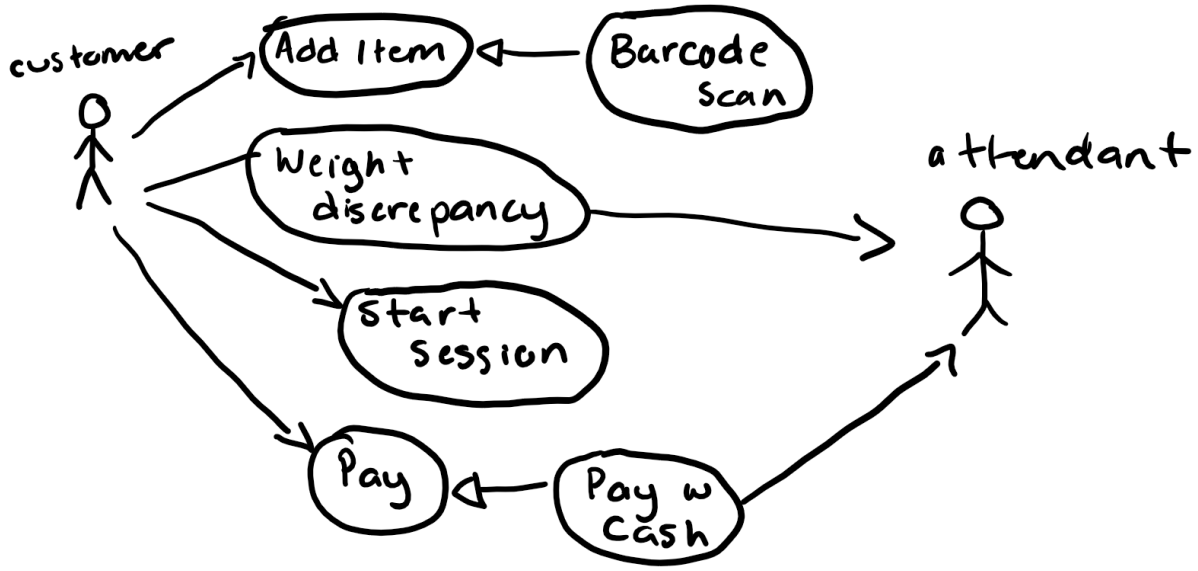
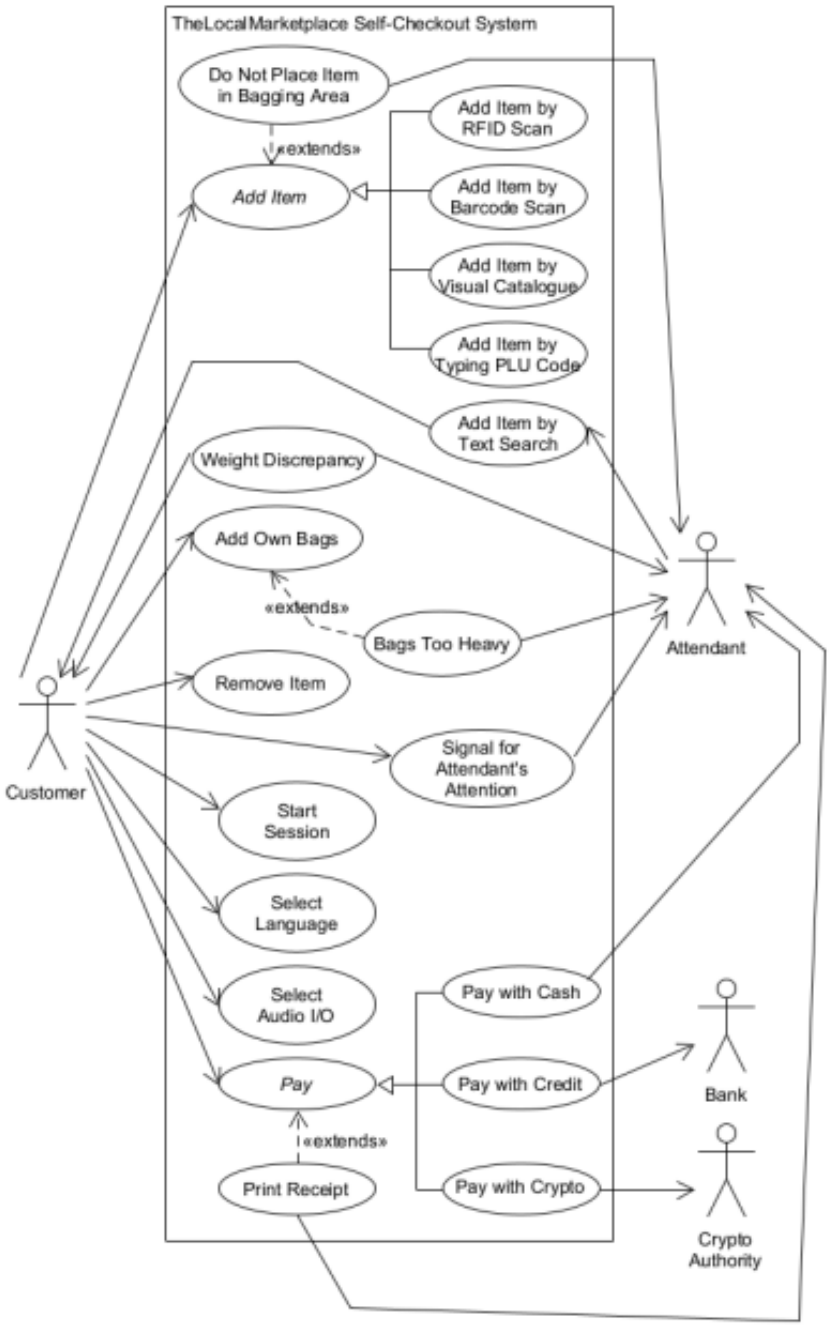
SENG 300
GROUP 23
PROJECT ITERATION 1
IDEAS AND DOCUMENTATION

Robin Bowering UCID 30123373
Matt Gibson UCID 30117091
Kelvin Jamila
Nikki Kim
Hillary Nguyen


customer
Add Item
Barcode Scan
Weight discrepancy
attendant
Start Session
Pay
Pay w Cash

Meeting 1



Provided File

Meeting 1

Goal: to develop a portion of the control software for the self-checkout system, to support its most basic functionality.

- create two Eclipse Java projects, named `com.thelocalmarketplace.software` ("the software project") and `com.thelocalmarketplace.software.test` ("the test project").
    - The software project must depend on the hardware project that is attached to this description.
    - The test project must depend on the software project and the hardware project.
- **must** implement a portion of the control software for the self-checkout system to support the following use cases:
    - Start Session [this will have to be simulated via a method call since you do not have a GUI yet]
    - <V2: clarification>Add Item via Barcode Scan</V2>;
    - Pay via Coin [do not support making change yet, so any overpayment is simply kept];
    - Weight Discrepancy [you need to prevent further instances of Add Item by Scanning and Pay with Coin until the discrepancy is cleared by adding the missing item and/or removing an item that is not part of the order; a more complete implementation will come later].

Use case:	*Start Session*
Primary actor:	Customer.
Goal in context:	To allow the customer to start a new session.
Preconditions:	The system not currently in a session.
Trigger:	The customer wishes to place an order.
Scenario: 1. System: Displays an initial splash screen, with an indication of "Touch Anywhere to Start" or similar message. 2. Customer: Touches the screen. 3. System: Ready for further customer interaction.	
Exceptions: None.	
Priority:	High, essential to system operation.
When available:	First iteration.

- **SimulationException: An exception that can be raised when the behaviour within the simulator makes no sense, typically when it has not been configured correctly.**

Use case:	*Pay with Cash*
Primary actor:	Customer.
Goal in context:	To permit the customer to provide coins and/or banknotes as payment.
Preconditions:	The system has to be ready to take payment.
Trigger:	The customer has indicated that they want to pay cash for their bill.
Scenario: 1. Customer: Inserts coins and/or banknotes in the System. 2. System: Reduces the remaining amount due by the value of the inserted cash. 3. System: Signals to the Customer the updated amount due after the insertion of each coin or banknote. 4. System: If the remaining amount due is greater than 0, go to 1. 5. System: If the remaining amount due is less than 0, dispense the amount of change due. 6. Once payment in full is made and change returned to the customer, see **Print Receipt**.	
Exceptions: 1. If the Customer inserts cash that is deemed unacceptable, this will be returned to the customer without involving the System, presumably handled in hardware. 2. If insufficient change is available, the Attendant should be signalled as to the change still due to the Customer and the station should be suspended so that maintenance can be conducted on it.	
Priority:	Medium, essential in some business contexts, undesired in others.
When available:	Second iteration.
Frequency of use:	Many times per day.
Channel to actor:	Graphical user interface; cash-handling interfaces.
Secondary actors:	Attendant.
Channels to secondary actors: Local-area network; graphical user interface.	
Open issues: 1. The hardware has to handle invalid cash, to reject it without involving the control software. 2. Should mixed modes of payment be supported (e.g., part cash, part crypto)?	

Possible classes needed:
Coin (instances of this class represent individual coins)
CoinOverloadException (represents situations where a component has been overloaded with cash)
CoinDispenserObserver (observes events emanating from a coin dispenser)
CoinSlot (represents a simple coin slot component that has one output channel)
CoinSlotObserver (observes event emanating from a coin slot)
CoinStorageUnit (represents components that store coins)
CoinStorageUnitObserver (observes events emanating from a coin storage unit)
CoinValidator (represents a component for optically and/or physically validating coins)
CoinValidatorObserver (observes events emanating from a coin validator)
ComponentFailure (issued when a component cannot perform its usual functions because it has malfunctioned)
ICoinDispenser (the base type of components that dispense coins)

InvalidSimulationArgumentException (an exception that can be raised when an invalid argument is passed inside the simulation)
InvalidStateSimulationException (an exception that can be raised when an invalid argument is passed inside the simulation)
OneWayChannel<T> (represents a simple device that allows things to move in one direction between other devices)
Sink<T> (represents an abstract component that can receive cash)
Source<T> (simple interface for components that emit cash)

TO DO:
Specify

Use case:	*Add Item by Barcode Scan*
Primary actor:	Customer
Goal in context:	To permit the customer to scan a barcode of an item to add it to their order.
Preconditions:	The system is ready to accept customer input.
Trigger:	The customer wishes to scan a barcode on an item that possesses one.
Scenario: 1. System: Detects a barcode. 2. System: Blocks the self-checkout station from further customer interaction. 3. System: Determines the characteristics (weight and cost) of the product associated with the barcode. 4. System: Updates the expected weight from the bagging area. 5. System: Signals to the Customer to place the scanned item in the bagging area. 6. System: Signals to the System that the weight has changed. 7. System: Unblocks the station.	
Exceptions: 1. The weight in the bagging area does not correspond to expectations; see **Weight Discrepancy**. 2. An item is scanned when a customer session is not in progress. The scanned information shall simply be ignored.	
Priority:	High, must be implemented.
When available:	Iteration 1.
Frequency of use:	Every customer transaction.
Channel to actor:	Hardwired connection.
Secondary actors:	None.
Channels to secondary actors: N/A.	
Open issues: None.	

Add item (Hardware)

Highlighted = Not sure to include

PLU only used for look up items

- BarcodedProduct: Represents products with barcodes
- ElectronicScaleListener: Listens for events emanating from an electronic scale.
- ElectronicScale: Represents electronic scales on which can be placed one or more items.
- PLU codedItem: Represents items for sale, each with a particular price-lookup code and weight
- PLUCodedProduct: Products with price lookup
- PriceLookUpCode: Represents a PLU value, a sequence of digits that, in principle, could lie anywhere in the range 0000-99999.
- Product: Abstract base class for products
- ComponentFailure: Issued when a component cannot perform its usual functions because it has malfunctioned.
- DisabledDevice: Issued when a device cannot perform its usual functions because it has been disabled.

- DisabledException: Issued when a component cannot perform its usual functions because it has been disabled.
- AbstractBarcodeScanner: Abstract base class for barcode scanners.
- Barcode: Represents a barcode value (not the graphic barcode itself).
- BarcodedItem: Represents items for sale, each with a particular barcode and weight.
- BarcodeScanner: A complex device hidden behind a simple simulation.
- BarcodeScannerListener: Listens for events emanating from a barcode scanner.
- IBarcodeScanner: Abstract base type for barcode scanners.
- Mass.MassDifference: Represents the difference between two masses.
- Mass: Represents the mass of an item, measured in micrograms by default.
- Item: Abstract base class of items for sale, each with a particular mass.
- NullPointerSimulationException: An exception that can be raised when a null pointer is passed inside the simulation.
- NoPowerException: Signals that there was an attempt to use a device that was not turned on.
- Product Databases: Represents a cheap and dirty version of a set of databases that the simulation can interact with.

Rough idea

- when customer scans an item
    - try to scan barcode
        - catch any exceptions
        - throw new exception
    - If else
        - Weightdiscrepency == true // replace with method
        - //

      Else
        - weight discrepancy == false // replace with method

```
import com.thelocalmarketplace.hardware.external.ProductDatabases;

public class AddItem {

	public static void addItemViaScan(BarcodedItem item) throws Exception {

		if(Start.hasStarted()==false) {
			throw new Exception();

		}
		// get mass and barcode of scanned item
		double itemMass = item.getMass().inGrams().doubleValue();
		Barcode itemBarcode = item.getBarcode();


		BarcodedProduct product =
ProductDatabases.BARCODED_PRODUCT_DATABASE.get(itemBarcode);

		double expectedMass = product.getExpectedWeight();

		if (expectedMass != itemMass) {
			// do something

		}
		//else keep going







		//exception class


	}


	public static void main(String[] args){
		//addItemViaScan();



	}
```

}

Use case:	*Weight Discrepancy*
Primary actor:	None.
Goal in context:	To react to a difference in the actual weight of the bagging area and the expected weight, due to items being moved improperly into or out of the bagging area, or due to true weights differing from ideal weights or measured weights.
Preconditions:	The system is in the midst of a customer's session.
Trigger:	The System detects a weight change, and it deems that this weight is unacceptably different from expectations.
Scenario: 1. System: Blocks the self-checkout station from further customer interaction. 2. System: Signals to the Customer regarding the weight discrepancy. 3. System: Signals to the Attendant regarding the weight discrepancy. There are three options: a. Customer: Adds or removes an item in response; OR, b. Customer: Signals the System about a do-not-bag request (see **Do Not Place Item in bagging area**); OR, c. Attendant: Signals the System of a weight-discrepancy approval. 4. System: Unblocks the self-checkout station.	
Exceptions: 1. If the attendant does not approve the weight discrepancy, they will need to manually investigate what has been placed in the bagging area and what is supposed to be there. 2. If the weight change from the bagging area still does not concur with expectations, the weight discrepancy will continue.	
Priority:	High, must be implemented for the system to function.
When available:	Second iteration.
Frequency of use:	Potentially constant.
Channel to actor:	N/A.
Secondary actors:	Customer; Attendant.

- PLUCodedItem: items for sale, each with a particular price-lookup code and weight.

- PLUCodedProduct: Represents products with price-lookup (PLU) codes. Such products always have* prices per-kilogram.
- PriceLookUpCode: PLU code used by the above two
- AbstractElectronicScale: Abstract base class of electronic scales on which can be placed one or more items.
- **ElectronicScale: Represents electronic scales on which can be placed one or more items.**
- **ElectronicScaleListener: Listens for events emanating from an electronic scale.**
- **EmptyDevice: Represents the situation when a device is emptied but an attempt is made to remove something from it.**
- **IElectronicScale: Abstract base type of electronic scales on which can be placed one or more items.**
- **IllegalDigitException: Signals that an illegal character has been used where a digit (0-9) was expected.**
- **Item: Abstract base class of items for sale, each with a particular mass.**
- **Mass: Represents the mass of an item, measured in micrograms by default.**
- **Mass.MassDifference: Represents the difference between two masses.**
- **NoPowerException: Signals that there was an attempt to use a device that was not turned on.**
    - **IsOn?**
- **OverloadedDevice: Represents situations where a device has been overloaded, in terms of weight, quantity of items, etc.**
- **Product: Abstract base class for products**
- **ProductDatabases: Represents a cheap and dirty version of a set of databases that the simulation can interact with.**
- **SimulationException: An exception that can be raised when the behavior within the simulator makes no sense, typically when it has not been configured correctly.**

Code plan:

- isBlocked = when true, blocks the entire session
- Notify() = when called, notifies the object that there is a weight discrepancy
    - Common method that can be called by Customer and Attendant
- Need weight discrepancy measure -> electronic scale has mass limit of 100 kg and mass sensitivity of 100mg

Meeting 2

Documentation

Start Session Use Case Order of Operations

-StartSession() is called
-All relevant hardware components are checked
-A SelfCheckoutSession object is instantiated

Add Item Via Barcode Scan Order of Operations

-An item is scanned by the barcode scanner (if it is enabled and powered on)
-Some Scanner Listener (Likely Session object in this case) receives information about barcode, looks up product and updates order with:
Price
Item Name
Expected Weight of the Item
-Scale Controller has Weight Updated
-Add price of product to cart price

Weight discrepancy detected

When expected weight or actual weight change via method call:
-If Expected and Actual mass don't match (margin of scale sensitivity), set weightDiscrepancy = true, calls IsBlocked() on session

Weight discrepancy not detected

-If Expected and Actual mass do match, set weightDiscrepancy = false, calls IsNotBlocked() on session (or IsBlocked as a field that becomes false)

Pay With Coin

- Coin is added
- Coin is validated
- Subtract coin value from total
- Display new total
- Repeat until total <= 0
- Adding coin is the only thing to do, everything else is blocked

Classes needed

- Simulation(?)
- SelfCheckOutController
    - Instantiated hardware
        - CoinSlot coinslot = new CoinSlot;
    - Instantiate Session
- Session
    - Global Variables
        - int totalCartPrice = 0
        - boolean isBlocked = false

        - BigInteger expectedWeight
        - BigInteger actualWeight
        - boolean weightDiscrepancy = false
    - Methods
        - StartSession()
        - Additem(Item item){
            - try { barcodeScanner.scan(item)}
            - Catch {NullPointerException e}
            - Throw new exception
            - Get weight and name of item

          expectedWeight += getWeight(item)
          weightDiscrepancy = true
          isBlocked = trued
            - totalCartPrice += getprice(item)
            - WeightDiscrepancy detected() == true
            -
        - PayWithCoin(int amount)
            - if(IsBlocked){ CoinSlot.IsDisabled()}
        - WeightDiscrepancyDetected()
            - println("Customer Screen: Weight discrepancy has been detected.");
            - println("Attendant Screen: Weight discrepancy has been detected.");
                - How to separate the Customer and Attendant?
            - IsBlocked == true
        - theMassOnTheScaleHasChanged()
            - if expectedWeight != actualWeight
                - weightDiscrepancy = true
                - isBlocked = true
            - if expectedWeight == actualWeight
                - weightDiscrepancy = false
                - isBlocked = false
        -

To Ask:

-Notify Customer/Attendant??