CSC 868.01 – Spring 2013
Advanced Object Oriented Software Design and Development
Prof. Levine

POST 1

Group 1, Team 2
February 11, 2013

Group members *(Team 2 members in italics)*

Robert (Bob) Bierman
Victor Woeltjen
Jason Lum
Steven Gimeno
Ying Kit Ng (Kent)
Bianca Uy
Kay Choi

1. Output

The following illustrates an execution of the POST 1 program for the given example product catalog and transaction test file.

Input files:

Product catalog *(products.txt)*

```
A000     Hamburger meat (lb)      0005.99
A001     American cheese (lb)     0002.49
A002     Hamburger sauce(gal)     0001.99
A003     Hamburger bun(dozen)     0001.29
XBHH     Paper towels (dozen)     0004.99
```

Transactions file (*transactions.txt)*

```
Sam
A000      3
A001
CASH $50.0

John
A002      10
CREDIT 1234123412341234

Wilfred
A001      3
CHECK $100.0
```

Program output:

Transaction log (*log.txt*); also written to console:

```
STORE NAME
123 Address St.

Sam        2013-02-11 08:00:11
American cheese (lb)        1 @ $2.49        $2.49
Hamburger meat (lb)         3 @ $5.99        $17.97
------
Total: $20.46
Amount Tendered: $50.00
Amount Returned: $29.54

STORE NAME
123 Address St.

John        2013-02-11 08:00:11
Hamburger sauce(gal)        10 @ $1.99        $19.90
------
Total: $19.90
Paid by Credit Card 1234123412341234

STORE NAME
123 Address St.

Wilfred        2013-02-11 08:00:11
American cheese (lb)        3 @ $2.49        $7.47
------
Total: $7.47
Paid by check
```

2. UML Specifications

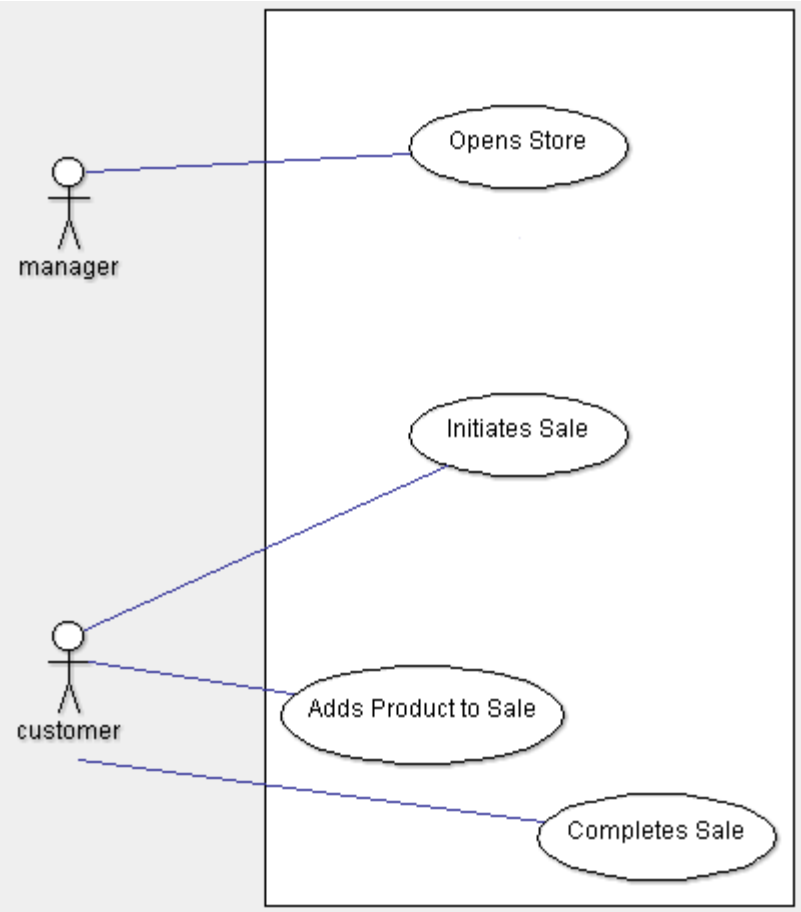
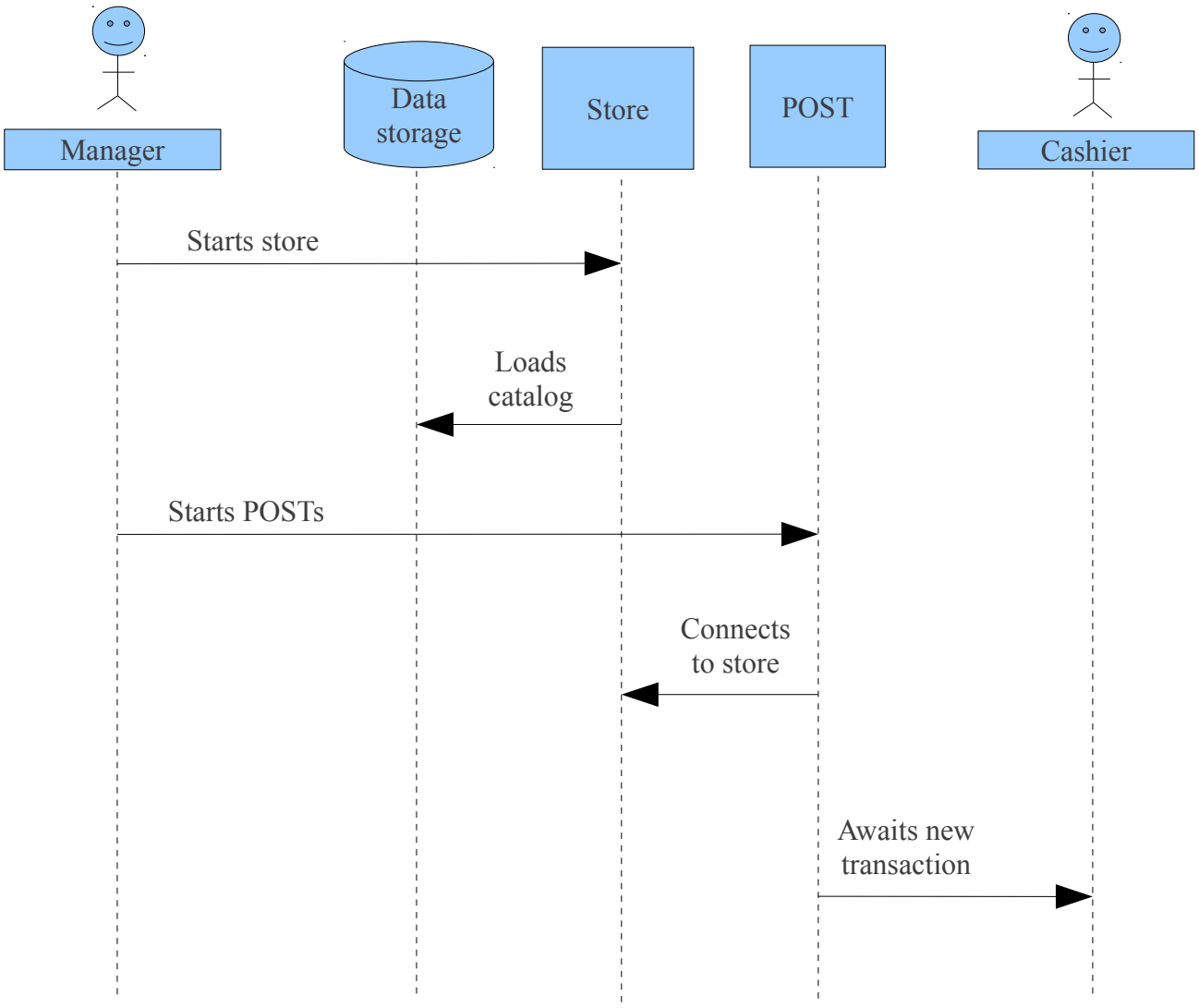


Figure 1: Use Case Overview

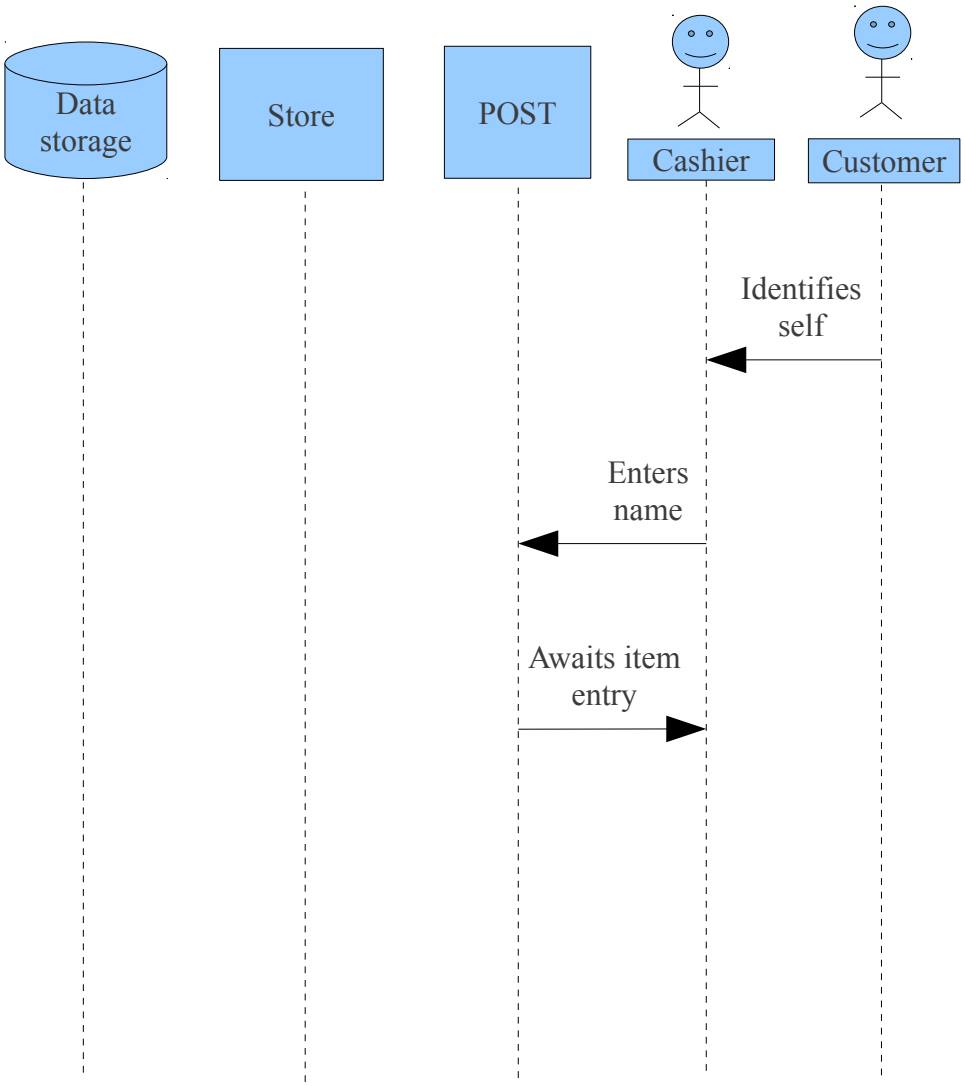
Use case #	1
Use case name	Store opens
Summary	The manager starts systems and opens the store.
Dependency	
Actor	Manager
Precondition	Store is closed.

Description	• Manager starts “store” software (server)
	  ○ Catalog is loaded.
	• Manager starts one or more POSTs.
	  ○ Each POST connects to the “store” (server)

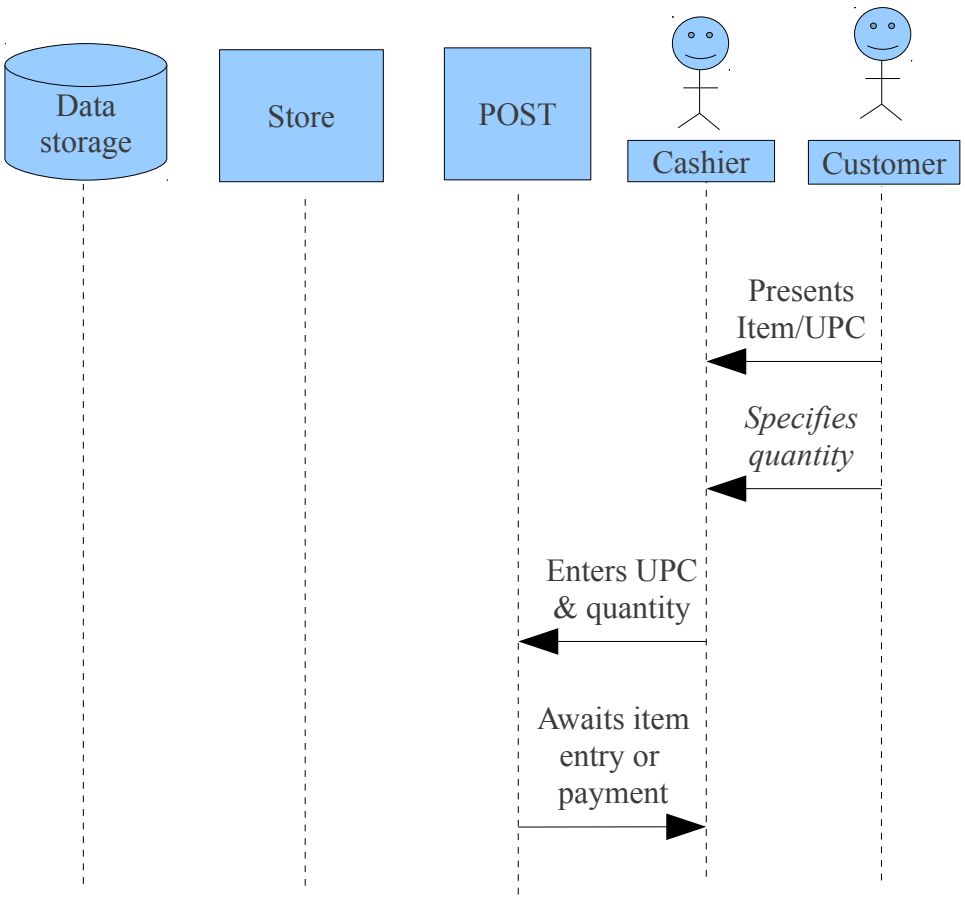
Alternative	
Postcondition	Store is ready to record transactions. POSTs wait for sales to be initiated.



Use case #	2
Use case name	Sale initiation.
Summary	A customer approaches the POS to initiate a sale.
Dependency	Store opens
Actor	Customer
Precondition	The store is open, and no sale is currently in progress.
Description	• The customer identifies him- or herself to the cashier / POST.
Alternative	
Postcondition	POST is ready to continue the sale (by adding products, accepting payment, or abandoning the sale); sale initially includes no products and no amount due.



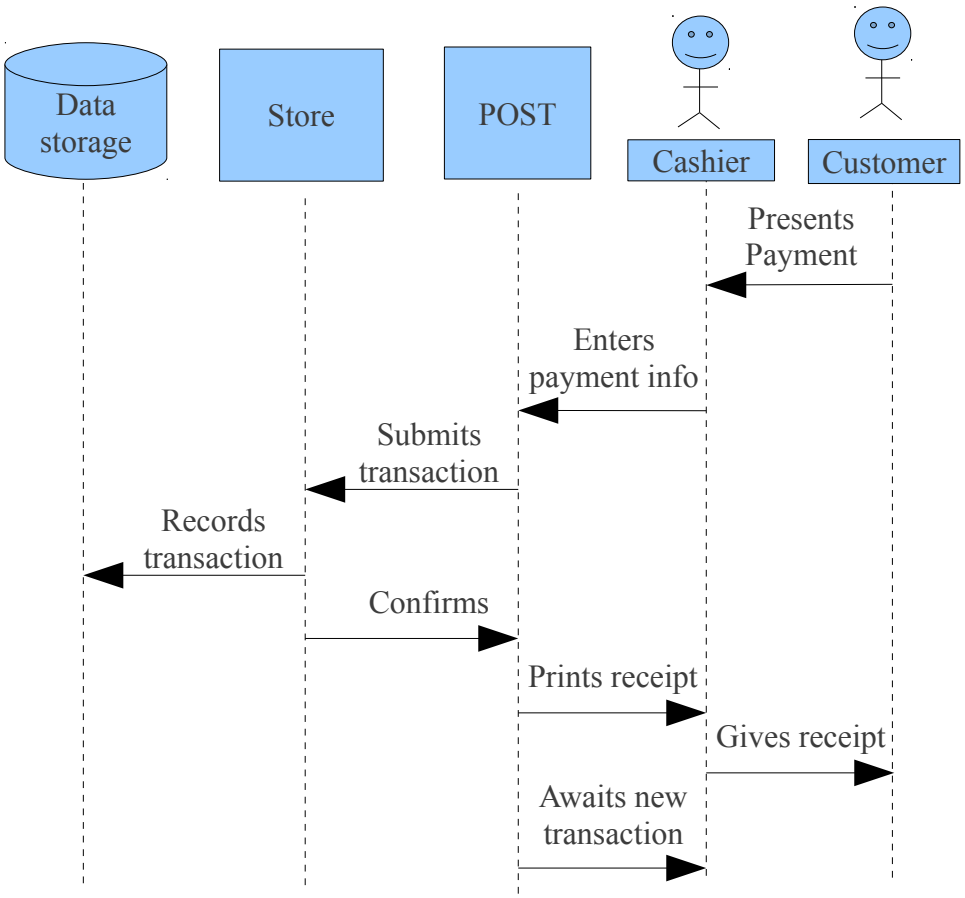
Use case #	3
Use case name	Product addition.
Summary	A customer adds some quantity of product to a sale in progress.
Dependency	Sale initiation
Actor	Customer
Precondition	A sale is currently in progress.
Description	• The customer presents the UPC for the product desired, followed by the quantity of product. • System records the quantity of item as well as the new total amount due as part of the sale in progress.
Alternative	No quantity specified. • Quantity is assumed to be 1. Invalid UPC. • Warning is issued to the POST that the requested UPC is not recognized. • Sale remains in progress, but no changes are made to the products involved or amount due.
Postcondition	Sale remains in progress.



Use case #	4
Use case name	Sale completion.
Summary	Customer issues payment for the products involved in the current sale.
Dependency	Sale initiation; one or more product addition.
Actor	Customer.
Precondition	A sale is currently in progress.

Description	• The customer indicates their preferred payment type.
	  ◦ If cash, the amount of cash is then indicated.
	  ◦ If credit, credit card information is then given.
	  ◦ If check, the check amount is assumed to match amount due.

Alternative	No items have been added to sale. • Sale is abandoned. No transaction is recorded. POST returns to waiting state (no sale in progress). Cash payment amount is less than amount due. • Sale remains open. Amount due is reduced by the amount of cash paid.
Postcondition	Transaction representing the sale is reported to store server and recorded in sales log. A receipt is issued listing all details from the transaction. The sale is closed and the POST returns to its waiting state (no sale in progress).



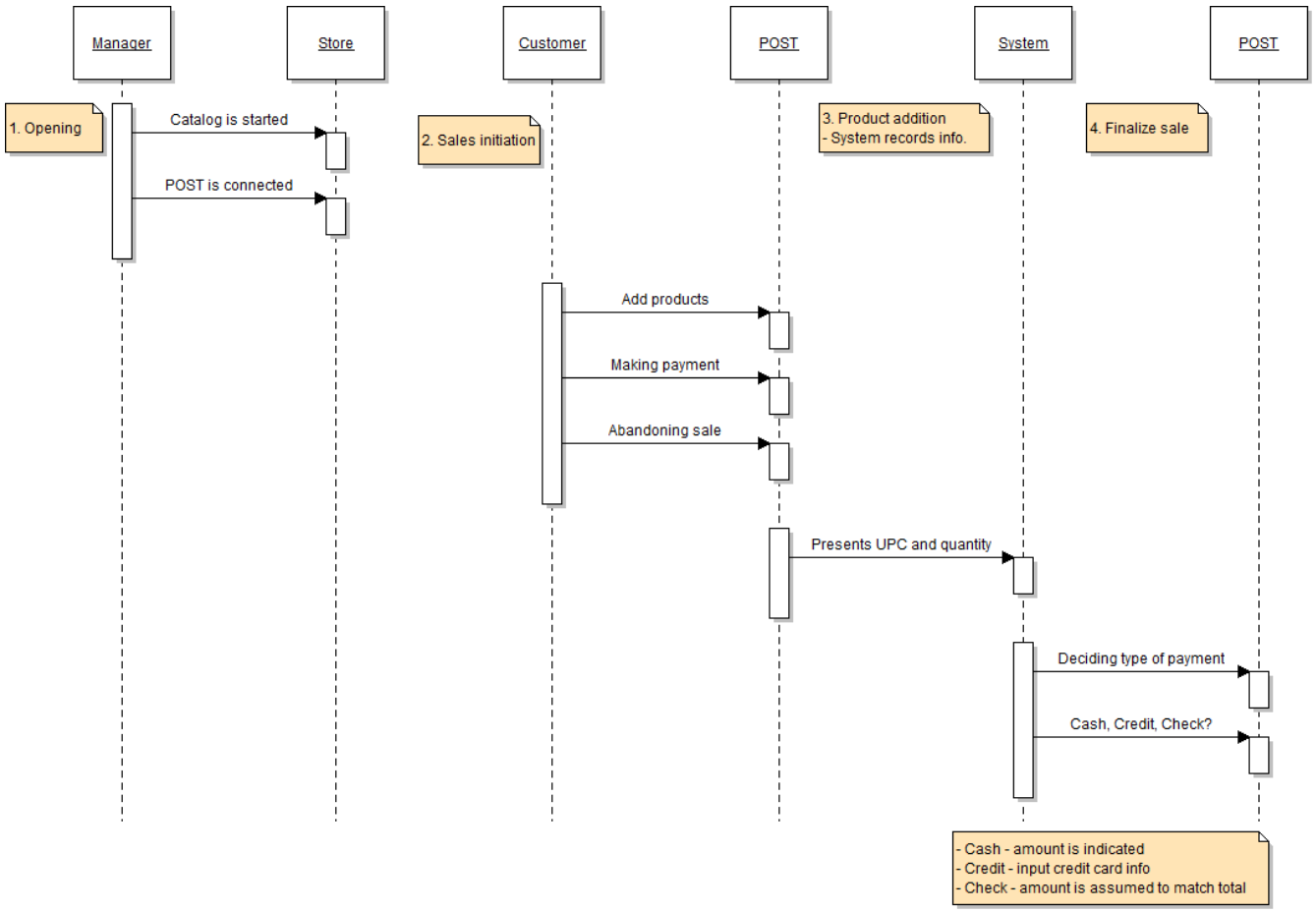


Figure 2: Sequence Diagram (summary)

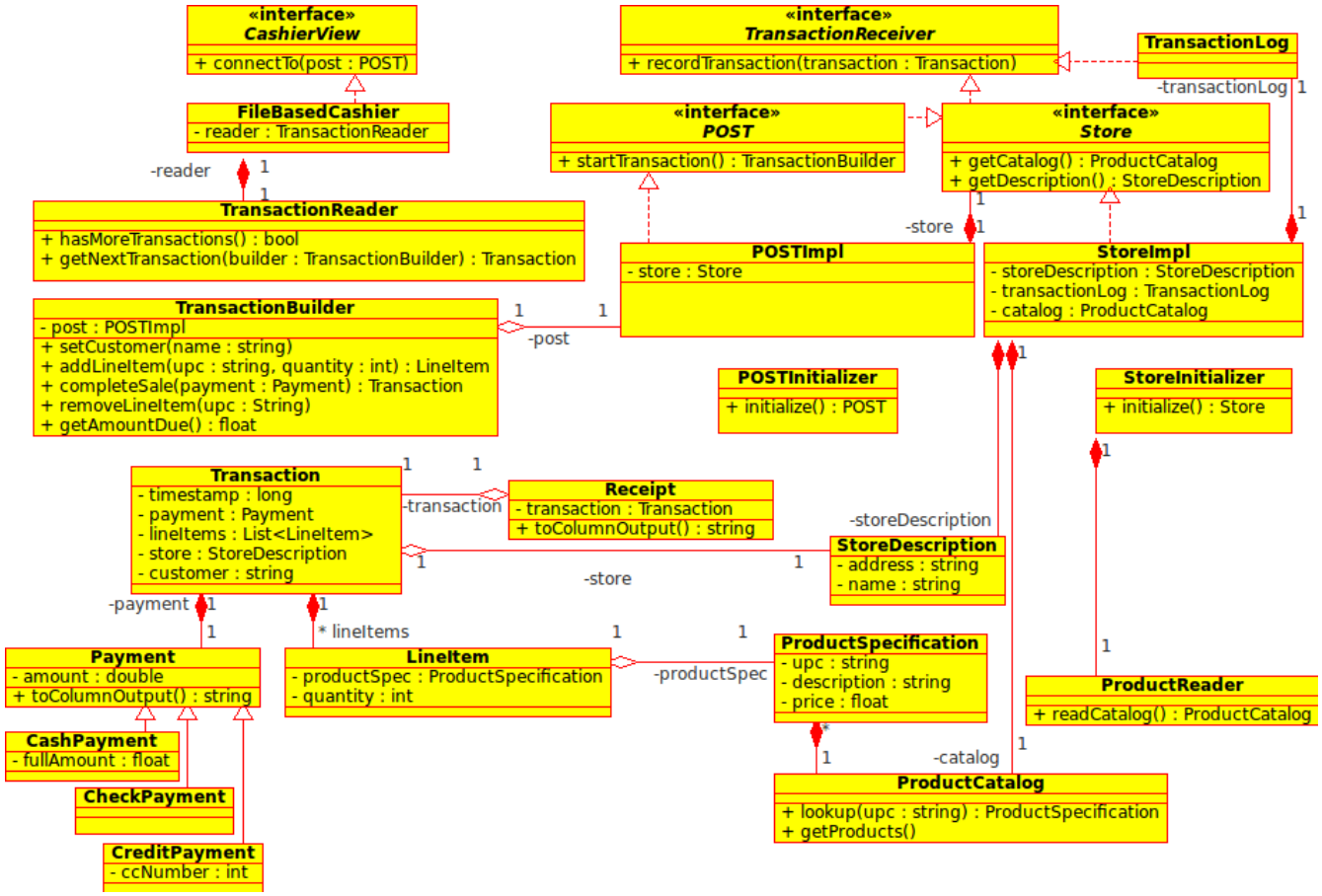


Figure 3: Class Diagram

3. Source code

```
//BEGIN Source code for post/client/view/FileBasedCashier.java

package post.client.view;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import post.client.controller.POST;
import post.model.Receipt;
import post.model.Transaction;

/**
 * A file-based cashier simulates the cashier's activity by reading
 * transactions from a file and issuing them back to the POST.
 *
 * @author woeltjen
 */
public class FileBasedCashier implements CashierView {
    private static final String DEFAULT_TRANSACTIONS = "transaction.txt";
    private TransactionReader reader;

    /**
     * Create a new "Cashier" that reads from a file. This no-argument form
     * will read from the default file name, "transaction.txt"
     * @throws IOException
     */
    public FileBasedCashier() throws IOException {
        this(new File(DEFAULT_TRANSACTIONS));
    }

    /**
     * Create a new "Cashier" that reads from a specified file"
     * @param f the file containing formatted transaction information
     * @throws IOException
     */
    public FileBasedCashier(File f) throws IOException {
        this(new TransactionReader(new FileReader(f)));
    }

    /**
     * Create a new "Cashier" that reads transactions from the
     * specified TransactionReader
     * @param reader the source for reading transactions
     */
    public FileBasedCashier(TransactionReader reader) {
        this.reader = reader;
    }

    /**
     * Connect this CashierView to a POST. For file-based cashier, this
     * will initiate reading from the transaction file via the transaction
     * reader, and delivering results back to the POST.
     * @param post the POST we are connected to
     */
    public void connectTo(POST post) {
        while (reader.hasMoreTransactions()) {
            Transaction t = reader.getNextTransaction(post.startTransaction());
```

```
            Receipt     r = post.recordTransaction(t);
            System.out.println(r.toColumnOutput() + "\n");
        }
    }
}

//END Source code for post/client/view/FileBasedCashier.java

//BEGIN Source code for post/client/view/CashierView.java

package post.client.view;

import post.client.controller.POST;

/**
 * A CashierView is an object that can be connected to a POST, and will
 * typically issue transactions to it.
 * @author woeltjen
 */
public interface CashierView {
    /**
     * Connect this cashier's view to the specified POST. This will allow the
     * view to begin issuing transactions to the POST.
     * @param post
     */
    public void connectTo(POST post);
}

//END Source code for post/client/view/CashierView.java

//BEGIN Source code for post/client/view/TransactionReader.java

package post.client.view;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;
import java.util.List;
import post.client.controller.TransactionBuilder;
import post.model.*;

/**
 * Class TransactionReader
 */
public class TransactionReader {

    private BufferedReader reader;
    private List<String> nextBlock;

    //
    // Constructors
    //
    public TransactionReader(Reader r) {
        reader = new BufferedReader(r);
        prepareBlock();
    }

    /**
```

```
 * @return boolean
 */
public boolean hasMoreTransactions() {
    return !nextBlock.isEmpty() && nextBlock.size() > 2;
}

/**
 * @return Transaction
 * @param builder
 */
public Transaction getNextTransaction(TransactionBuilder builder) {
    builder.setCustomer(nextBlock.get(0).trim());
    for (int i = 1; i < nextBlock.size() - 1; i++) {
        String lineItem = nextBlock.get(i);
        String upc = lineItem.substring(0, 4);
        int quantity = 1;
        if (lineItem.length() > 10) { // ...contains explicit quantity
            quantity = Integer.parseInt(lineItem.substring(10));
        }
        builder.addLineItem(upc, quantity);
    }
    Payment p = parsePayment(
            nextBlock.get(nextBlock.size()-1), builder.getAmountDue());
    prepareBlock();
    return builder.completeSale(p);
}

private Payment parsePayment(String line, float amountDue) {

    int x = 1;

    do{
    try{
    if (line.startsWith("CASH")) {
        float amount = Float.parseFloat(line.substring(line.indexOf('$') + 1));
        return new CashPayment(amount, amountDue);
    } else if (line.startsWith("CREDIT")) {
        long ccNumber = Long.parseLong(line.substring(line.indexOf(' ') + 1));
        return new CreditPayment(amountDue, ccNumber);
    } else if (line.startsWith("CHECK")) {
        return new CheckPayment(amountDue);
    }
    }

    catch(Exception e){
           System.out.println("Error: Specify payment type.");
    }
    }while(x==1);

    return null;
}

private void prepareBlock() {
    nextBlock = new ArrayList<String>();
    String nextLine;
    try {
        while ((nextLine = reader.readLine()) != null) {
            if (nextLine.trim().isEmpty()) return; // Stop after newline
            nextBlock.add(nextLine);
```

```
            }
        } catch (IOException ioe) {
        }
    }
}

//END Source code for post/client/view/TransactionReader.java

//BEGIN Source code for post/client/controller/POST.java

package post.client.controller;

import post.server.controller.Store;

/**
 * A Point of Sale Terminal. Acts as a Store, but additionally allows
 * transactions to be initiated.
 * @author woeltjen
 */
public interface POST extends Store {
    /**
     * Initiate a new transaction. The returned TransactionBuilder object
     * may be used by POST's users to create submittable Transaction objects;
     * this is favored over Views constructing their own transaction, as it
     * allows the POST to inject its own logic during the transaction cycle
     * (for instance, to validate UPCs)
     * @return       TransactionBuilder
     */
    public TransactionBuilder startTransaction(  );
}

//END Source code for post/client/controller/POST.java

//BEGIN Source code for post/client/controller/POSTImpl.java

package post.client.controller;

import post.model.ProductCatalog;
import post.model.Receipt;
import post.model.StoreDescription;
import post.model.Transaction;
import post.server.controller.Store;

/**
 * Class POSTImpl implements the POST interface to instantiate
 * @author Kay Choi
 */
public class POSTImpl implements POST {
    private Store store;

    /**
     * Class constructor.
     * @param store the Store that the POSTImpl belongs to
     */
    public POSTImpl(Store store) {
        this.store = store;
    }

    /**
```

```
     * @return Receipt
     * @param transaction
     */
    @Override
    public Receipt recordTransaction(Transaction transaction) {
        return store.recordTransaction(transaction);
    }

    /**
     * @return ProductCatalog
     */
    @Override
    public ProductCatalog getCatalog() {
        return store.getCatalog();
    }

    /**
     * @return TransactionBuilder
     */
    @Override
    public TransactionBuilder startTransaction() {
        return new TransactionBuilder(this, System.currentTimeMillis());
    }

    @Override
    public StoreDescription getDescription() {
        return store.getDescription();
    }
}

//END Source code for post/client/controller/POSTImpl.java

//BEGIN Source code for post/client/controller/POSTInitializer.java

package post.client.controller;


import post.server.controller.Store;


/**
 *
 * @author Kay Choi
 */
public class POSTInitializer {
    private Store store;

    /**
     * Class constructor.
     * @param store
     */
    public POSTInitializer(Store store) {
        this.store = store;
    }

    /**
     *
     * @return POSTImpl
     */
```

```
    public POST initialize() {
        return new POSTImpl(store);
    }
}

//END Source code for post/client/controller/POSTInitializer.java

//BEGIN Source code for post/client/controller/TransactionBuilder.java

package post.client.controller;

import java.util.HashMap;
import post.model.Payment;
import post.model.ProductSpecification;
import post.model.Transaction;

/**
 * Class TransactionBuilder collects transaction data from a customer and uses
 * it to build a Transaction object.
 * @author Kay Choi
 */
public class TransactionBuilder {

    private HashMap<String, Integer> items;
    private final long time;
    private final POST post;
    private float amountDue;
    private String customer;

    /**
     * Create a new transaction builder. The specified POST is used for
     * validation when needed (for instance, confirming valid UPCs)
     *
     * Note that the timestamp is in milliseconds since the UNIX epoch
     * (January 1, 1970 00:00:00 GMT)
     *
     * @param parent the POST used for validation
     * @param timestamp the time (in ms since UNIX epoch)
     */
    public TransactionBuilder(POST parent, long timestamp) {
        items = new HashMap<String, Integer>();
        time = timestamp;
        amountDue = 0f;
        post = parent;
    }

    /**
     * Set the name of the customer involved in this transaction.
     * @param name the name of the customer
     */
    public void setCustomer(String name) {
        customer = name;
    }

    /**
     * Adds a new item to the transaction. The UPC of the item is stored, along
     * with the quantity. The subtotal is also updated.
     * @return true if the UPC was valid
     * @param upc the UPC of the item being purchased
```

```
     * @param quantity the number of items being purchased
     */
    public boolean addLineItem(String upc, int quantity) {
        ProductSpecification product = post.getCatalog().lookup(upc);

        if(product != null){
            if(items.get(upc) != null) {  //UPC previously entered
                int num = items.get(upc);
                items.put(upc, num + quantity);
            } else {
                items.put(upc, quantity);
            }

            amountDue += quantity * product.getPrice();
            return true;
        } else
            return false;
    }

    /**
     * Removes an item from the transaction.
     * @param upc the UPC of the item
     */
    public void removeLineItem(String upc) {
        items.remove(upc);
    }

    /**
     * Issue payment to complete a transaction. This will return a full
     * Transaction object; note that this transaction is not necessarily
     * recorded to the originating POST.
     * @return Transaction an object representing the processed transaction, if
     *   the payment amount is sufficient
     * @param payment the specific payment for this transaction
     */
    public Transaction completeSale(Payment payment) {
        if(payment.getAmount() >= amountDue)
            return new Transaction(customer, time, payment, items, post);
        else
            return null;
    }

    /**
     * Get the current amount due (total price of all items)
     * @return float the current subtotal
     */
    public float getAmountDue() {
        return amountDue;
    }
}

//END Source code for post/client/controller/TransactionBuilder.java

//BEGIN Source code for post/server/controller/TransactionReceiver.java

package post.server.controller;

import post.model.Receipt;
import post.model.Transaction;
```

```
/**
 * A TransactionReceiver is simply eligible to record transactions. It will
 * also provide Receipt objects to confirm that the transaction has been
 * recorded.
 *
 * @author woeltjen
 */
public interface TransactionReceiver {

    /**
     * Record the specified transaction. Note that this method is responsible
     * for issuing a Receipt object (this may be null in cases where errors
     * have occurred.)
     * @return       Receipt a receipt for the transaction
     * @param        transaction the transaction to record
     */
    public Receipt recordTransaction(Transaction transaction);
}

//END Source code for post/server/controller/TransactionReceiver.java

//BEGIN Source code for post/server/controller/StoreInitializer.java

package post.server.controller;

import java.io.File;
import java.io.FileReader;
import post.model.ProductCatalog;
import post.model.ProductReader;
import post.model.StoreDescription;

/**
 * The StoreInitializer is responsible for setting up the Store.
 *
 * During launch, the StoreInitializer is responsible for generating the
 * Store object that will be used for future interactions. This may include
 * initializing a product catalog and connecting to the transaction log.
 *
 * @author woeltjen
 */
public class StoreInitializer {
    private static final String DEFAULT_STORE_NAME = "STORE NAME";
    private static final String DEFAULT_ADDRESS    = "123 Address St.";

    private static final String DEFAULT_LOG_NAME = "log.txt";
    private static final String DEFAULT_CATALOG_NAME = "products.txt";

    private String logName;
    private String productFileName;

    public StoreInitializer() {
        this(DEFAULT_LOG_NAME, DEFAULT_CATALOG_NAME);
    }

    public StoreInitializer(String logName, String productFileName) {
        this.logName = logName;
        this.productFileName = productFileName;
    }
```

```
    /**
     * Create and set up new Store, connected to the appropriate log files and
     * product catalog.
     * @return
     */
    public Store initialize() {
        try {
            StoreDescription desc =
                    new StoreDescription(DEFAULT_STORE_NAME, DEFAULT_ADDRESS);
            TransactionLog log = new TransactionLog(new File(logName));
            ProductCatalog catalog =
                    new ProductReader(new FileReader(productFileName))
                    .readCatalog();
            return new StoreImpl(desc, log, catalog);
        } catch (Exception e) {
            // TODO: How to log this?
            return null;
        }
    }

//    public static void main(String[] args) {
//        // Test method
//        Store s = new StoreInitializer().initialize();
//        System.out.println(s.getDescription().getName());
//        System.out.println(s.getCatalog().getProducts().size());
//    }
}

//END Source code for post/server/controller/StoreInitializer.java

//BEGIN Source code for post/server/controller/TransactionLog.java

package post.server.controller;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import post.model.Receipt;
import post.model.Transaction;


/**
 * Logs transaction information to a file.
 * @author woeltjen
 */
public class TransactionLog implements TransactionReceiver {
    private Writer log;

    /**
     * Create a new transaction log. Transaction information will be recorded
     * to the specified file.
     * @param f
     * @throws IOException
     */
    public TransactionLog(File f) throws IOException {
        this(new FileWriter(f, true));
    }
```

```
    /**
     * Create a new transation log. Transaction information will be written
     * to the specified stream.
     * @param w
     */
    public TransactionLog(Writer w) {
        log = w;
    }

    @Override
    public Receipt recordTransaction(Transaction transaction) {
        try {
            Receipt r = new Receipt(transaction);
            log.write(r.toColumnOutput() + "\n");
            log.flush();
            return r;
        } catch (Exception e) {
            // TODO: How to handle write errors? Return receipt anyway?
            return null;
        }
    }

//    public static void main(String[] args) {
//        // Test method
//        try {
//            TransactionLog log =
//                    new TransactionLog(new OutputStreamWriter(System.out));
//            final Store store = new StoreImpl(new StoreDescription("T1", "T2"),
log, null);
//            Transaction t = new Transaction(null, System.currentTimeMillis(),
null, null, store) {
//
//                @Override
//                public String getCustomer() {
//                    return "Test customer";
//                }
//
//                @Override
//                public List<LineItem> getLineItems() {
//                    return super.getLineItems();
//                }
//
//                @Override
//                public Payment getPayment() {
//                    return new CreditPayment(14.99f, 1234);
//                }
//
//                @Override
//                public StoreDescription getStore() {
//                    return super.getStore();
//                }
//
//                @Override
//                public long getTimestamp() {
//                    return super.getTimestamp();
//                }
//            };
//            log.recordTransaction(t);
```

```
//        } catch (Exception e) {
//            e.printStackTrace();
//        }
//    }
}

//END Source code for post/server/controller/TransactionLog.java

//BEGIN Source code for post/server/controller/StoreImpl.java

package post.server.controller;

import post.model.ProductCatalog;
import post.model.Receipt;
import post.model.StoreDescription;
import post.model.Transaction;

/**
 * Provides a general implementation of a store.
 *
 * This simply connects the Store's interface to a specific group of relevant
 * objects provided during the constructor call.
 *
 * @author woeltjen
 */
public class StoreImpl implements TransactionReceiver, Store {
    private StoreDescription storeDescription;
    private TransactionLog transactionLog;
    private ProductCatalog catalog;

    /**
     * Create a new store.
     * @param storeDescription a description of the store
     * @param transactionLog the transaction log the store should write to
     * @param catalog a catalog of products available through the store.
     */
    public StoreImpl(StoreDescription storeDescription, TransactionLog
transactionLog, ProductCatalog catalog) {
        this.storeDescription = storeDescription;
        this.transactionLog = transactionLog;
        this.catalog = catalog;
    }

    @Override
    public StoreDescription getDescription() {
        return storeDescription;
    }

    @Override
    public ProductCatalog getCatalog() {
        return catalog;
    }

    @Override
    public Receipt recordTransaction(Transaction transaction) {
        return transactionLog.recordTransaction(transaction);
    }
}
```

```
//END Source code for post/server/controller/StoreImpl.java

//BEGIN Source code for post/server/controller/Store.java

package post.server.controller;

import post.model.ProductCatalog;
import post.model.StoreDescription;

/**
 * Represents a Store. In addition to receiving transactions, a store also
 * provides information about itself (description, product catalog).
 */
public interface Store extends TransactionReceiver {

    /**
     * Get a description of this store
     * @return
     */
    public StoreDescription getDescription();

    /**
     * Get a catalog of products available through this store.
     * @return       ProductCatalog
     */
    public ProductCatalog getCatalog();
}

//END Source code for post/server/controller/Store.java

//BEGIN Source code for post/POST1Main.java

package post;

import post.client.controller.POST;
import post.client.controller.POSTInitializer;
import post.client.view.CashierView;
import post.client.view.FileBasedCashier;
import post.server.controller.Store;
import post.server.controller.StoreInitializer;

/**
 * Main point of entry for POST1. Creates a Store and a POST with default
 * values, then connects a file-based cashier to the POST.
 * @author woeltjen
 */
public class POST1Main {
    public static void main (String[] args) {
        Store store = new StoreInitializer().initialize();
        POST  post  = new POSTInitializer(store).initialize();

        try {
            CashierView view = new FileBasedCashier();
            view.connectTo(post);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
//END Source code for post/POST1Main.java

//BEGIN Source code for post/model/ProductReader.java

package post.model;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;
import java.util.List;

/**
 * Reads in a product catalog from a text stream.
 *
 * @author woeltjen
 */
public class ProductReader {
    private Reader reader;

    /**
     * Create a new ProductReader. This will look at the supplied Reader
     * object in order to find product information.
     * @param r
     */
    public ProductReader(Reader r) {
            this.reader = r;
    }

    /**
     * Read in the product catalog. This produces a ProductCatalog containing
     * ProductSpecifications, which can be used to look up products by UPC
     * and so forth.
     * @return ProductCatalog
     */
    public ProductCatalog readCatalog() throws IOException {
        BufferedReader r = new BufferedReader(reader);
        List<ProductSpecification> products =
            new ArrayList<ProductSpecification>();

        String nextLine;
        while ( (nextLine = r.readLine()) != null) {
            String upc   = nextLine.substring(0, 4);
            String desc  = nextLine.substring(9, 29);
            float  price = Float.parseFloat(nextLine.substring(34));
            products.add(new ProductSpecification(upc, desc, price));
        }

        return new ProductCatalog(products);
    }

//    public static void main (String[] args) {
//        // Test ProductReader
//        Reader r;
//
//        try {
//            r = new FileReader("products.txt");
```

```
//            ProductCatalog cat = new ProductReader(r).readCatalog();
//            System.out.println(cat.lookup("XBHH").getDescription());
//        } catch (Exception e) {
//            e.printStackTrace();
//        }
//    }
}

//END Source code for post/model/ProductReader.java

//BEGIN Source code for post/model/CheckPayment.java

package post.model;

/**
 * Represents payment made by check.
 * @author woeltjen
 */
public class CheckPayment extends Payment {
    public CheckPayment(float amount) {
        super(amount);
    }

    @Override
    public String toColumnOutput() {
        return "Paid by check";
    }
}

//END Source code for post/model/CheckPayment.java

//BEGIN Source code for post/model/LineItem.java

package post.model;


/**
 * A LineItem describes the purchase of a specific product, at a specific
 * quantity.
 *
 * @author woeltjen
 */
public class LineItem {
    private ProductSpecification productSpec;
    private int quantity;

    /**
     * Create a new line item. This includes both the product purchased, and the
     * quantity purchased.
     *
     * @param productSpec
     * @param quantity
     */
    public LineItem(ProductSpecification productSpec, int quantity) {
        this.productSpec = productSpec;
        this.quantity = quantity;
    }

    /**
```

```
     * Get the product involved in this line item.
     *
     * @return the specification for the product purchased.
     */
    public ProductSpecification getProductSpec() {
        return productSpec;
    }

    /**
     * Get the quantity of the item purchased.
     *
     * @return the quantity of the item purchased
     */
    public int getQuantity() {
        return quantity;
    }
}

//END Source code for post/model/LineItem.java

//BEGIN Source code for post/model/CashPayment.java

package post.model;

/**
 * A CashPayment represents payment given by cash, which should equal or
 * exceed the amount due.
 * @auther woeltjen
 */
public class CashPayment extends Payment {
    private float fullAmount;

    /**
     * Create a new cash payment. This includes both the amount offered
     * by the customer, and the amount due (the change received is implicitly
     * the difference between the two.)
     * @param tendered
     * @param due
     */
    public CashPayment(float tendered, float due) {
        super(due);
        fullAmount = tendered;
    }

    @Override
    public String toColumnOutput() {
        return "Amount Tendered: " + formatPrice(fullAmount) + "\n"
                + "Amount Returned: " + formatPrice(fullAmount - getAmount());

    }

    private String formatPrice(float price) {
        return String.format("$%.2f", price);
    }
}

//END Source code for post/model/CashPayment.java

//BEGIN Source code for post/model/StoreDescription.java
```

```
package post.model;

/**
 * Provides a basic description of a specific store.
 * @author woeltjen
 */
public class StoreDescription {
    private String address;
    private String name;

    /**
     * Create a new description for a store.
     * @param name the name of the store
     * @param address the store's address
     */
    public StoreDescription(String name, String address) {
        this.address = address;
        this.name = name;
    }

    /**
     * Get the store's address
     *
     * @return the store's address
     */
    public String getAddress() {
        return address;
    }

    /**
     * Get the store's name
     *
     * @return the store's name
     */
    public String getName() {
        return name;
    }
}

//END Source code for post/model/StoreDescription.java

//BEGIN Source code for post/model/ProductSpecification.java

package post.model;

/**
 * Describes relevant information about a specific product in inventory.
 * @author woeltjen
 */
public class ProductSpecification {
    private String upc;
    private String description;
    private float price;

    /**
     * Create a new product specification
     * @param upc the products 4-digit locator
     * @param description a text description of the product
```

```
     * @param price the products price, in dollars
     */
    public ProductSpecification(String upc, String description, float price) {
        this.upc = upc;
        this.description = description;
        this.price = price;
    }

    /**
     * Get the product's UPC code
     *
     * @return the value of upc
     */
    public String getUpc() {
        return upc;
    }

    /**
     * Get a short human-readable description of the product
     *
     * @return the value of description
     */
    public String getDescription() {
        return description;
    }

    /**
     * Get the price of the product
     *
     * @return the value of price
     */
    public float getPrice() {
        return price;
    }
}

//END Source code for post/model/ProductSpecification.java

//BEGIN Source code for post/model/Transaction.java

package post.model;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import post.server.controller.Store;

/**
 * Represents a completed sale
 * @author woeltjen
 */
public class Transaction {

    private String customer;
    private long timestamp;
    private Payment payment;
    private List<LineItem> lineItems;
    private StoreDescription store;
```

```
    /**
     * Create a new transaction object to describe a sale.
     * @param customer the name of the customer
     * @param timestamp the time of the sale (ms since UNIX epoch)
     * @param payment the payment issued by the customer
     * @param lineItems the products purchased (UPC/quantity pairs)
     * @param store the store from which these objects were purchased
     */
    public Transaction(String customer, long timestamp, Payment payment,
            Map<String, Integer> lineItems, Store store) {
        this.customer = customer;
        this.timestamp = timestamp;
        this.payment = payment;
        this.store = store.getDescription();
        this.lineItems = new ArrayList<LineItem>();

        for (Entry<String, Integer> entry : lineItems.entrySet()) {
            this.lineItems.add(new
LineItem(store.getCatalog().lookup(entry.getKey()), entry.getValue()));
        }
    }


    /**
     * Get the time of purchase. This is in milliseconds since the UNIX epoch
     * (January 1, 1970 00:00:00)
     * @return the time at which the sale occurred
     */
    public long getTimestamp() {
        return timestamp;
    }

    /**
     * Get the payment issued for this sale
     * @return the payment
     */
    public Payment getPayment() {
        return payment;
    }

    /**
     * Get all LineItems for this purchase. Each line item represents an
     * individual product purchased at some quantity.
     * @return the items purchased
     */
    public List<LineItem> getLineItems() {
        return lineItems;
    }

    /**
     * Get a description of the store where the sale was made.
     * @return the store description
     */
    public StoreDescription getStore() {
        return store;
    }

    /**
```

```
     * Get the name of the customer involved in the sale
     * @return
     */
    public String getCustomer() {
        return customer;
    }
}

//END Source code for post/model/Transaction.java

//BEGIN Source code for post/model/Payment.java

package post.model;

/**
 * A Payment represents monetary value exchanged for products.
 *
 * Payment is an abstract class. Specific payment forms are represented as
 * subclasses.
 */
public abstract class Payment {

    private float amount;

    /**
     * Create a payment for the specified amount due.
     *
     * @param amount
     */
    public Payment(float amount) {
        this.amount = amount;
    }

    /**
     * Get the amount paid.
     *
     * @return the amount paid
     */
    public float getAmount() {
        return amount;
    }

    /**
     * Describe this payment in a manner appropriate for display on a
     * receipt or in a transaction log.
     * @return
     */
    public abstract String toColumnOutput();
}

//END Source code for post/model/Payment.java

//BEGIN Source code for post/model/CreditPayment.java

package post.model;

/**
 * Represents payment made by credit card.
 * @author woeltjen
```

```
 */
public class CreditPayment extends Payment {
    private long ccNumber;

    /**
     * Create a new object representing a payment of
     * the specified amount, using the specified credit card number.
     * @param amount
     * @param ccNumber
     */
    public CreditPayment(float amount, long ccNumber) {
        super(amount);
        this.ccNumber = ccNumber;
    }

    /**
     * Get the value of ccNumber
     *
     * @return the value of ccNumber
     */
    public long getCcNumber() {
        return ccNumber;
    }

    @Override
    public String toColumnOutput() {
        return "Paid by Credit Card " + ccNumber;
    }
}

//END Source code for post/model/CreditPayment.java

//BEGIN Source code for post/model/ProductCatalog.java

package post.model;

import java.util.*;

/**
 * A catalog of all available products within the store. Useful for identifying
 * products by UPC, or for getting a full listing of products available.
 * @author woeltjen
 */
public class ProductCatalog {
    private Map<String, ProductSpecification> productMap =
            new HashMap<String, ProductSpecification>();

    /**
     * Create a new product catalog for the specified group of products.
     * @param products the products available
     */
    public ProductCatalog(Collection<ProductSpecification> products) {
        for (ProductSpecification p : products) {
            productMap.put(p.getUpc(), p);
        }
    }

    /**
     * Look up a product by its UPC. Note that the return value may be
```

```
     * null if there is no product with the specified UPC in this catalog.
     * @return ProductSpecification
     * @param upc
     */
    public ProductSpecification lookup(String upc) {
        return productMap.get(upc);
    }

    /**
     * Get a list of all products available in this catalog.
     * @return List<ProductSpecification>
     */
    public List<ProductSpecification> getProducts() {
        List<ProductSpecification> copy = new ArrayList<ProductSpecification>();
        copy.addAll(productMap.values());
        return copy;
    }
}

//END Source code for post/model/ProductCatalog.java

//BEGIN Source code for post/model/Receipt.java

package post.model;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
 * Create a receipt for the specified transaction. This is primarily to
 * provide formatted text output for the transaction.
 *
 * @author woeltjen
 */
public class Receipt {
    private static final DateFormat DATE_FORMAT =
            new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

    private Transaction transaction;

    /**
     * Create a new receipt for the specified transaction.
     * @param t
     */
    public Receipt(Transaction t) {
        transaction = t;
    }

    /**
     * Get a formatted text representation of the transaction described by
     * this receipt.
     * @return       String
     */
    public String toColumnOutput() {
        Calendar cal = new GregorianCalendar();
        cal.setTimeInMillis(transaction.getTimestamp());
```

```
        String output = "";
        output += transaction.getStore().getName() + "\n";
        output += transaction.getStore().getAddress() + "\n";
        output += "\n";
        output += transaction.getCustomer() + "\t";
        output += DATE_FORMAT.format(cal.getTime()) + "\n";

        for (LineItem item : transaction.getLineItems()) {
            float itemPrice = item.getProductSpec().getPrice();
            float totalPrice = item.getQuantity() * itemPrice;
            output += item.getProductSpec().getDescription() + "\t";
            output += item.getQuantity() + " @ ";
            output += formatPrice(itemPrice) + "\t";
            output += formatPrice(totalPrice) + "\n";
        }

        output += "------\n";
        output += "Total: " + formatPrice(transaction.getPayment().getAmount());
        output += "\n";
        output += transaction.getPayment().toColumnOutput();

        return output;
    }

    private String formatPrice(float price) {
        return String.format("$%.2f", price);
    }
}

//END Source code for post/model/Receipt.java
```