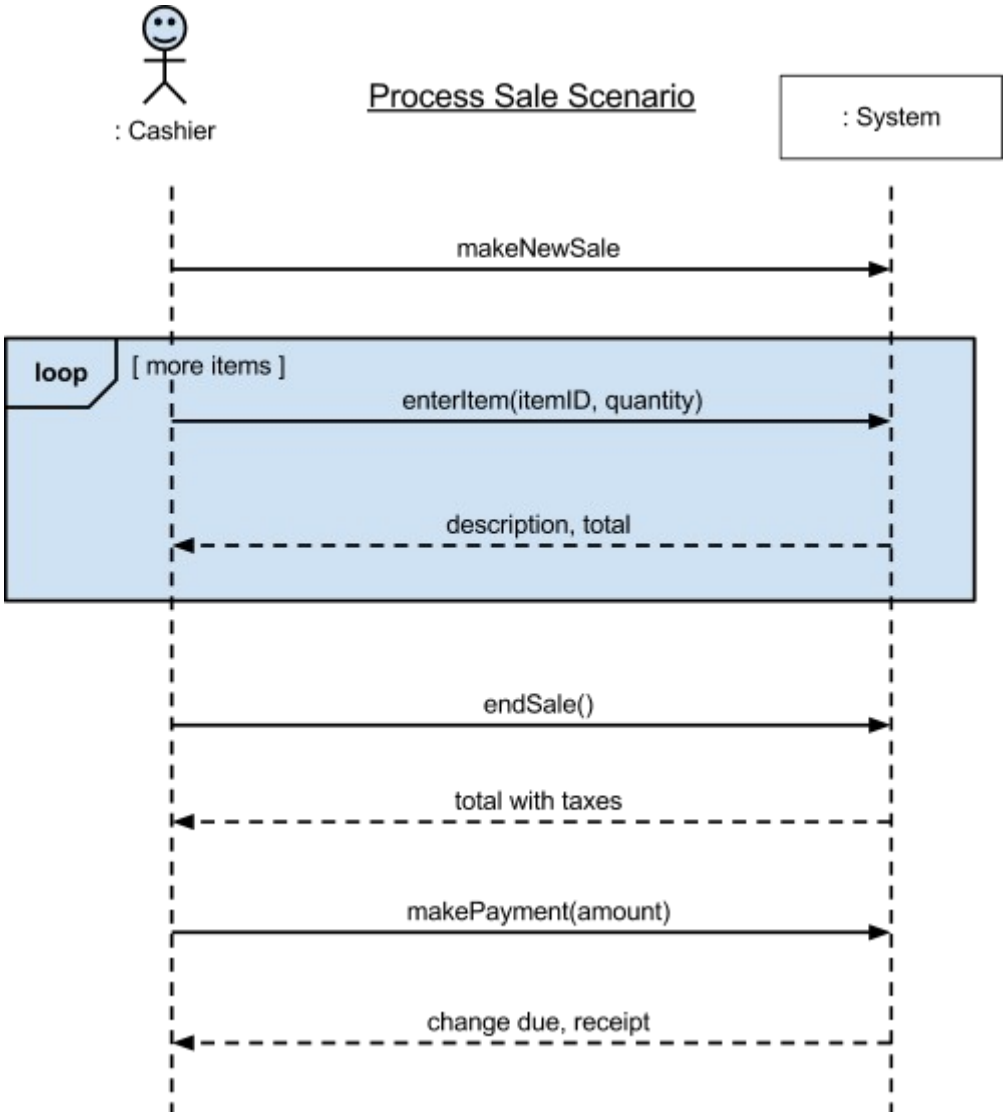
7) Domain Model



8) Data Model


SalesLineItem
quantity: Integer
getSubtotal(): Money
ProductCatalog
itemID: ItemID
description: Text
price: Money
getProductDesc(...)
Store
address: Address
name: Text
addCompleteSale(...)
Sale
isComplete: Boolean
time: DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
Register
...
makeNewSale()
enterItem(id: itemID, qty: Integer)
makePayment(cashTendered: Money)
endSale()
Payment
amount: Money
...
1...*
1
1
1
*
1
1
1

9) System Operations



10) System Operation Contract

Operation:

submitPayment()

Cross reference:

Process Sale

Preconditions:

Cashier has input all items checked out by the Customer. The point of sale system has correctly calculated the total price with tax. Cashier informs the Customer of the total price and the Customer decides on a payment type, such as cash, credit, or check payment.

```
Register.java
public void enterItem(ItemID ItemID, int qty)throws SQLException{
       description = catalog.getProductDescription(ItemID, qty);
       //No such ItemID or out of stock returns description = null
       if(description == null){
          Ui_NewGenPOS.clearProductInput();
       }
       else
       {
          currentSale.makeLineItem(description, qty);
       }
}
Sale.java
public void calcTotal(){
this.total = this.pricing.calcTotal(this.subTotal);
       Ui_NewGenPOS.setDisplay(this.total);
}
Payment.java
class Payment {
…
}
class CashPayment extends Payment {
…
}
class CreditPayment extends Payment {
…
}
class CheckPayment extends Payment {
…
}
```

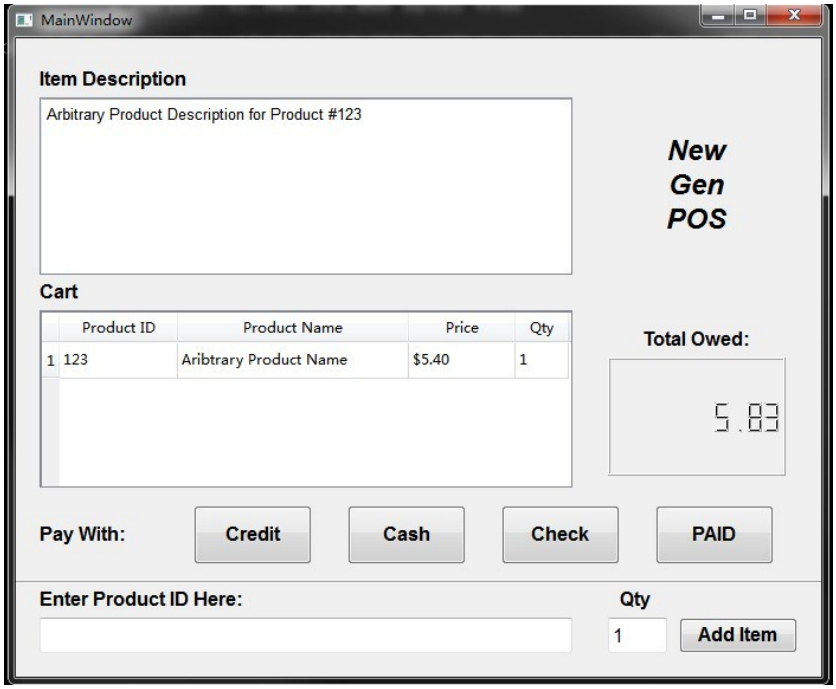
Postconditions:

Changes were received by the customer if there were any. The authorization request was submitted by the system if the customer pays with credit or check. Sale was logged. Receipt was generated by the POS system and Cashier hands it to the customer. Customer leaves with the receipt and items purchased.

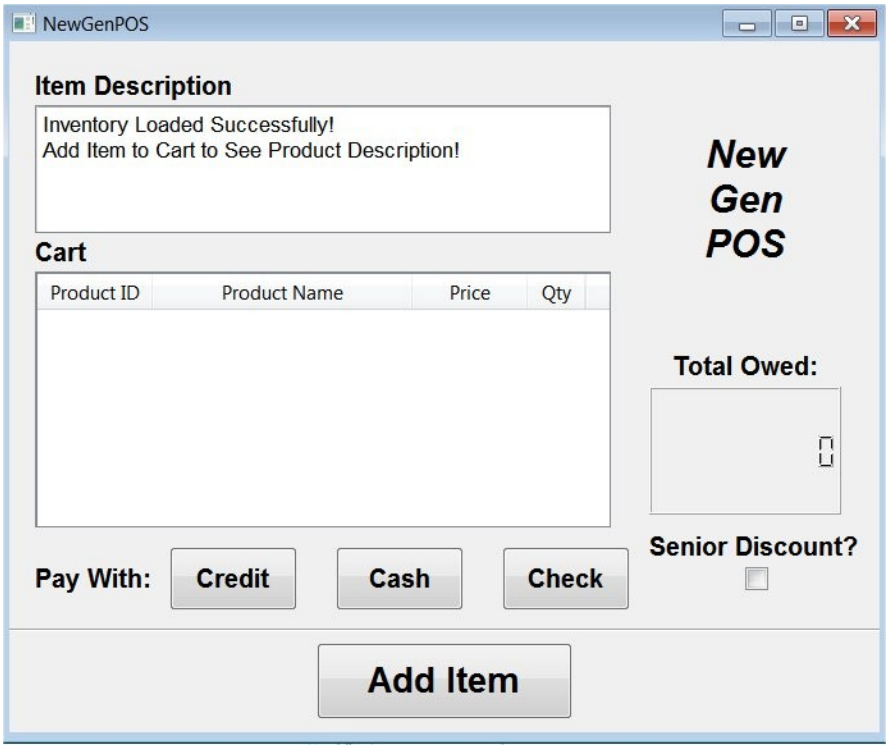
```
Register.java
public void endSale(){
currentSale.becomeComplete();
        Ui_NewGenPOS.setText("Thank You for Shopping!");
        Ui_NewGenPOS.clearProductInput();
        Ui_NewGenPOS.clearCart();
        Ui_NewGenPOS.clearDiscount();
        Ui_NewGenPOS.setDisplay(0);
        this.currentSalesNumber = this.currentSalesNumber+1;
}
public void createReceipt(){
…
}
public void makePayment(...
public void recordSale(){
…
}
```

11) Interface Prototype and Snapshot of All Windows

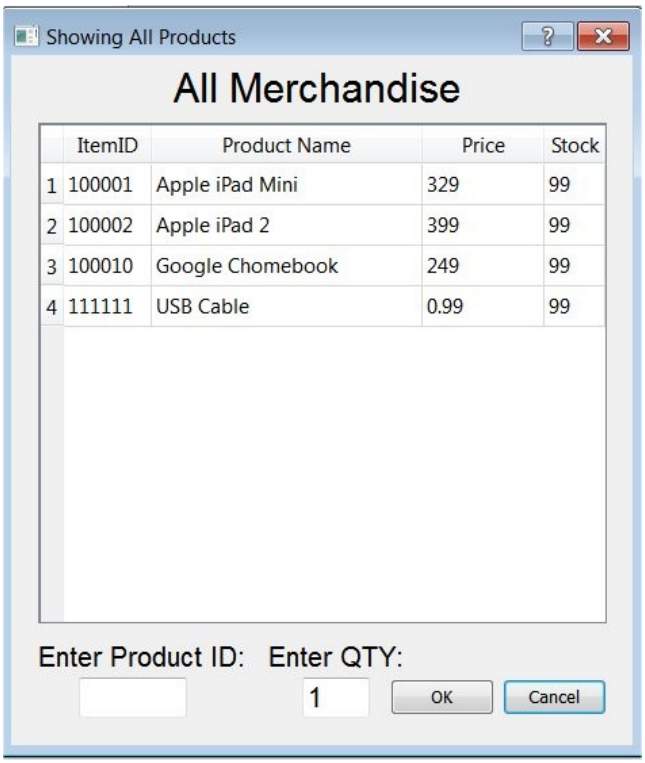
Initial Prototype:



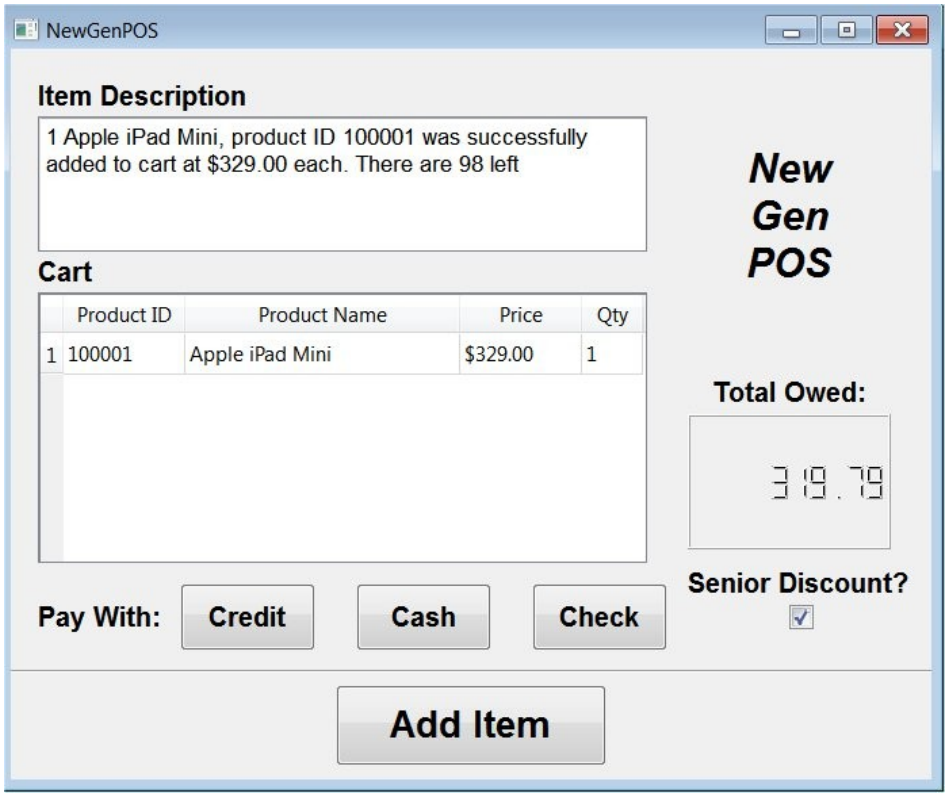
New Interface Images:



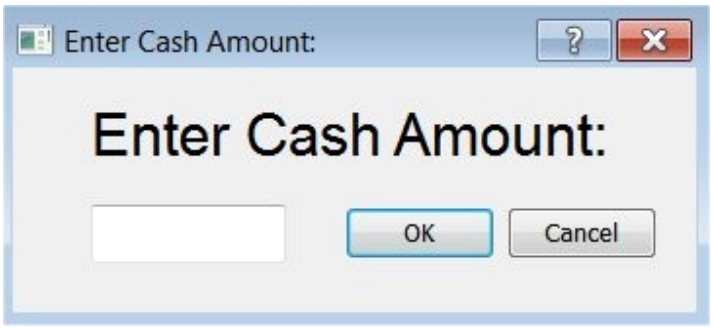
Add Item Dialog:



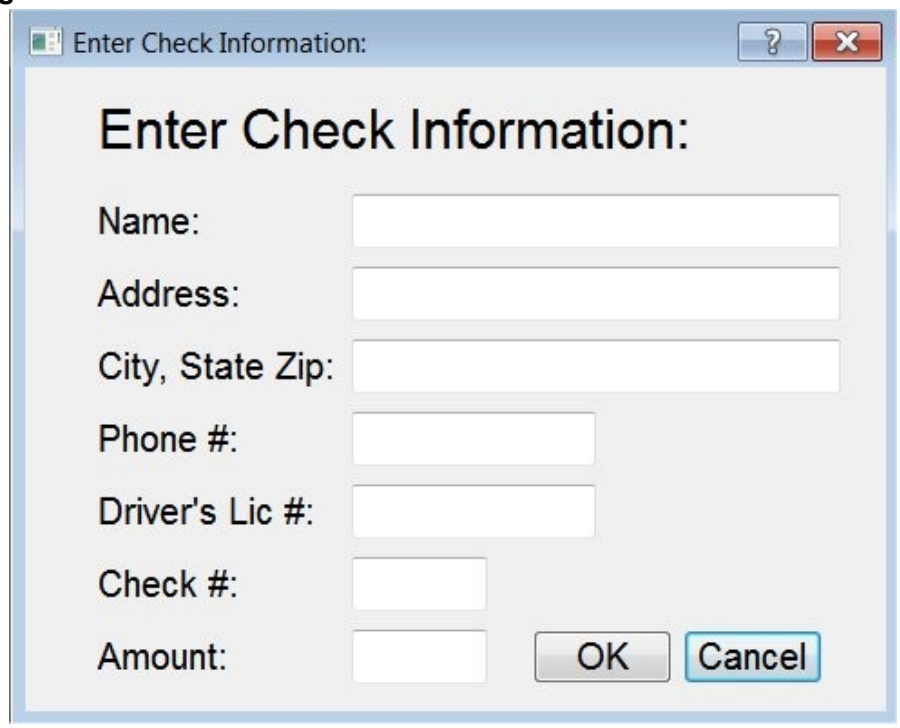
After item is successfully added to cart:



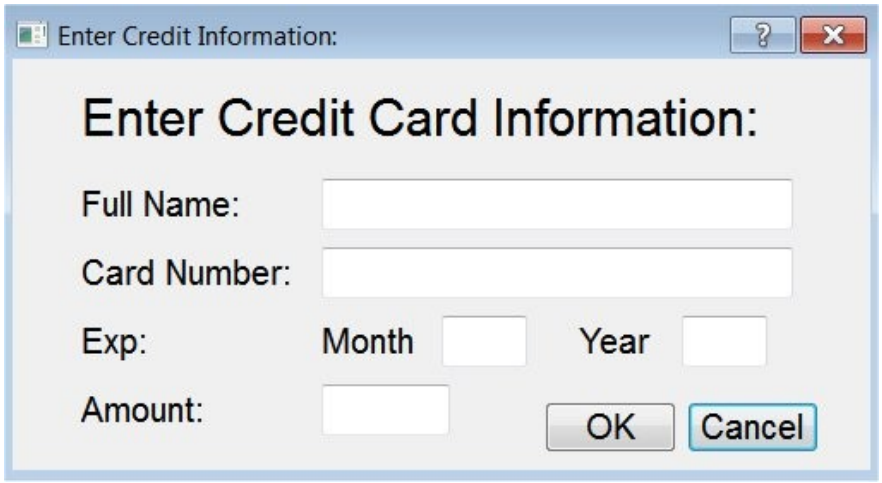
Cash Dialog:



Check Dialog:

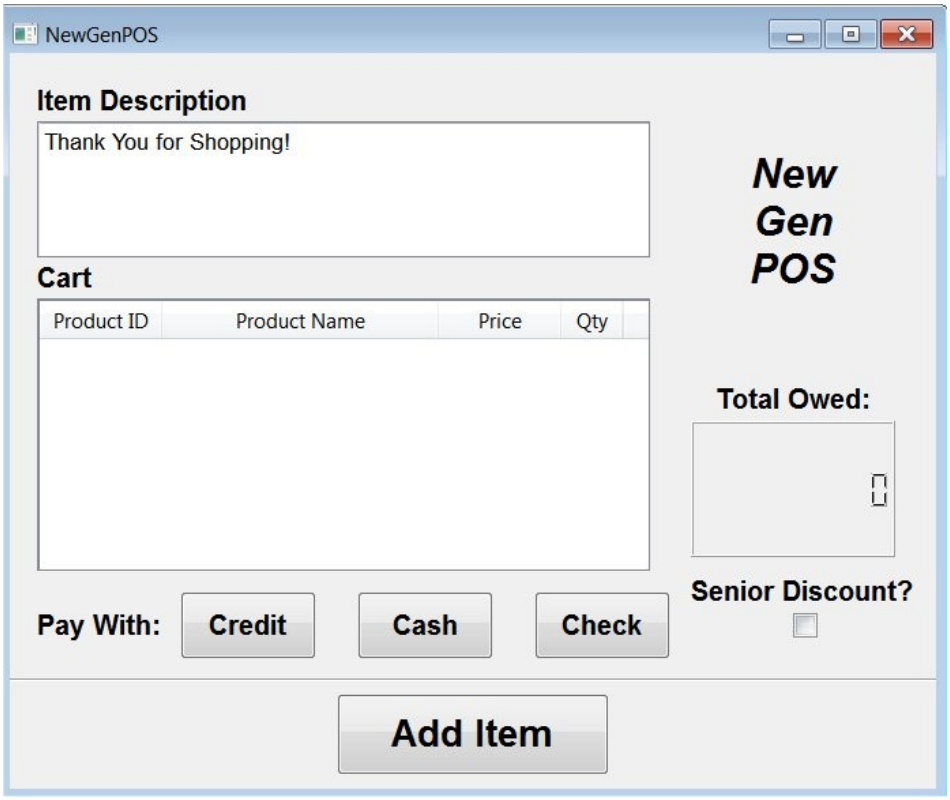


Credit Dialog:



Receipt Dialog:

After sale is final:



12) CRC model

Class: Store	
Responsibility:	**Collaborator:**
maintains product catalog	ProductCatalog
stores items	Item
logs sale orders	Sale
manages register status	Register

Class: ProductCatalog	
Responsibility:	**Collaborator:**
manages item id/price	Item, ItemID, Money
stores item descriptions	Item
genarates catalog for store	Store

Class: Item	
Responsibility:	**Collaborator:**
define item id/price	ProductCatalog
define stock status	Store

Class: ItemID	
Responsibility:	**Collaborator:**
keeps item ID in right format	

Class: Sale	
Responsibility:	**Collaborator:**
adds SaleLineItem	SaleLineItem
logs sale date/time	Store
logs sale total payment	Payment
defines sale status	Interface, Register

Class: SaleLineItem	
Responsibility:	**Collaborator:**
shows item id/price	Item
shows item quantity	Sale

Class: Payment	
Responsibility:	**Collaborator:**
defines payment total amount	Sale
computes amount for change	Money
verifies payment	Register

Class: Register	
Responsibility:	**Collaborator:**
define register status	Store
starts a new sale	Sale
adds item to sale	SaleLineItem, ItemID
accepts tendered money	Payment, Money

Class: Money	
Responsibility:	**Collaborator:**
converts price to right format	

Class: Interface	
Responsibility:	**Collaborator:**
displays sale detail	Sale, SaleLineItem
displays price	Money
clears cart	
accepts register's commands	Register

The CRC model covers all the requirements directly specified or implied in the use cases. All of the responsibilities within the CRC matches up with all of the system functions and use cases.

13) Responsibility Assignments

Store Class
Holds information about the product catalog, items in the store, a log of sales orders and manages the registers' status. Follows the Pure Fabrication pattern by saving data and objects in a database. In this case it is responsible for storing items and logging sales orders.

Product Catalog Class
Uses the Indirection pattern to information on product ID, product price, storing item descriptions and generating catalogs for the store. It acts as an intermediary between the Item, Store, and Sale classes. Moreover, High Cohesion keeps objects focused, focused, manageable and understandable.

Item ID Class
Follows the Low Coupling pattern by supporting low dependency, low change of impact and increased reuse. Also acts as an Indirection between two elements by being assigned as the mediator to an intermediate object.

Sale Class
Stresses the importance of Information Expert determine where to delegate responsibilities. The Sale class works with the major classes in completing the sale transaction and is the main function of the POS system. Also uses High Cohesion to keep objects appropriately focused, manageable and understandable; has control to several of the system methods that control the system.

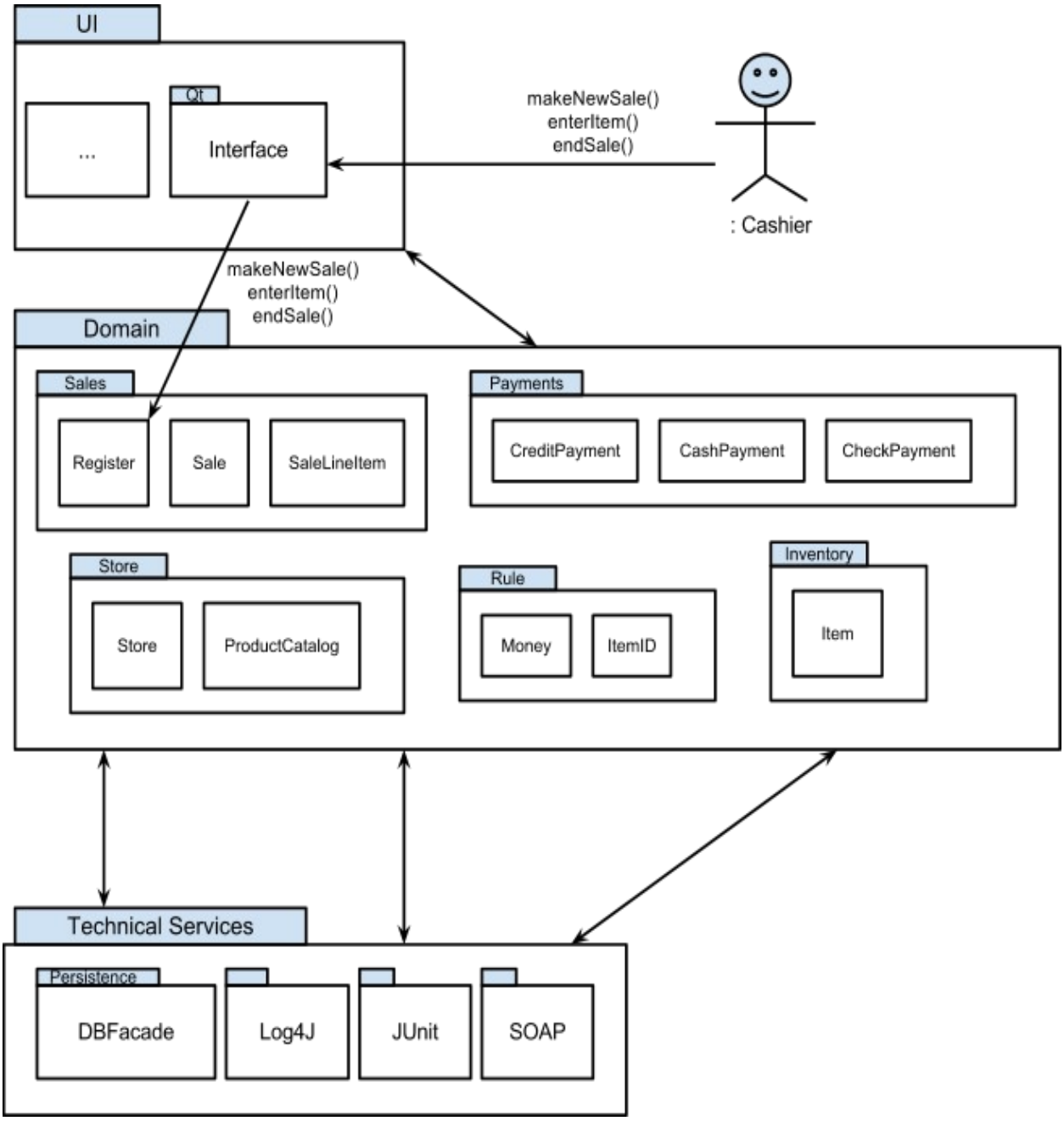
Sale Line Item Class
Acts as an intermediary in holding product information such as product id, product price, and product quantity. This is follows the Indirection pattern because it the responsibility is placed on this class to provide the information requested by the Item and Sale classes.

Payment Class
This class defines the payment total amount, computes the amount for change and verifies payments. The Payment class abides by the GRASP patterns in that it is polymorphic by being able to be instantiated into different forms of payments: check payment, credit payment and cash payment. Responsibility for when behavior varies by type such as the case in the various types of payments using polymorphic operations.

Register Class
Uses the Information Expert pattern by to determine which classes to delegate responsibilities to. Interacts with several classes such as Store, Sale, SaleLineItem, Payment to execute several of the system methods. Acts as the Controller since it is the main class that functions as the main class of the system.

Interface Class
This class follows the Protected Variations pattern by protecting elements from the variation on other elements such as objects, systems and subsystems by wrapping the focus on instability with the interface and creating various interfaces using polymorphism for implementation.

14) Architecture Diagram



Usability - The POS System is an application that is made to be easier for the user to interact with as well as meet the requirements of the user and consumer. The usability quality attribute focuses in on the overall user experience by providing intuitive, easy to localize and globalize and easy access software.

Maintainability - Our POS system should be able to undergo changes with a certain amount of ease because these changes could impact several parts of the system such components, services, features and the interface. That is why when adding or changing the functionality, fixing errors, and meeting business requirements the system should be easy to accommodate these changes.

Reliability - A system should be able to remain operational over time without much trouble. A POS system running at a grocery store should have very little probability of failing to perform its intended functions over a specific time interval since it is possible that they are running 24 hours per day.

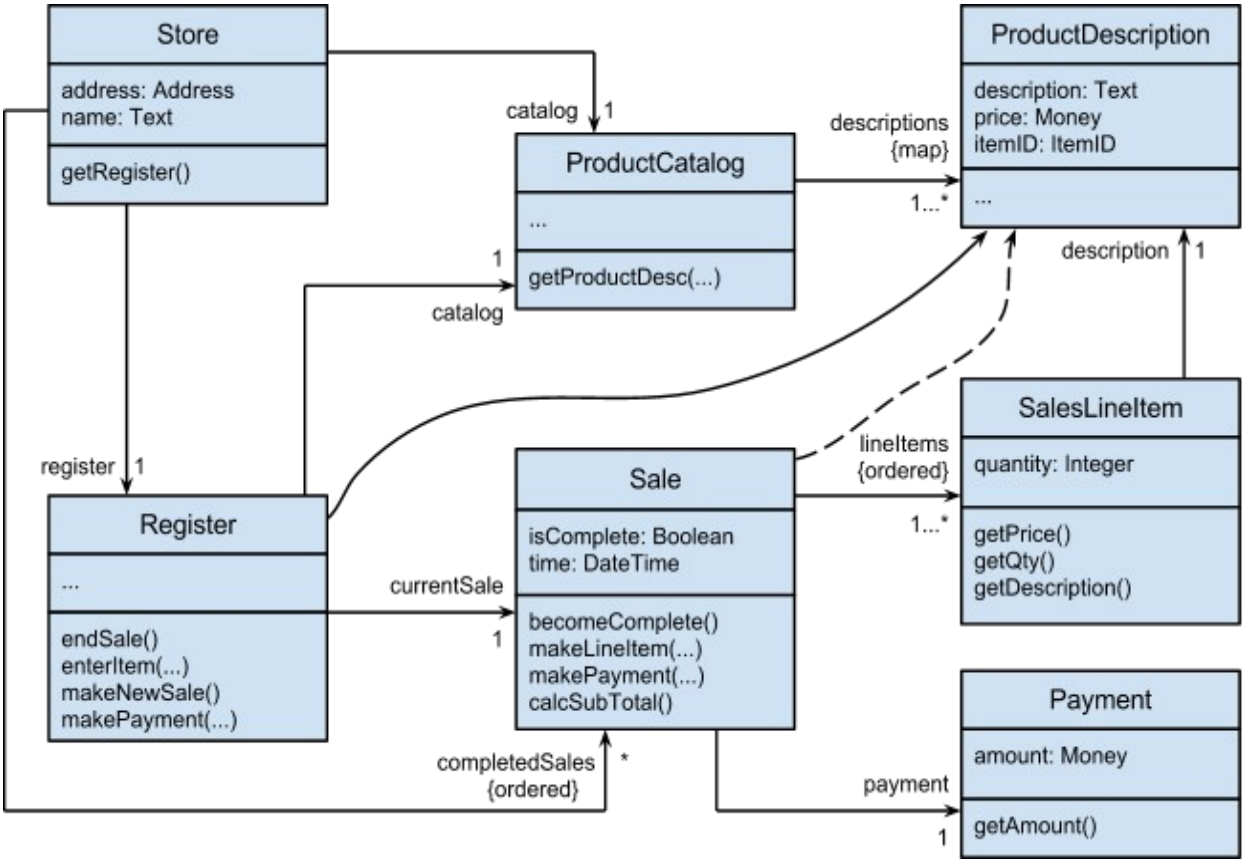
Scalability - In the future it is possible that the system would be required to handle more processes, be able to accommodate new resources (such as hardware), or an expansion of data. To protect against increases in load without sacrificing performance or having the ability to be readily enlarge our system scalability is an important quality attribute to handle these changes and safeguard against future goals while ensuring that the system can be utilized.

Security - Due to our system dealing with third-party entities such as the authorization of checks or credit cards during the payment process we want our system to be secure. A secure system has the capability of preventing malicious or accidental actions outside of its usage and to prevent the leakage, retrieval and modification of personal information.

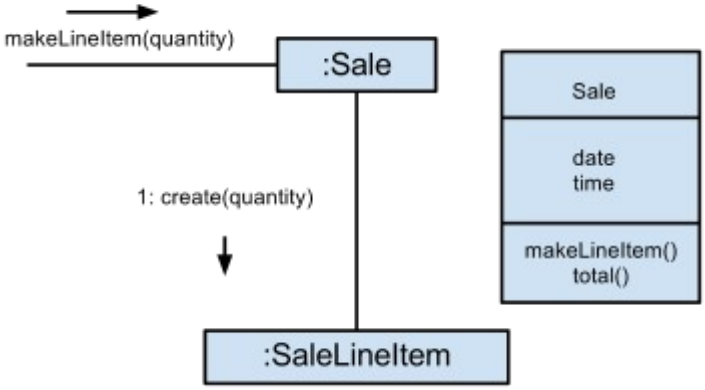
Availability - Similar to reliability, our system must be available for a period of time where it is functional and working. We can measure the system's rate of availability as a percentage of the total system downtime over a predefined period. System availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.

15) GRASP RDD pattern

Design Class Diagram (DCD)



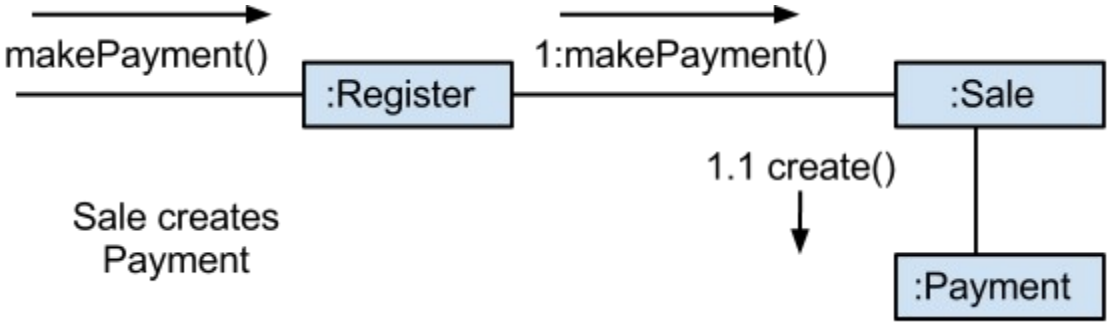
Creator. supports low coupling, increases clarity, encapsulation and reusability.



Information Expert. Product Description class has the information that holds the item's ID number, the description of the item, the price of the item and the number of items left in stock.

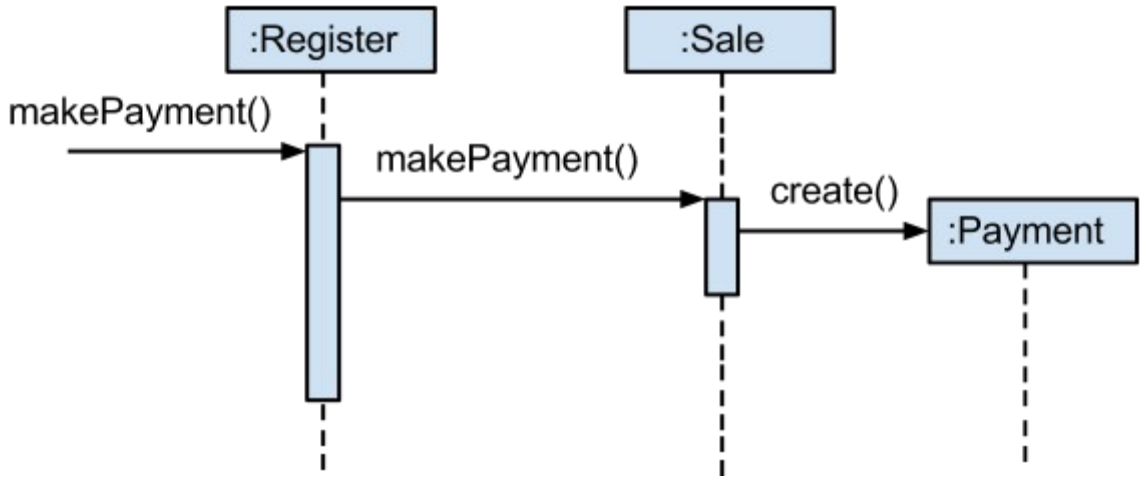
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/ProductDescription.java

Low Coupling. Low coupling is easier to adapt to change, maintain, reuse and understand, the changes are localized.. In our POS, we have the register to make the payment by calling the function makePayment() to Sale class, and the Sale class creates payment to fulfill low coupling principle.

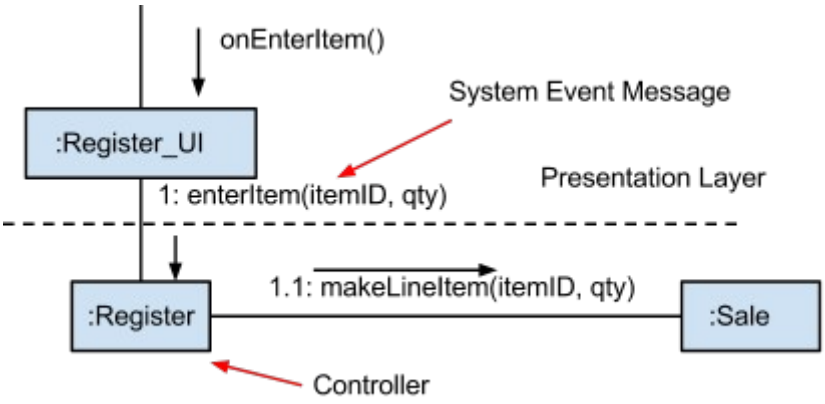


Delegate the payment creation responsibility to the Sale supports higher cohesion in the Register, supporting both high cohesion and low coupling

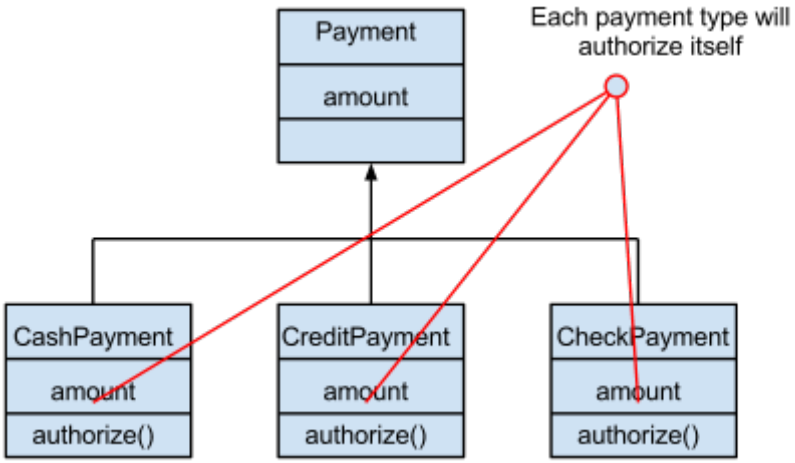
High Cohesion. Classes are clear and easy to understand, easier to maintain, reuse because of fine grained responsibility. Low coupling is often supported.



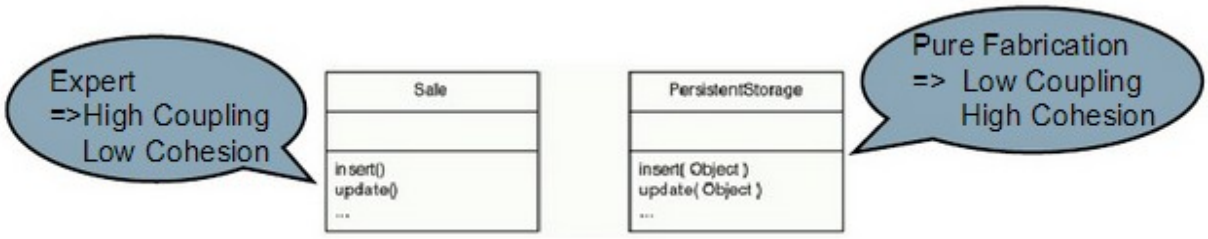
Controller. Having a controller class, it increased potential for reuse and pluggable interfaces, the same benefits of model-view separation principle, and opportunity to reason about the state of the use case.



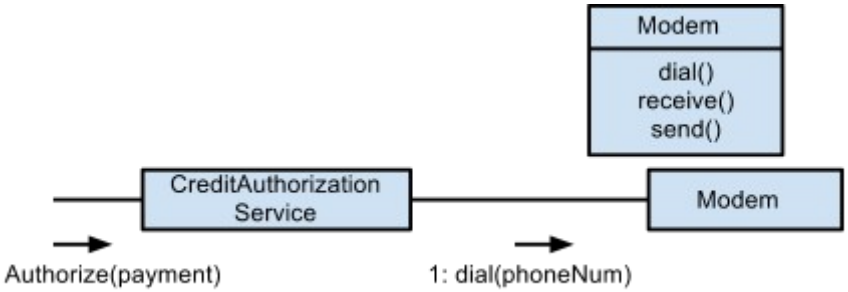
Polymorphism. Payment Class



Pure Fabrication.Benefits: High Cohesion, Increased reuse potential.



Indirection.The Register UI uses a modem to transmit a credit payment request.



Protected Variations.

Every single variable in our design are privately protected, we have data encapsulation, polymorphism, pure fabrication, indirection. Other related principles including, OPen-Closed, Liskov Substitution, Law of Demeter, Dependency inversion, etc.

Benefits are: extensions required for new variation are easy to add, new implementations can be introduced without affecting clients, and the impact or cost of changes can be lowered due to lower coupling.

16) OO Design Principles

Based on the seven OO design principles discussed in class, our design meets with all seven design principles.

- **Command-Query Separation Principle**

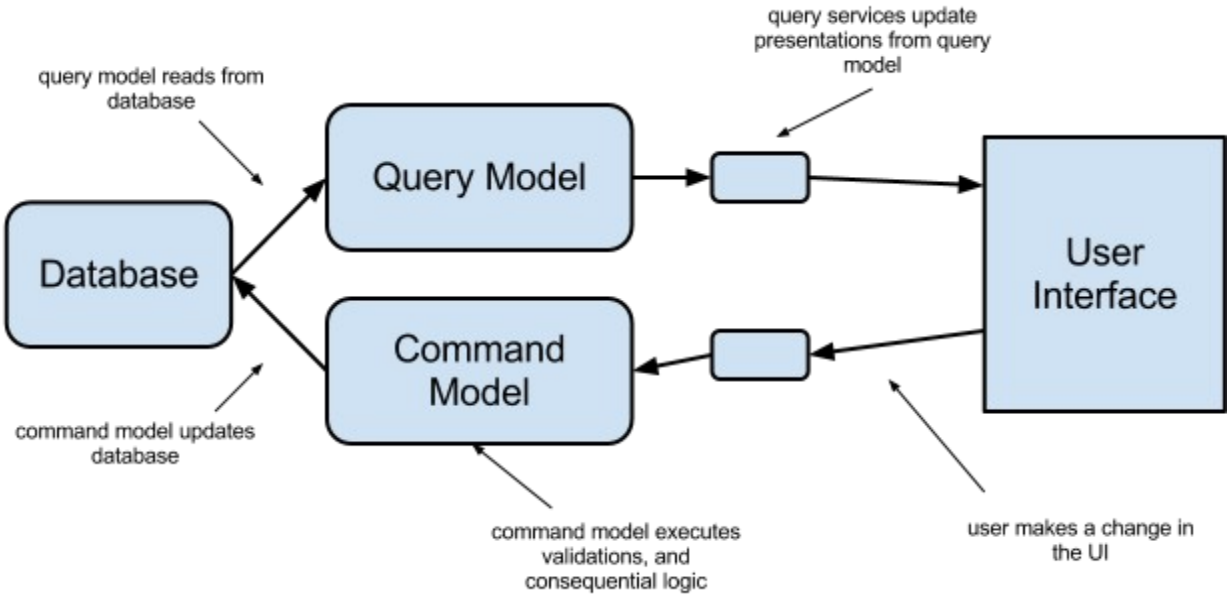
Every method are either command method or query method. For example:

Query:

```
public String getAmount(){
    return this.amountInput.text();
}
public String getCardNumber(){
    return this.cardNumberInput.text();
}
public String getName(){
    return this.nameInput.text();
}
public String getMonth(){
    return this.monthInput.text();
}
public String getYear(){
    return this.yearInput.text();
}
```

Command:

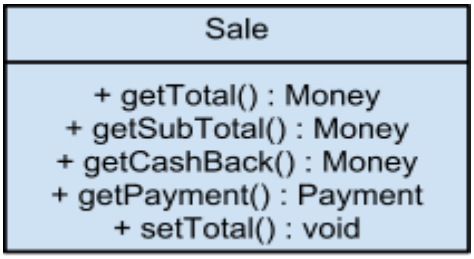
```
private void successfulTransaction()throws SQLException{
    register.createReceipt();
    register.recordSale();
    register.endSale();
    register.makeNewSale();
}
```

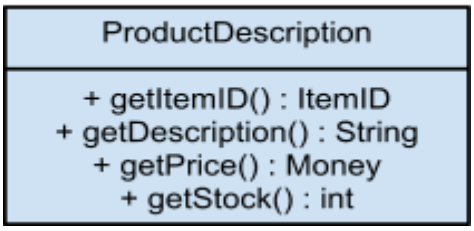


- **Law of Demeter**

Each object have only limited knowledge about other objects, and each object should only talk to its immediate friends. Example:

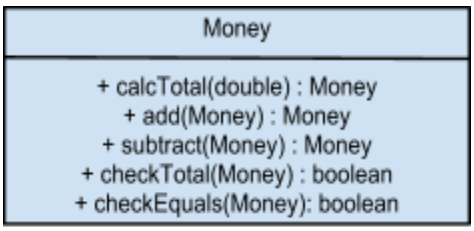
```
public static void addItemToTable(SalesLineItem cart){
    ProductDescription desc = cart.getDescription();
    int qty = cart.getQty();
    ItemID ID = desc.getItemID();
}
```

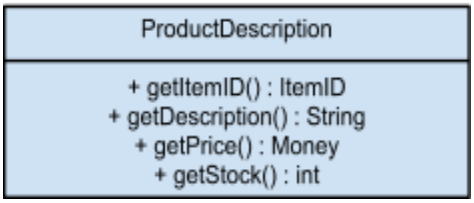




- **Single-Responsibility Principle**

A subsystem, module, class or function, should not have more than one reason to change. Each class in our coding represent each responsibility of the system. For example, we have Money class that deal with the total amount of money being calculated. We also have payment class that deal with cash, check, or credit payments. ProductDescription class that handles item description, ID, price, and how many items are left in stock.





- **Open-Closed Principle**

Module should be both open (for extension; adaptable) and closed (the module is closed to modification in ways that affect clients).

In payment class, the class is open for extension, ex. cash, check, credit payments. and closed for any modification that would affect it.

In Register class, the payment method is called.

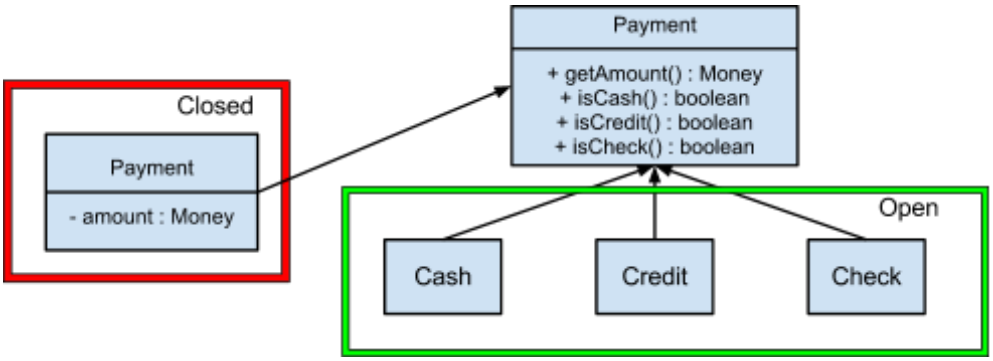
```
public void makePayment(Payment paymentAmount){
 currentSale.makePayment(paymentAmount);
  }
```

Then, the Sale class handles payment.

```
public void makePayment(Payment paymentAmount){
       this.payment = paymentAmount;
       Money payment = this.payment.getAmount();
       this.cashBack = payment.subtract(this.total);
   }
```

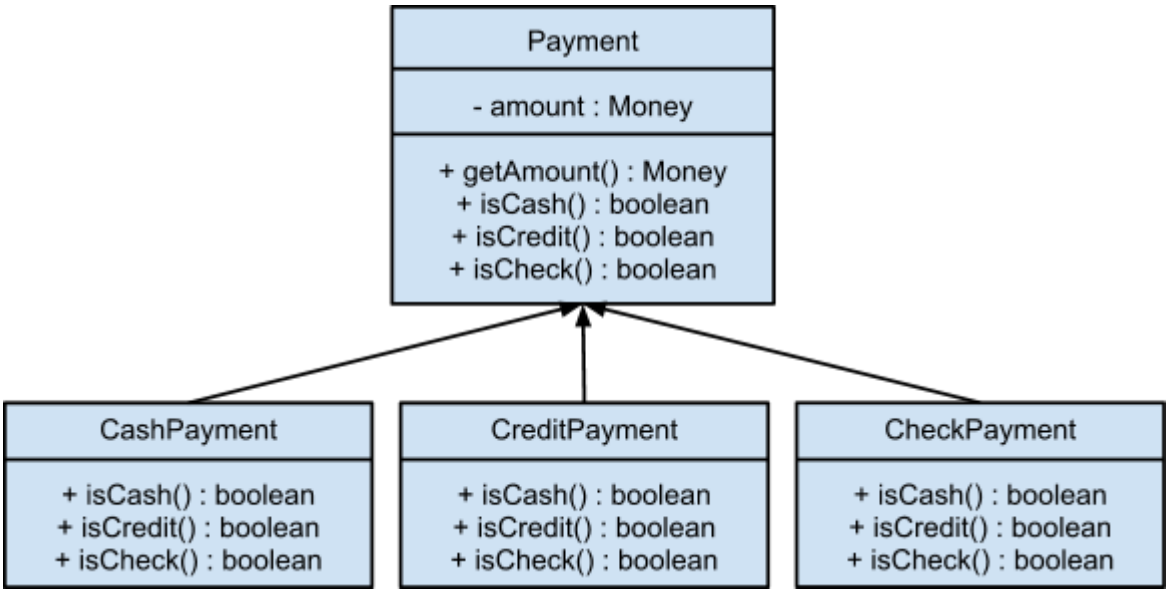
Lastly, Payment class handles the type of the payment being made.

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Payment.java



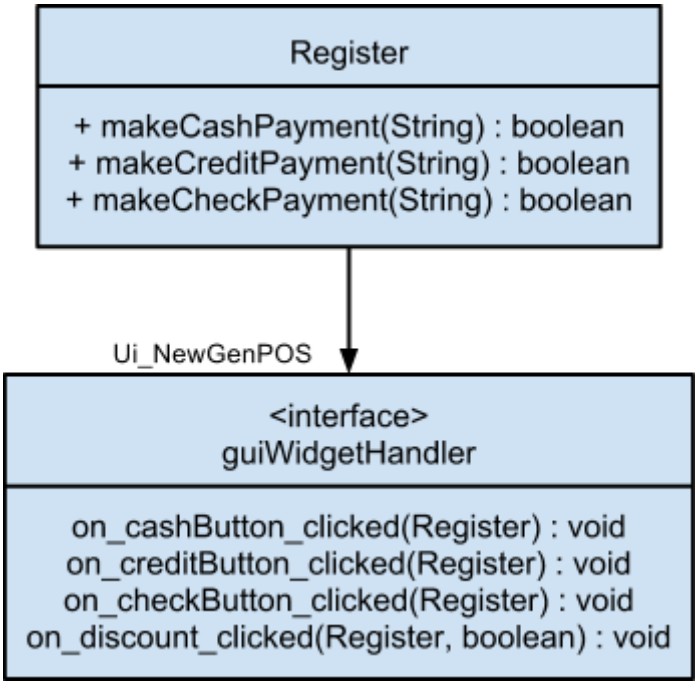
- **Liskov Substitution Principle**

A module that uses a base class should continue to function properly if a class derived from the base class is passed to the module instead. Derived types must be completely substitutable for their base types.
In the payment class, we have payment as our base class, then cash, credit and check payment classes derived from the payment class, and they are all substitutable for their base class.



- **Dependency-Inversion Principle**

The more a module depends on concrete modules rather than abstraction such as interface, the more difficult it will be to extend. Depend on abstractions. Do not depend on concretions.
In the payment class, we have a payment interface that holds amount that is being paid, and check to see if it is cash, credit or check payment. Then we have cash, credit and check payment class that extends payment which those details depend on abstractions. Lastly, we have a submit payment class that does the payment submission to the system.



- **Interface Segregation Principle**

Each major category of clients should create a specialized interface to serve it. Clients should not be forced to depend upon interfaces that they do not use.
In the payment class, each method has its unique function to complete the payment process, payment class has to identify the item number and what the payment transaction that the customer choose to finish the payment. We also create a function to calculate the final amount. The amount will also include the tax plus the purchase price, and the information will show on the monitor to let the customer know.

Ui_NewGenPOS.java - default user interface, displays product description, cart, and total owed
Ui_CashDialog.java - appears once a cash button is clicked and allows user to input the amount to be paid
Ui_CheckDialog.java - appears once a check button is clicked and allows user to input the amount to be paid, check holder's name, check number, phone, address, and license
Ui_CreditDialog.java - appears once a credit button clicked and allows user to input the amount to be paid, credit card holder's name, card number, and month/year expiration date.
Ui_ReceiptDialog.java - appears once the payment have been approved and processed.
Ui_AddItem.java - appears once the user clicks on Add Item button, user is allowed to input item ID and quantity.

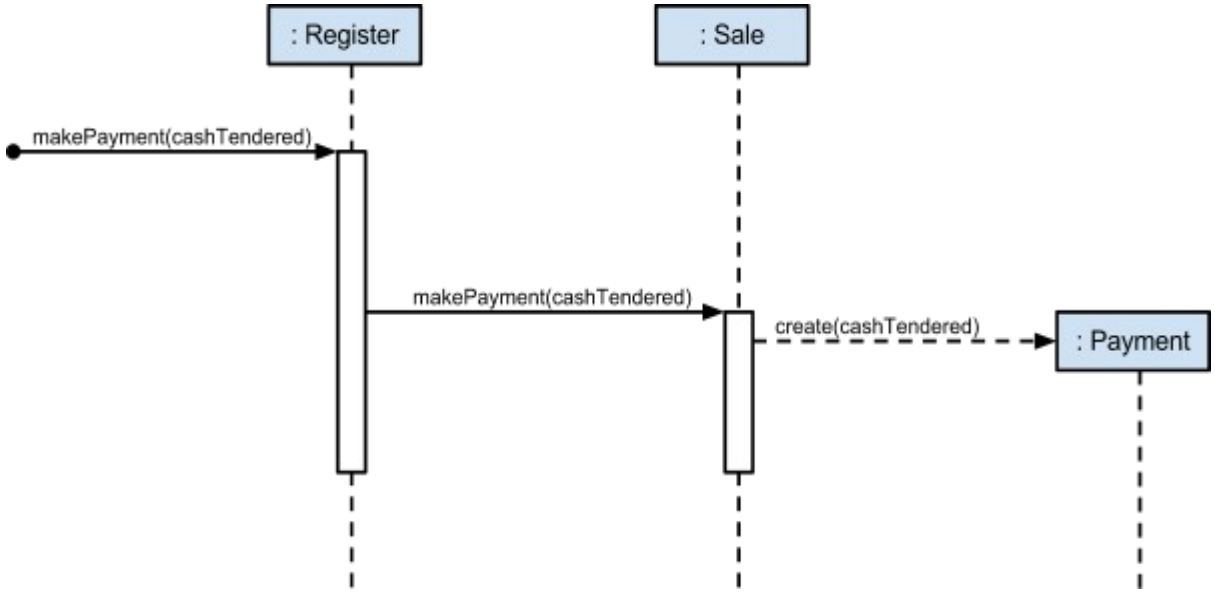
17) Class Hierarchy

- ## Payment Class Hierarchy

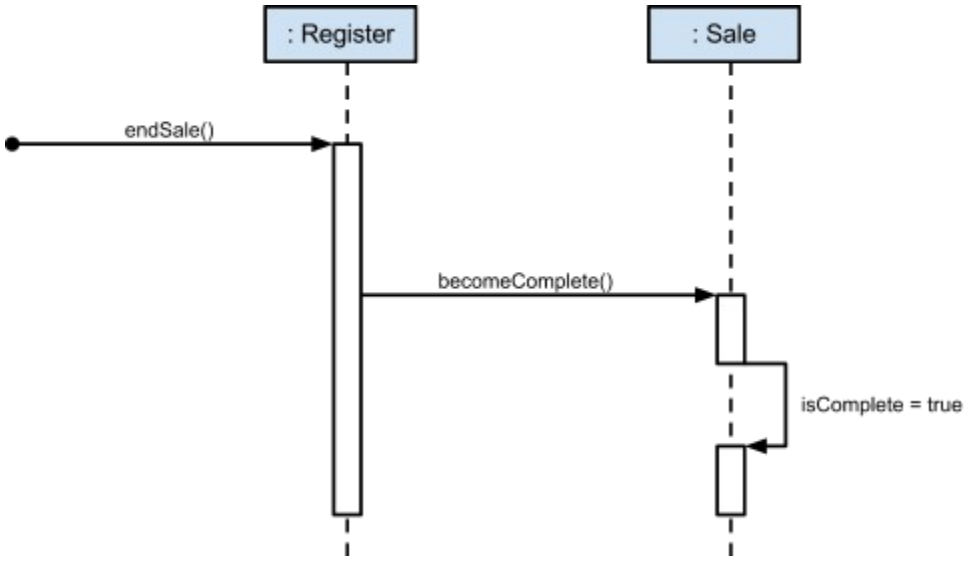
The subclasses Cash Payment, Credit Payment and Check Payment are mutually overlapping with the Payment class as they all deal with the transaction of a payment by the customer. This is to say that we can group these 3 classes as members (or subclasses) of the Payment class, which we will label as the superclass since it is the more general of the classes, to identify their general relationship. Moreover, the Domain Model satisfies the IS-A rule by illustrating the superclass and subclass' set membership conformance due to this membership and how each member has a unique total amount to display, which was inherited from the Payment class. The subclasses represent variations of similar concepts, have similar attributes, in this case the amountTotal attribute, and associations that can be factored out and expressed in the superclass. The subclasses' concepts are similar in that they are responsible for distinguishing and processing different forms of payments chosen by the customer. Furthermore, their association with the superclass is how they are handled differently; compared to Cash Payment, Check Payment and Credit Payment need to be authorized as a precondition before the sale is complete. The 100% rule is satisfied because the superclass' attributes, operations, and associations are available to the subclasses as we can see they are defined methods in the Domain Model.

18) Object Design Models

- **Sequence diagram for makePayment**



- **Sequence diagram for endSale**



19) Design Patterns

- **Factory**

- Added a PaymentFactory class that returns the right Payment Type based off paymentMethod.

```
public class PaymentFactory {
   private int paymentMethod;
   public PaymentFactory(int method){
       this.paymentMethod = method;
   }
   public Payment getPayment(Money amount) { //Factory Method
       if(this.paymentMethod == 1){
           return new CreditPayment(amount);
       }
       else if(this.paymentMethod == 2){
           return new CheckPayment(amount);
       }
       else {
           return new CashPayment(amount);
       }
   }
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/PaymentFactory.java

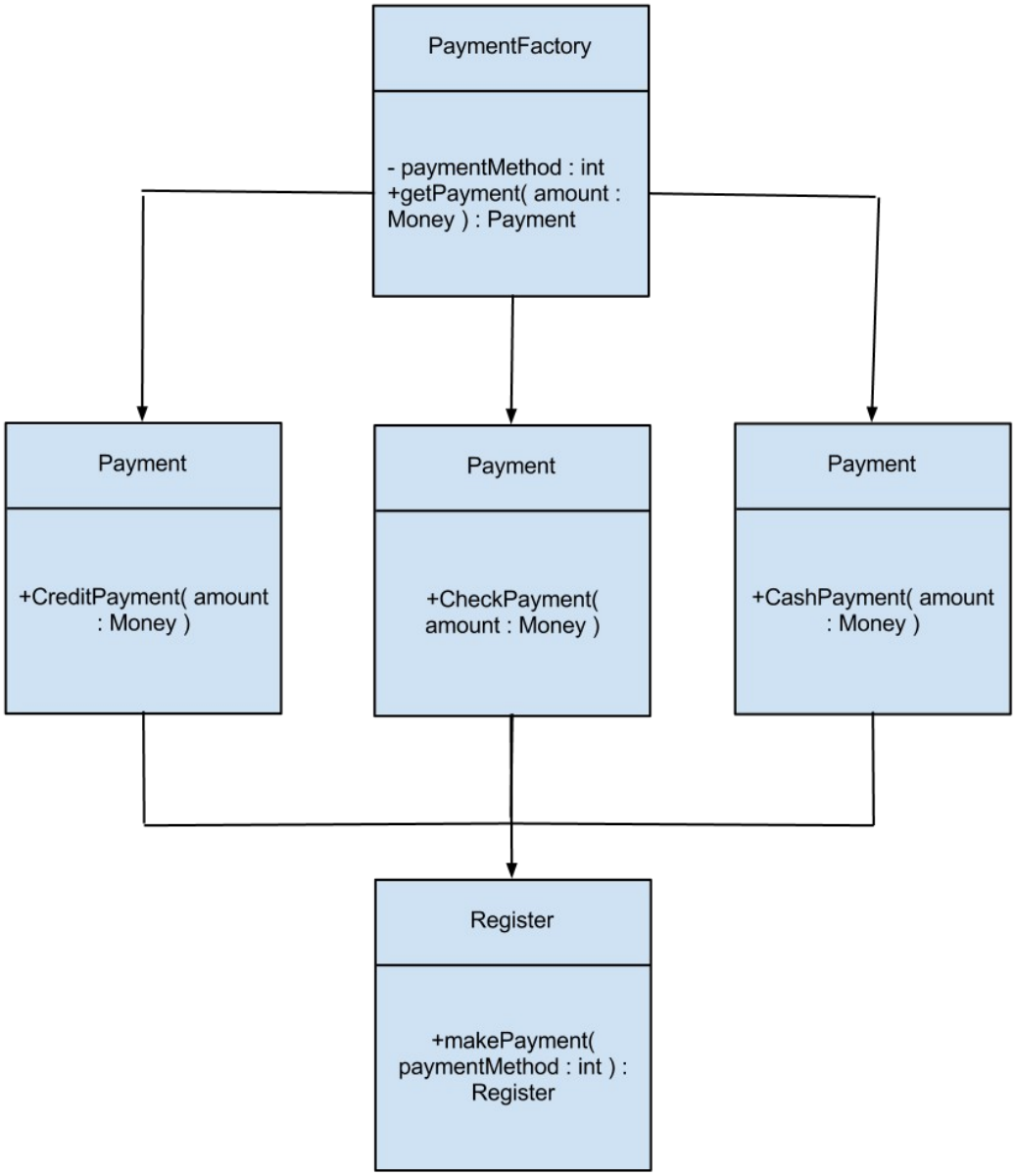
- A Payment Factory object is created inside the Register Class in the makePayment method and it returns the correct Payment type between CashPayment, CheckPayment, or CreditPayment.

```
public void makePayment(int paymentMethod){
       PaymentFactory factoryObj = new PaymentFactory(paymentMethod);
       Payment payment = factoryObj.getPayment(this.paymentAmount);
       currentSale.makePayment(payment);
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Register.java

- Here is a .diff file for what changes were required in order to implement the factory design pattern:

https://github.com/miro2005/CS462-Project/commit/fc315b25f3176e6582ff7b30e25271eb256ccdaf



Here we have created a factory class called PaymentFactory with a factory method called getPayment. This method returns the right payment type, either CreditPayment, CheckPayment or CashPayment from the Payment class. In the Register class makePayment creates an object of the PaymentFactory class, which returns the payment type.

- **Singleton**

- Modified the Register class to be a singleton by changing the constructor to be private with a getRegister method to ensure that only one instance of a register object is active at one time.

```
private static Register theInstance = null;
private Register(int salesNumber, ProductCatalog pc, int id, String addr, String name){
       this.catalog = pc;
       this.storeID = id;
       this.storeAddr = addr;
       this.storeName = name;
       this.currentSalesNumber = salesNumber;
   }
   //HW5.1 Singleton Design Pattern
   public static Register getRegister(int salesNumber, ProductCatalog pc, int id,
     String addr, String name){
       if(theInstance == null){
           theInstance = new Register(salesNumber, pc, id, addr, name);
       }
       return theInstance;
   }
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Register.java

- Here is a .diff file for what changes were required in order to implement the singleton design pattern for the Register class:

https://github.com/miro2005/CS462-Project/commit/02b356801c3dcb2795848b596297acf2cbfd9f6a

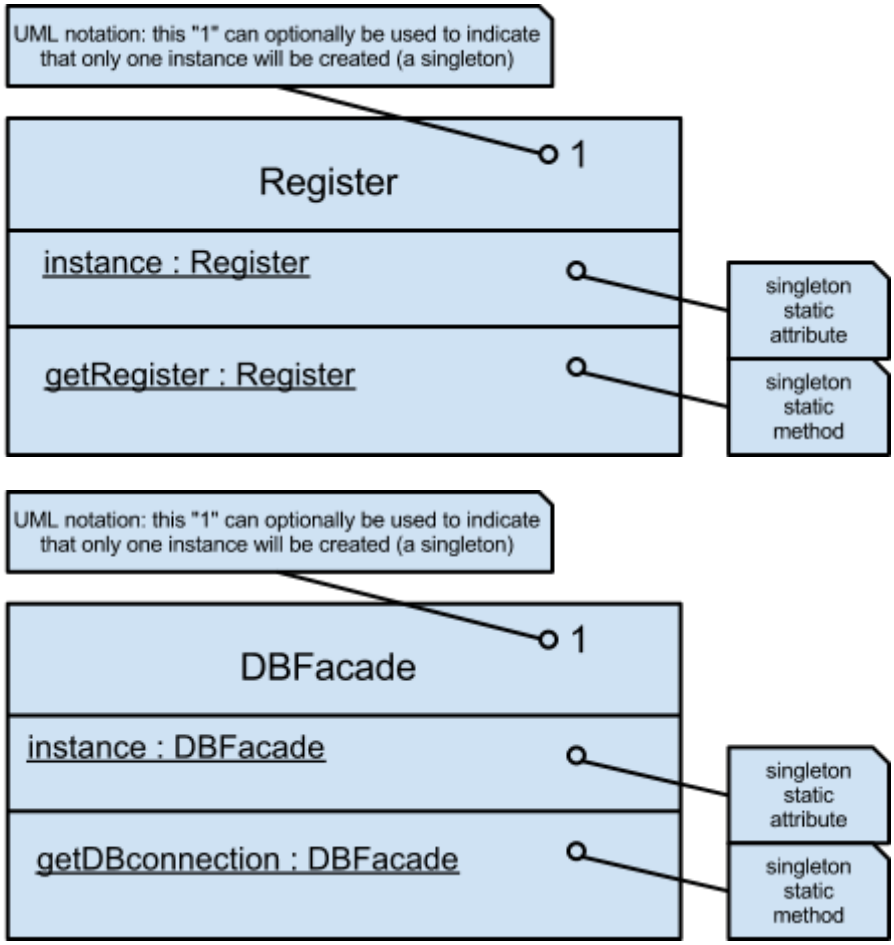
- A second application of singleton is present in the DBFacade class:

```
private static DBFacade instance = null;
private DBFacade() throws SQLException{
    try {
        Class.forName("com.mysql.jdbc.Driver");
        this.con = DriverManager.getConnection("jdbc:mysql://localhost/NewGenPOS",
          "root", "cs462");
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(DBFacade.class.getName()).log(Level.SEVERE, null, ex);
    }
}

public static synchronized DBFacade getDBconnection(){
    if(instance == null){
        try {
            instance = new DBFacade();
        } catch (SQLException ex) {
            Logger.getLogger(DBFacade.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    return instance;
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/DBFacade.java

- The getDBconnection method ensures that only one instance of a connection to the database exists at one time.



The diagram illustrates how only one instance of the Register or DBFacade classes can be created in order to abide by the Singleton design pattern. In the Register class this is to assure that one object is active at a single time or in DBFacade a single connection is exists as one time.

- **Facade**

- The Facade design pattern was implemented by creating a DBFacade class that is used for all communication with the database and since it is also a singleton ensures that only one connection to the database exists at one time. Methods include select, insert, and update items in the database.

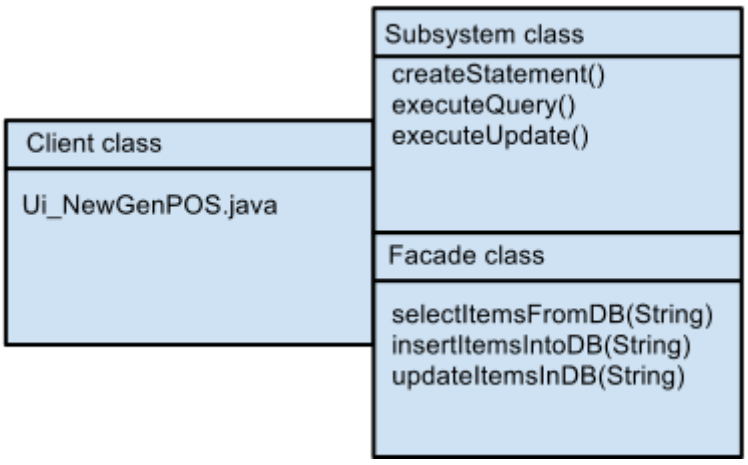
```
public ResultSet selectItemsFromDB(String sqlStmt) throws SQLException{
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(sqlStmt);
    return rs;
}
public void insertItemsIntoDB(String sqlStmt) throws SQLException{
    Statement st = con.createStatement();
    st.executeUpdate(sqlStmt);
}
public void updateItemsInDB(String sqlStmt) throws SQLException{
    Statement st = con.createStatement();
    st.executeUpdate(sqlStmt);
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/DBFacade.java

- Here is all changes that were made to implement this design pattern:

https://github.com/miro2005/CS462-Project/commit/a0f07f1b95986213739bdb38fcdc462dbaf5d967

- The majority of the changes in other class consisted of removing duplicated lines of code to communicate with the database and replaced them with method calls to DBFacade to perform all communication with the database.



This diagram illustrates how a client class such as the user interface can communicate with our facade class, DBFacade. DBFacade is also a Singleton class to make sure that one connection to the database is established. This facade class shows methods that can select, insert and update database items. When talking to the database queries are executed that will retrieve all necessary information for you and return it to the facade and client classes.

- **Adapter**

- The Adapter design pattern was implemented to perform cash, credit, and check validations in order to ensure that all required input is valid.

```
public interface IVerifyPayment {
   public boolean verifyPayment(Money payment, Money total);
   public boolean verifyPayment(Money payment, Money total, String name,
           String addr1, String addr2, String checkNum, String license,
           String phone);
   public boolean verifyPayment(Money payment, Money total, String name,
           String cardNumber, String month, String year);
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/IVerifyPayment.java

- verifyPaymentAdapter is used to call the correct methods from the corresponding classes in order to validate the input from the cash,check, and credit dialog windows.

```
public class verifyPaymentAdapter implements IVerifyPayment{
   private boolean success;
   private verifyCash cash = new verifyCash();
   private verifyCheck check = new verifyCheck();
   private verifyCredit credit = new verifyCredit();

   //Cash Payment
   @Override
   public boolean verifyPayment(Money payment, Money total) {
       success = cash.verifyAmount(payment, total);
       return success;
   }...
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/verifyPaymentAdapter.java

```
public class verifyCash{
   public boolean verifyAmount(Money payment, Money total){...
```

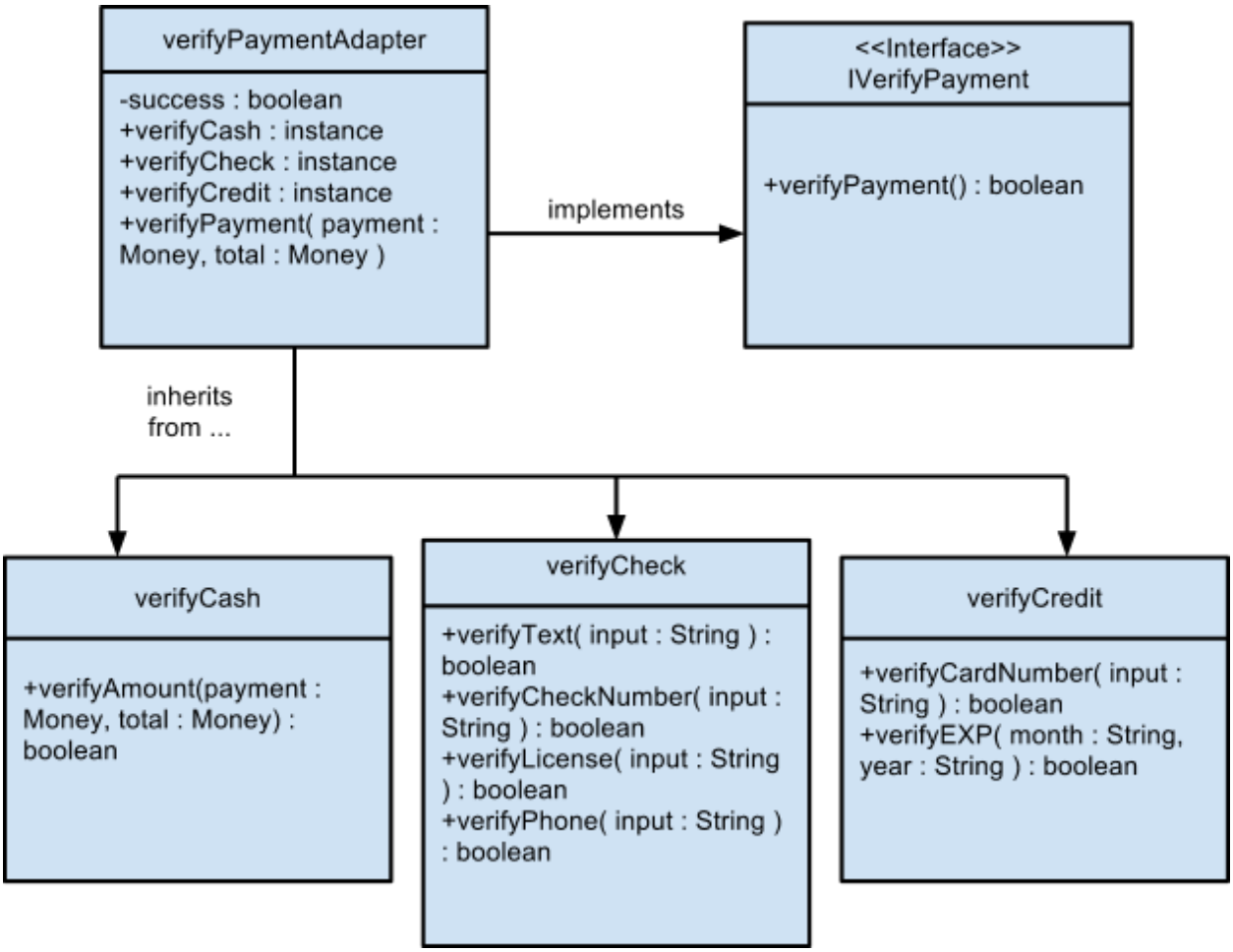
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/verifyCash.java

```
public class verifyCheck {...
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/verifyCheck.java

```
public class verifyCredit {...
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/verifyCredit.java



The Adapter design pattern has a class named verifyPaymentAdapter that implement the IVerifyPayment pattern. The verifyPaymentAdapter calls the methods in the payment type classes and verifies that all required input is valid via the payment verification interface.

- **Strategy**

- The Strategy design pattern was used in order to implement different pricing algorithms. We currently have a standard pricing strategy for a normal sale with 8% sales tax as well as a senior discount pricing strategy that implements a 8% sales tax and a 10% discount. New algorithms can be added just by adding new classes that implements the interface IPricingStrategy:

```
public interface IPricingStrategy {
   Money calcTotal(Money subTotal);
   Money calcDiscount(Money subTotal);
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/IPricingStrategy.java

```
public class StandardPricing implements IPricingStrategy{...
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/StandardPricing.java

```
public class SeniorDiscountPricing implements IPricingStrategy{...
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/SeniorDiscountPricing.java

- The Strategy objects are set inside the Sale class:

```
public void setPricingStrategy(boolean discount){
        IPricingStrategy strategy;
        if(discount){
            strategy = new SeniorDiscountPricing();
        }
        else {
            strategy = new StandardPricing();
        }
        this.pricing.setPricingStrategy(strategy);
}
```

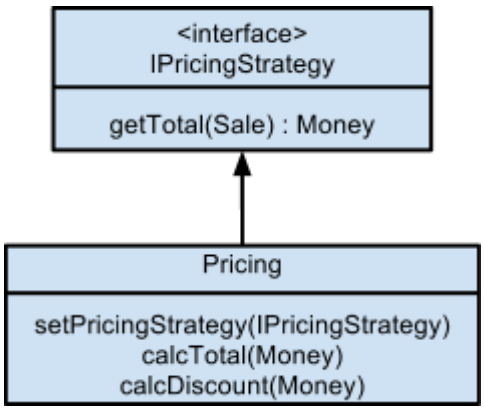
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Sale.java

- The total is calculated in the Sale class by:

```
this.total = this.pricing.calcTotal(this.subTotal);
where pricing is of type Pricing:
public class Pricing {
   private IPricingStrategy currentStrat;

   public Pricing(){
       setPricingStrategy(new StandardPricing());
   }
   public void setPricingStrategy(IPricingStrategy strat){
       currentStrat = strat;
   }
   public Money calcTotal(Money subTotal){
       return currentStrat.calcTotal(subTotal);
   }
   public Money calcDiscount(Money subTotal){
       return currentStrat.calcDiscount(subTotal);
   }
}
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Pricing.java



Applying the Strategy design pattern to our application involved creating a IPrivingStrategy interface that allows for new algorithms to add in discounts or other tax rates. When a class implements this interface, as can be seen above with the Pricing class, it can set individual totals and discounts, which Strategy objects are set inside the Sale class where it calculates the new sales transaction total.

- **Observer**

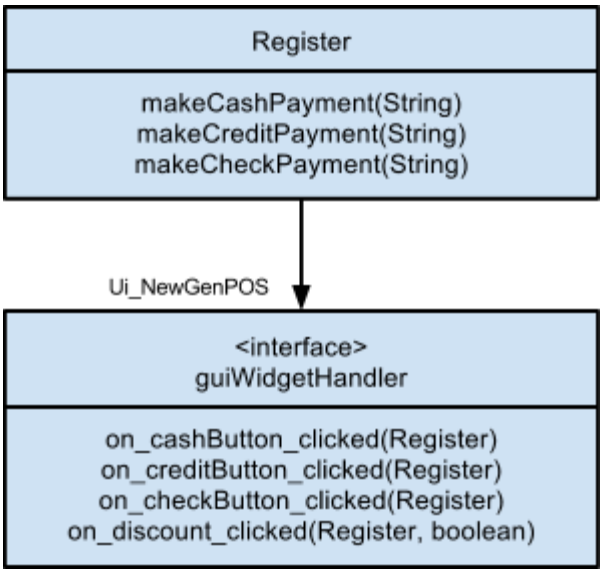
- The Observer design pattern was implemented by adding the guiWidgetHandler class to handle all widgets and propagate commands to the Register or Sale classes. It also handles button presses from the mainWindow and initializes dialog windows or requests updates from the Sale or Register classes. Lastly, it gathers all input from these dialog windows and propagates this data to the Register class to be processed.

```
public class guiWidgetHandler{...
public void on_cashButton_clicked(Register inRegister) {
        dialog = new QDialog();
        UICashdialog = new Ui_CashDialog();
        UICashdialog.setupUi(dialog);
        dialog.setWindowTitle("Enter Cash Amount:");
        dialog.show();
...

public void on_creditButton_clicked(Register inRegister) {...
public void on_checkButton_clicked(Register inRegister) {...
public void on_discount_clicked(Register inRegister, boolean discount){...
...
```

https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/guiWidgetHandler.java

- Changes required to implement this included taking out widget initializations from the Register class and using this 3rd party class to communicate between the UI and the Register and Sale classes.



This diagram shows an example of how the guiWidgetHandler class handles the widget s and propagates commands to the Register class. The Register classes implements the guiWidgetHandler interface which holds methods that communicates with the user interface. When the user clicks one of the payment type buttons then that input data is propagated through to the Register class to be processed.

20) Implement Class Invariant/Operation Contract from DCD

Register class (Full code is available at: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Register.java)

```
class Register {...
...
    public void endSale(){
        currentSale.becomeComplete();
        Ui_NewGenPOS.setText("Thank You for Shopping!");
        Ui_NewGenPOS.clearProductInput();
        Ui_NewGenPOS.clearCart();
        Ui_NewGenPOS.clearDiscount();
        Ui_NewGenPOS.setDisplay(0);
        this.currentSalesNumber = this.currentSalesNumber+1;
    }
...
    public void enterItem(ItemID ItemID, int qty)throws SQLException{
        description = catalog.getProductDescription(ItemID, qty);
        //No such ItemID or out of stock returns description = null
        if(description == null){
            Ui_NewGenPOS.clearProductInput();
        }
        else
        {
            currentSale.makeLineItem(description, qty);
        }
    }
...
    public void makeNewSale(){
        currentSale = new Sale();
    }
    public void makePayment(int paymentMethod){
        PaymentFactory factoryObj = new PaymentFactory(paymentMethod);
        Payment payment = factoryObj.getPayment(this.paymentAmount);
        currentSale.makePayment(payment);
    }
…
}
```

Operation Contract:
- Class Invariant: All supplied values for cash, check, or credit payment methods are valid (i.e. the amount is of type double, check and credit fields are filled in with correct characters, and the shopping cart is not empty).

- Operation Precondition: The Ui_CashDialog(), Ui_CheckDialog(), or Ui_CreditDialog() has returned with accepted values to the guiWidgetHandler class which has passed control over to the register to perform payment verifications for the supplied values.

```
    public boolean makeCashPayment(int paymentMethod, String input){
        try{
            Double paymentInput = Double.parseDouble(input);
            this.paymentAmount = new Money(paymentInput);
            this.total = currentSale.getTotal();
        }
        catch(NumberFormatException e){
            Ui_NewGenPOS.setText("Payment amount must contain ONLY numbers and must NOT be
blank! Try Again!");
            return false;
        }
        if(this.total.checkEquals(new Money(0))){
            Ui_NewGenPOS.setText("Cart is empty, add item and try again!");
            return false;
        }

        //Ensure payment is >= the total
        boolean success = adapter.verifyPayment(this.paymentAmount, this.total);
        if(success){
            this.makePayment(paymentMethod);
        }
        return success;

    }
    public boolean makeCreditPayment{...
    public boolean makeCheckPayment{...
```

- Operation Postcondition: If the payment was performed successfully, a value of true is returned, otherwise, a value of false is returned.

21) Implement all classes in DCD

```
public class Store {...

    ...
    public Store(int StoreID, String Address, String Name)throws SQLException{
        this.storeID = StoreID;
        this.address = Address;
        this.name = Name;

        try{
            catalog = new ProductCatalog();
        } catch (Exception e) {
            System.out.println("Database Not Connected, Product IDs will not be found!");
        }
        int salesNumber = 1;
        try{
            salesNumber = catalog.getSalesNumber();
        } catch (Exception e) {
            System.out.println("Database Not Connected, Sales cannot be recorded!");
        }
        register = Register.getRegister(salesNumber, this.catalog, this.storeID,
this.address, this.name);
    }

    public Register getRegister(){
        return register;
    }
}
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Store.java

```
public class ProductCatalog {
    private ProductDescription description;
    private DBFacade DBconnection;

    public ProductCatalog(){
        DBconnection = DBFacade.getDBconnection();
    }
    public ProductDescription getProductDescription(ItemID ItemID, int qty){
        ...
        return description;
    }
...
}
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/ProductCatalog.java

```
class ProductDescription {
    private String description;
    private Money price;
    private ItemID itemID;
    private int stockRemaining;

    public ProductDescription(ItemID ID, String Name, Money Price, int Stock){
        this.itemID = ID;
        this.description = Name;
        this.price = Price;
        this.stockRemaining = Stock;
    }
…(getters)
}
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/ProductDescription.java

See Question #20 for Register class and links.

```
class Sale {
...
    public Sale(){
        this.isComplete = false;
        this.time = new Date();
        this.subTotal = new Money(0);
        this.total = new Money(0);
        this.pricing = new Pricing();
    }
...
public void becomeComplete(){
        isComplete = true;
    }
public void makeLineItem(ProductDescription desc, int qty){
        item = new SalesLineItem(desc, qty);
        Ui_NewGenPOS.addItemToTable(item);
        Ui_NewGenPOS.displayDescription(item);

        this.cart.add(item);
        this.calcSubTotal(item, qty);
        this.calcTotal();

        Ui_NewGenPOS.setDisplay(this.total);
    }
public void makePayment(Payment paymentAmount){
        this.payment = paymentAmount;

        Money payment = this.payment.getAmount();
        this.cashBack = payment.subtract(this.total);
    }
    public void calcSubTotal(SalesLineItem item, int qty){
         Money ItemPrice = item.getPrice();
```

```
        for(int i=0;i<qty;i++){
            this.subTotal = this.subTotal.add(ItemPrice);
        }
    }
...
}
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Sale.java

```
public class SalesLineItem {
    private int quantity;
    private ProductDescription description;
    private Money price;

    public SalesLineItem(ProductDescription desc, int qty) {
        this.quantity = qty;
        this.description = desc;
        this.price = this.description.getPrice();
    }
    public Money getPrice(){
        return price;
    }
    public ProductDescription getDescription(){
        return this.description;
    }
    public int getQty(){
        return this.quantity;
    }
}
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/SalesLineItem.java

```
class Payment {
    private Money amount;
...
}
class CashPayment extends Payment {...
class CheckPayment extends Payment {...
class CreditPayment extends Payment {...
```

Full Code: https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/src/newgenpos/Payment.java

22) Test Cases

We have made test cases in JUnit for five major functions at least from three different classes

in Register.java
- makeCashPayment()
- makeCreditPayment()
- makeCheckPayment()

in Money.java
- calcTotal()

in Sale.java
- calcSubTotal()

Link to see full code of

RegisterTest.java
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/test/newgenpos/RegisterTest.java

MoneyTest.java
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/test/newgenpos/MoneyTest.java

SaleTest.java
https://github.com/miro2005/CS462-Project/blob/master/NewGenPOS/test/newgenpos/SaleTest.java

RegisterTest.java (portion of the code)

```
public class RegisterTest {
   // initialize register
public RegisterTest() {
   }
   /**
   * Test of makeCashPayment method, of class Register.
   */
   @Test
   public void test1MakeCashPayment() {
       Register instance = Register.getRegister(salesNumber, catalog, storeID, address,
         name);
       instance.makeNewSale();
       Sale currentSale = instance.getCurrentSale();
       System.out.println("Testing Method: Register.makeCashPayment");
       String input = "10.00";

       currentSale.setTotal(new Money(1.99));
       boolean expResult = true;
       boolean result = instance.makeCashPayment(0, input);
       assertEquals(expResult, result);
       System.out.println("Test 1 of Register.makeCashPayment passed!");
   }
    ...
   /**
     * Test of makeCreditPayment method, of class Register.
     */
```

```
@Test
public void test1MakeCreditPayment() {
    Register instance = Register.getRegister(salesNumber, catalog, storeID, address,
      name);
    instance.makeNewSale();
    Sale currentSale = instance.getCurrentSale();

    System.out.println("Testing Method: Register.makeCreditPayment");
    String inputAmount = "2";
    String inputCardNumber = "2222222222222222";
    String inputYear = "12";
    String inputMonth = "12";
    String inputName = "Name";

    currentSale.setTotal(new Money(1.99));
    boolean expResult = true;
    boolean result;
    try{
        result = instance.makeCreditPayment(1, inputAmount, inputCardNumber,
          inputYear, inputMonth, inputName);
    }catch(NullPointerException e){ //Catch gui unavailable error
        result = false;
    }
    assertEquals(expResult, result);
    System.out.println("Test 1 of Register.makeCreditPayment passed!");
}
 ...
/**
 * Test of makeCheckPayment method, of class Register.
 */
@Test
public void test1MakeCheckPayment() {
    Register instance = Register.getRegister(salesNumber, catalog, storeID, address,
     name);
    instance.makeNewSale();
    Sale currentSale = instance.getCurrentSale();

    System.out.println("Testing Method: Register.makeCheckPayment");
    String inputAmount = "2";
    String inputName = "Name";
    String inputAddr1 = "Addr1";
    String inputAddr2 = "Addr2";
    String inputCheckNumber = "1";
    String inputLicense = "22222222";
    String inputPhone = "7145552727";

    currentSale.setTotal(new Money(1.99));
    boolean expResult = true;
    boolean result;
    try{
        result = instance.makeCheckPayment(2, inputAmount, inputName, inputAddr1,
          inputAddr2, inputCheckNumber, inputLicense, inputPhone);
    }catch(NullPointerException e){ //Catch gui unavailable error
        result = false;
    }
    assertEquals(expResult, result);
    System.out.println("Test 1 of Register.makeCheckPayment passed!");
}
```

MoneyTest.java (portion of the code)

```
public class MoneyTest {
   /**
   * Test of calcTotal method, of class Money.
   */
   @Test
   public void test1CalcTotal() {
       System.out.println("Testing Method: Money.calcTotal");
       double tax = 1.08;
       Money subTotal = new Money(9.99);
       //The total should be $10.79
       Money expResult = new Money((9.99*1.08));
       Money result = subTotal.calcTotal(tax);
       System.out.println("Result was: "+result.getFormatted());
       System.out.println("Expected Result is: "+expResult.getFormatted());
       assert(result.checkEquals(expResult));
       System.out.println("Test 1 of Money.calcTotal passed!");
   }
}
```

SaleTest.java (portion of the code)

```
public class SaleTest {
   // initialize sale

   public SaleTest() {
   }

   @Before
   public void setUp() {
       this.testProductItemID = new ItemID(111111);
       this.testProductPrice = new Money(9.99);
       this.testProductStock = 99;
       this.testProductDesc = new ProductDescription(this.testProductItemID,
       this.testProductName, this.testProductPrice, this.testProductStock);
   }

   @After
   public void tearDown() {
   }

   /**
   * Test of calcSubTotal method, of class Sale.
   */
   @Test
   public void test1CalcSubTotal() {
       System.out.println("Testing Method: Sale.calcSubTotal");
       int qty = 5;
       SalesLineItem item = new SalesLineItem(this.testProductDesc,qty);
       Sale instance = new Sale();
       instance.calcSubTotal(item, qty);
       //Subtotal should be $49.95
       Money expectedTotal = new Money(49.95);
       assert(instance.getSubTotal().checkEquals(expectedTotal));
       System.out.println("Test 1 of Sale.calcSubTotal passed!");

   }
}
```

23) Code Organization

Classes are organized in individual .java files and are all located at:

https://github.com/miro2005/CS462-Project/tree/master/NewGenPOS/src/newgenpos.

All .java files include the package newgenpos.

List of files/classes:

- DBFacade.java
- IPricingStrategy.java
- IVerifyPayment.java
- ItemID.java
- Main.java
- Money.java
- Payment.java
- PaymentFactory.java
- Pricing.java
- ProductCatalog.java
- ProductDescription.java
- Register.java
- Sale.java
- SalesLineItem.java
- SeniorDiscountPricing.java
- StandardPricing.java
- Store.java
- Ui_AddItem.java
- Ui_CashDialog.java
- Ui_CheckDialog.java
- Ui_CreditDialog.java
- Ui_NewGenPOS.java
- Ui_ReceiptDialog.java
- guiWidgetHandler.java
- verifyCash.java
- verifyCheck.java
- verifyCredit.java
- verifyPaymentAdapter.java

24) Traceability Matrix

Requirements	Design	Implementation	Elements	Test Case
-Cashier start a new sale (Use-case 001)	-Process Sale Domain Model -System Sequence Diagram - CRC models	-ER Diagram for Process Sale Scenario -Architecture Diagram	-Ui_NewGenPOS.java -Sale.java	-SaleTest()
-Cashier enter items (Use-case 001)	-Six Essential Characteristics for Register Class -Operation Contract for enterItem()	-Register Class Defined by JAVA	-Ui_AddItem.java -SalesLineItem.java	
-System present item name and price (Use-case 001) -System present total price with tax (Use-case 001)	-Create User Interface -Pricing Strategy using Strategy Pattern	-ProductDescription Class Defined by JAVA -Test Cases for User Interface	-ProductDescription.java -IPricingStrategy.java -SeniorDiscountPricing.java	-testCalcTotal
-Customer pays and system handles payment (Use-case 001)	-Payment Class Hierarchy -Operation Contract for submitPayment() -Polymorphic payment class	-Sequence Diagram for makePayment() -DCD for payments class	-verifyCash.java -verifyCheck.java -verifyCredit.java -verifyPaymentAdapter.java -Payment.java -PaymentFactory.java	-testMakeCashPayment() -testMakeCreditPayment() -testMakeCheckPayment()
-System logs sales (Use-case 001)	-System Sequence Diagram	-Sequence Diagram for endSale()	-DBFacade.java	
-System prints receipt (Use-case 001)	-System Sequence Diagram	-Ui_ReceiptDialog.java	-Ui_ReceiptDialog.java	

25) Retrospective

- **Major issues**
  Due to our classes being tied together with the Qt framework, slight modifications to the classes were necessary in order to apply the six design patterns to our application. Moreover, we had to assure that our system implemented error handling for when the cashier inputs in unusual or unexpected information into the system. To assure a quality and secure system these measures must be taken into consideration.
- **Accomplishments**
  Successfully implemented the Qt framework to work with our domain objects and SQL server to get our software in working order. All group members contributed in all phases of the software development process in planning, designing, implementing and testing.
- **Lessons learned**
  There are certain trade-offs when working with a GUI framework such as Qt, which caused us to run into some minor problems, but nothing that we could not adapt to figuring out. In addition, when working in a group environment one must assure that every group member contributes to the project, and if problems arise or any one person cannot figure out how to complete a task then consultation between group members would be the suggested action to be taken. If nobody can figure out a problem then come together as a group to formulate a solution..
- **Team confidence**
  Team confidence in working together as a single entity to create a quality product. This includes thoroughly going over the planning, designing, implementing and testing phases together to assure that all areas are covered and all problems sorted out. Trust between group members is also an important factor in creating a quality product as it boosts team confidence both in each other and in the product as well.
- **Needs of improvement**
  Assuring that all group members do their part(s) and to work together as a team.

26) References

- Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Ed., Pearson Education, Prentice-Hall, 2005.
- CS462 Lectures 1-8, HW1-5
- No other references needed.

27) Appendix

- ## User's Guide

The program is written using Java with Qt user interface libraries. All tests are written using JUnit. To compile the source code, a Java IDE with a Java compiler or 'javac' will work.

However, in order to compile the source code, additional libraries are required. These additional libraries are for the Qt user interface. Information about Qt can be found at http://qt-jambi.org/. We are using version 4.5.2 which can be found at http://sourceforge.net/projects/qtjambi/files/4.5.2/. Libraries for win32, win64, mac, linux32, and linux64 can be found at that link.

Once Qt is installed, the POS program will compile and run, but in order for it to perform anything useful, a SQL server must be running on the localhost. The username for the database must be 'root' and the password is 'cs462'. The schema must be called 'NewGenPOS' and must include tables called 'Inventory' and 'Sales'. SQL scripts to initialize the schema and necessary tables including sample products for purchase can be found at https://github.com/miro2005/CS462-Project/blob/master/SetUpInventory.sql and https://github.com/miro2005/CS462-Project/blob/master/SetUpSales.sql.

In order to get Java to connect to the SQL server, an additional library has to be used: 'mysql-connector-java-5.0.8-bin.jar', which can be found at http://dev.mysql.com/downloads/connector/j/5.0.html.

All of our tests are created using JUnit and can be found at: https://github.com/miro2005/CS462-Project/tree/master/NewGenPOS/test/newgenpos. A Java IDE with JUnit support will run through and pass all of these tests.

A folder titled 'dist' will be included in the submission which will include the compiled Java code in a .jar file. Also in this folder will include a lib folder which will include 'qtjambi-4.5.2_01.jar' and 'mysql-connector-java-5.0.8-bin.jar' which should allow the compiled Java to run successfully assuming the above-mentioned database is running.

To run this compiled Java code, one must use the following command on the command line: 'java -jar NewGenPOS.jar' when inside the dist folder. If the database is not available, error messages stating "Connection Refused" or "Communications link failure." The GUI will still launch, but the user will be unable to add any items, since no items will be found from the database.