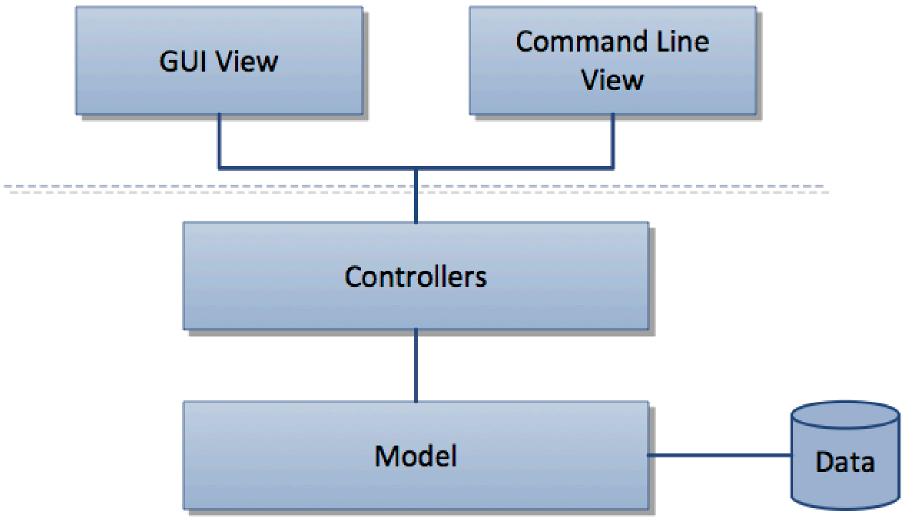
Product Design

Team:	f361-03c : “Team Caribou”

Architectural Model

This diagram represents the major subsystems of the product, which will conform to the model-view-controller design patter. A more specific approach to the MVC architecture used here is given in the Design Rationale section.



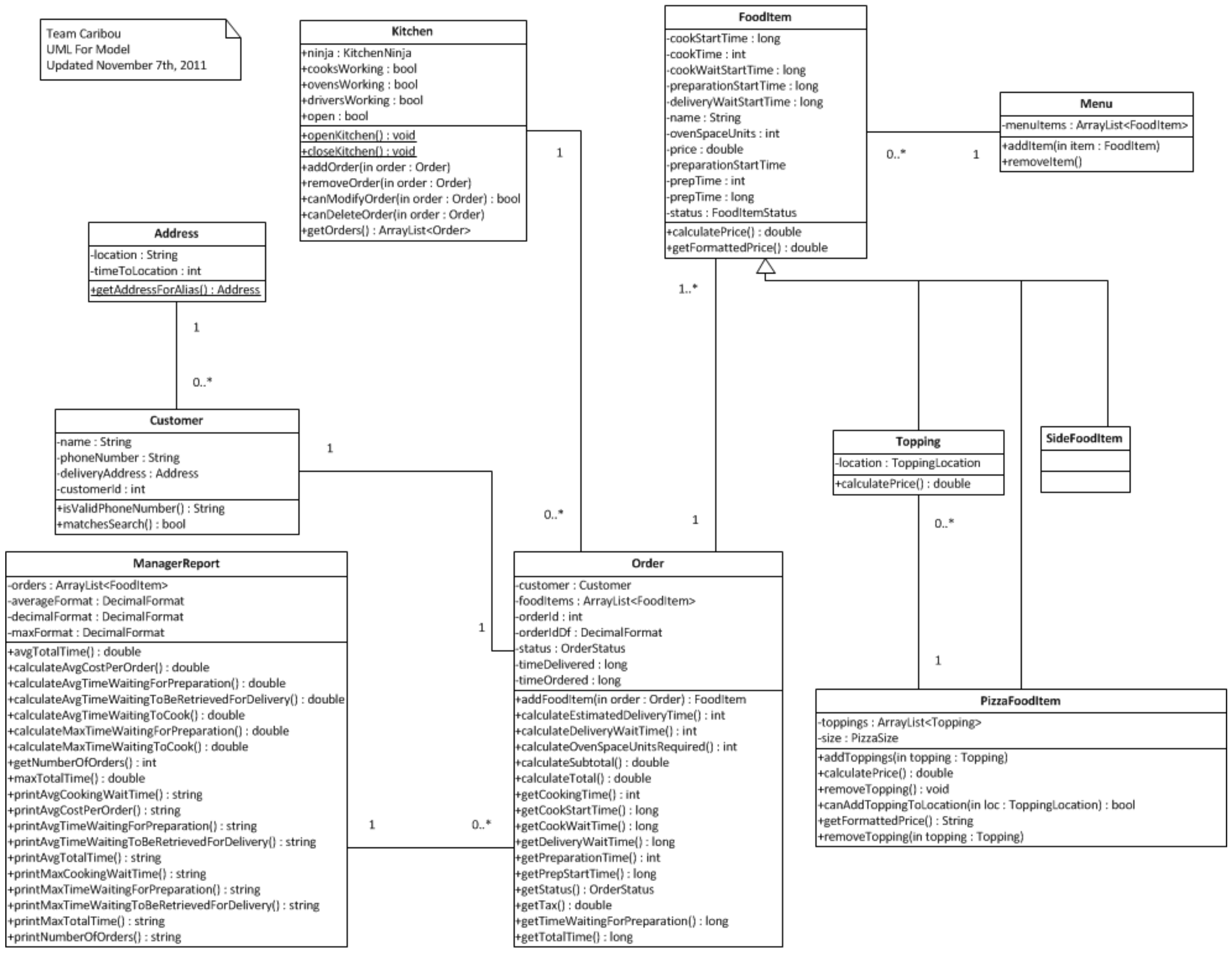
Components and Functions

Order Creation	**State** ● Current Order **Behavior** ● Carries out the process for creating an order based on customer request. ● Forwards completed orders to the kitchen.
Add Pizza	**State** ● Current Pizza **Behavior** ● Creates a pizza based on user input ● Adds pizza(s) to the order
Add Side	**State** ● Current Side **Behavior** ● Creates a side based on user input ● Adds side(s) to the order
Food Menu	**State** ● The list of all available food items. **Behavior** ● Provides access to the menu for the purpose of selecting an item to add to an order. ● Facilitates modification (adding/removing/changing) of menu items.
Customer Profiles	**State** ● The list of all customers including contact info and order histories. **Behavior** ● Provides access to the list of customers (can be searched by phone number and name). ● Facilitates modification (adding/removing/changing) of customers profiles.
User Profiles	**State** ● Profiles of the entities who use the system: cashiers (operators) and managers. **Behavior** ● Facilitates modification (adding/removing/changing) of user profiles.
Manager Reports	**State** ● A list of past orders. **Behavior** ● Provides access to the past orders list and allows filtering by date. ● Aggregates order information into statistical data that can be viewed by the manager.
Order Tracking	**State** ● The state of the kitchen, ovens, and delivery operations. ● The orders that have yet to be delivered (“active orders”). **Behavior** ● Tracks an order as it moves through the kitchen and delivery processes. ● Provides access to the list of active orders showing each order’s progress.
Viewing Current Orders	**State** ● List of the orders still within the kitchen **Behavior** ● Displays the orders within the kitchen

	<ul><li>Gives details about order status and contents</li><li>Allows modification of some of these orders</li></ul>
Manage Kitchen	**State**<ul><li>Number of cooks, ovens, drivers</li><li>Menu of available items</li></ul>**Behavior**<ul><li>Allows modification of kitchen preferences</li></ul>
Edit Users	**State**<ul><li>List of users</li></ul>**Behavior**<ul><li>Ability to edit the users accounts that may log into the system</li></ul>
Viewing Past Orders	**State**<ul><li>List of all past orders</li></ul>**Behavior**<ul><li>Displays details about the orders that have been placed in the system previously</li></ul>
Authentication	**State**<ul><li>Profiles of acceptable users</li></ul>**Behavior**<ul><li>Manages administrator privileges</li><li>Provides security for log in to system</li></ul>
Data Manager	**State**<ul><li>The “filesystem” for the program -- that is, the data that is stored by other components for later use and must be carried over between separate executions of the program.</li></ul>**Behavior**<ul><li>Provides an interface for writing to and reading from the filesystem in a general manner.</li><li>Carries our periodic backups of the filesystem to the actual hard disk.</li></ul>

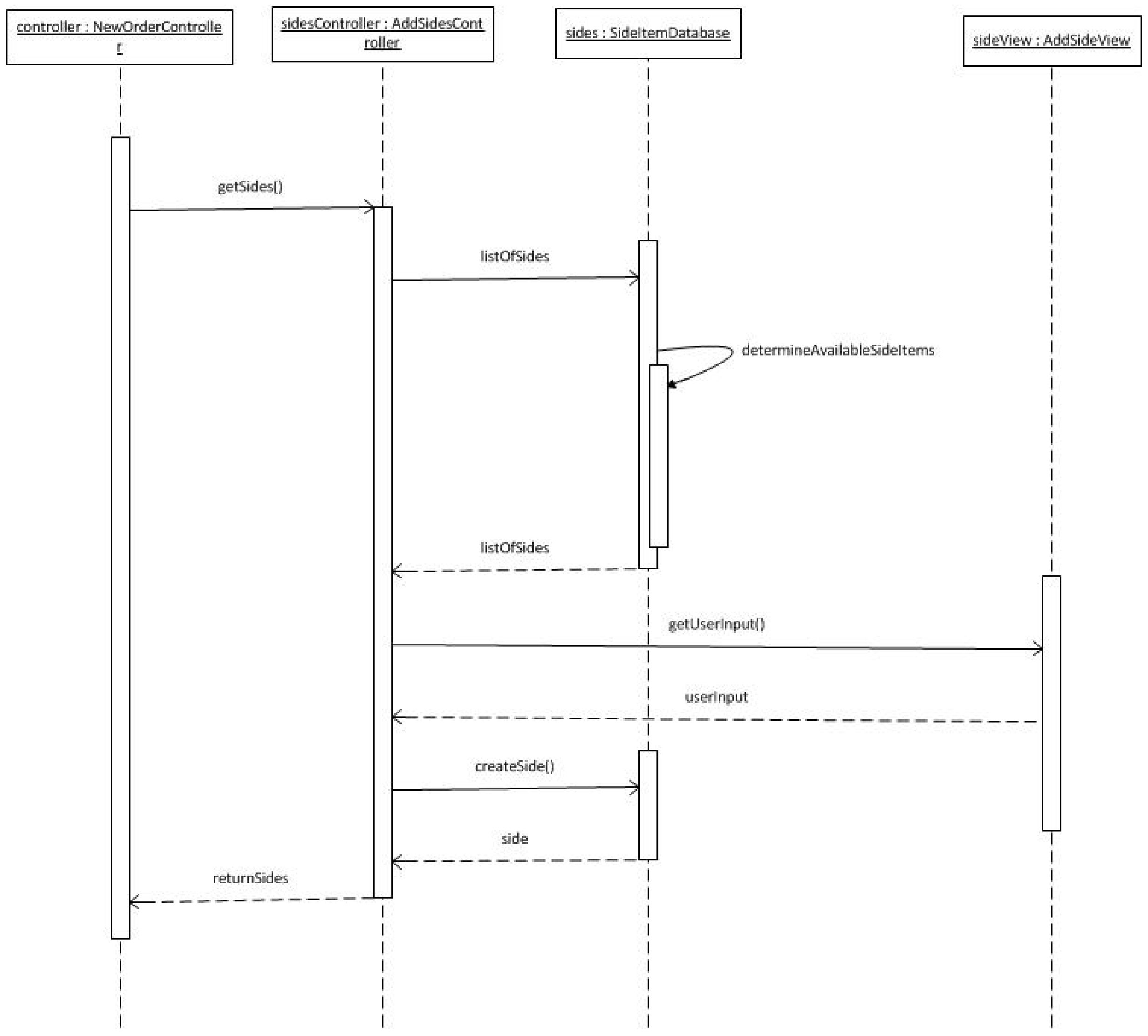
Class Diagram

UML class diagram representing the relationships between classes of the data model.


Team Caribou
UML For Model
Updated November 7th, 2011
Kitchen
+ninja : KitchenNinja
+cooksWorking : bool
+ovensWorking : bool
+driversWorking : bool
+open : bool
+openKitchen() : void
+closeKitchen() : void
+addOrder(in order : Order)
+removeOrder(in order : Order)
+canModifyOrder(in order : Order) : bool
+canDeleteOrder(in order : Order)
+getOrders() : ArrayList<Order>
FoodItem
-cookStartTime : long
-cookTime : int
-cookWaitStartTime : long
-preparationStartTime : long
-deliveryWaitStartTime : long
-name : String
-ovenSpaceUnits : int
-price : double
-preparationStartTime
-prepTime : int
-prepTime : long
-status : FoodItemStatus
+calculatePrice() : double
+getFormattedPrice() : double
Menu
-menuItems : ArrayList<FoodItem>
+addItem(in item : FoodItem)
+removeItem()
0..*
1
1
Address
-location : String
-timeToLocation : int
+getAddressForAlias() : Address
1
0..*
1..*
Customer
-name : String
-phoneNumber : String
-deliveryAddress : Address
-customerId : int
+isValidPhoneNumber() : String
+matchesSearch() : bool
1
0..*
1
Topping
-location : ToppingLocation
+calculatePrice() : double
SideFoodItem
0..*
ManagerReport
-orders : ArrayList<FoodItem>
-averageFormat : DecimalFormat
-decimalFormat : DecimalFormat
-maxFormat : DecimalFormat
+avgTotalTime() : double
+calculateAvgCostPerOrder() : double
+calculateAvgTimeWaitingForPreparation() : double
+calculateAvgTimeWaitingToBeRetrievedForDelivery() : double
+calculateAvgTimeWaitingToCook() : double
+calculateMaxTimeWaitingForPreparation() : double
+calculateMaxTimeWaitingToCook() : double
+getNumberOfOrders() : int
+maxTotalTime() : double
+printAvgCookingWaitTime() : string
+printAvgCostPerOrder() : string
+printAvgTimeWaitingForPreparation() : string
+printAvgTimeWaitingToBeRetrievedForDelivery() : string
+printAvgTotalTime() : string
+printMaxCookingWaitTime() : string
+printMaxTimeWaitingForPreparation() : string
+printMaxTimeWaitingToBeRetrievedForDelivery() : string
+printMaxTotalTime() : string
+printNumberOfOrders() : string
1
0..*
1
Order
-customer : Customer
-foodItems : ArrayList<FoodItem>
-orderId : int
-orderIdDf : DecimalFormat
-status : OrderStatus
-timeDelivered : long
-timeOrdered : long
+addFoodItem(in order : Order) : FoodItem
+calculateEstimatedDeliveryTime() : int
+calculateDeliveryWaitTime() : int
+calculateOvenSpaceUnitsRequired() : int
+calculateSubtotal() : double
+calculateTotal() : double
+getCookingTime() : int
+getCookStartTime() : long
+getCookWaitTime() : long
+getDeliveryWaitTime() : long
+getPreparationTime() : int
+getPrepStartTime() : long
+getStatus() : OrderStatus
+getTax() : double
+getTimeWaitingForPreparation() : long
+getTotalTime() : long
1
PizzaFoodItem
-toppings : ArrayList<Topping>
-size : PizzaSize
+addToppings(in topping : Topping)
+calculatePrice() : double
+removeTopping() : void
+canAddToppingToLocation(in loc : ToppingLocation) : bool
+getFormattedPrice() : String
+removeTopping(in topping : Topping)

Sequence Diagram

The following sequence diagram illustrates the “Add Side Item” scenario, which is a function of the “Order Creation” component.



Design Rationale

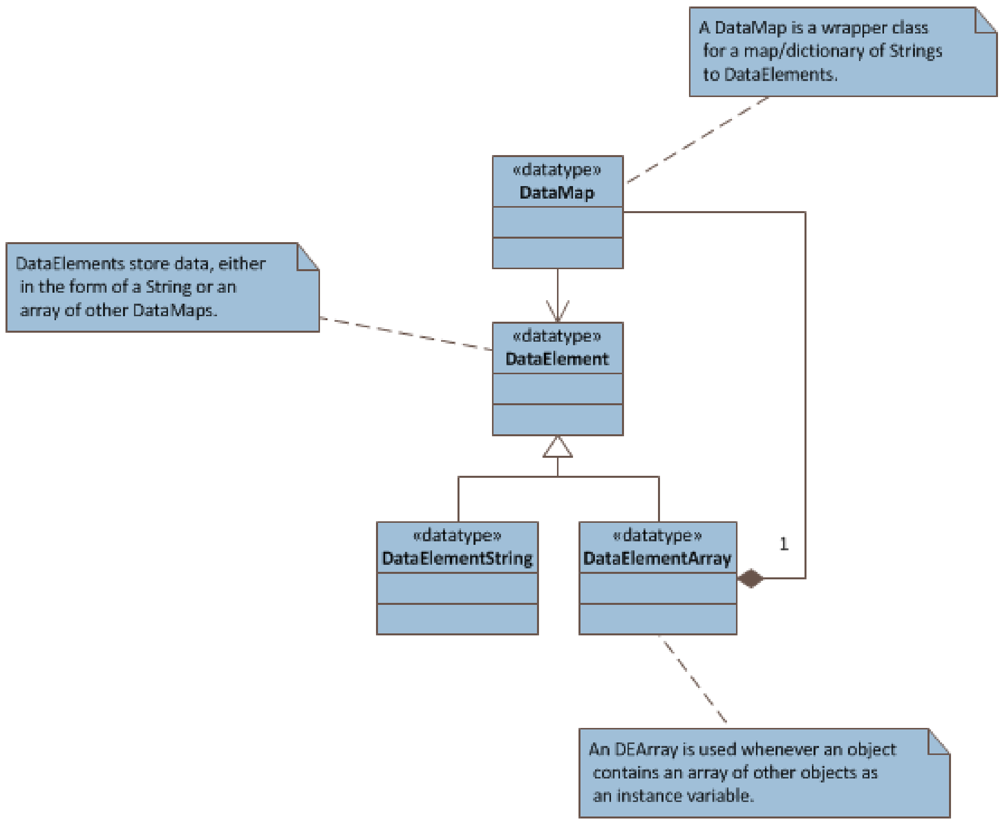
In this section, we outline some additional design points that have not been addressed above. Many of these are still being worked on and should not be taken to be a final design decision at this time.

Data Structure

The data structure described in the diagram below is meant to be as general as possible. Each object that must be stored persistently will be stored as a DataMap, which is a map with string keys and DataElement values. Essentially, this encodes each of an object's instance variables as a DataElement that is referenced by a string containing the variable's name.

A DataElement can be one of two datatypes: a string, which will hold basic scalar information; or an array, which will hold a collection of objects that are themselves encoded as DataMaps.

Each class that needs to be stored persistently will implement two methods of an interface: one to encode itself into a DataMap structure, and one to construct itself from a DataMap. This allows the object that writes the data to file to operate in a very general fashion without the need of any particular knowledge of the objects it is writing. Moreover, an end goal will be to store persistent data in XML files; since a DataMap structure consists solely of strings and arrays, this will make testing and debugging the data manager component much simpler.


A DataMap is a wrapper class for a map/dictionary of Strings to DataElements.
«datatype» DataMap
DataElements store data, either in the form of a String or an array of other DataMaps.
«datatype» DataElement
«datatype» DataElementString
«datatype» DataElementArray
1
An DEArray is used whenever an object contains an array of other objects as an instance variable.

Logic Controllers

By "logic controller," we mean the entity (most likely a class or set of classes) that controls the flow of of the execution of the program. A logic controller is the component that handles requirements such as:

- "When the program starts up, show the log-in screen."
- "Once the user logs in, display the main menu."
- "If the user presses the 'New Order' button, show the 'New Order' dialog / menu."

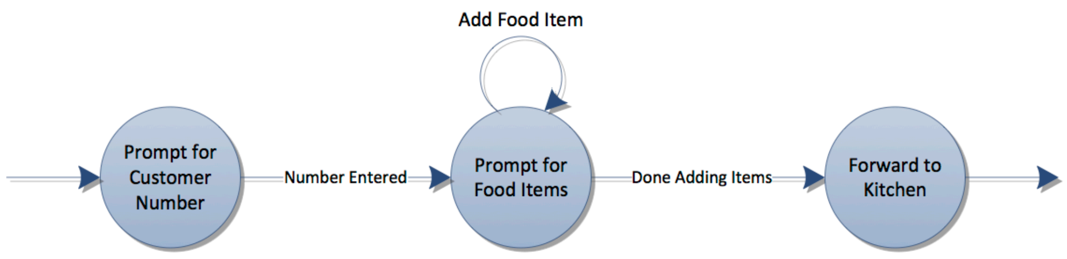
In essence, logic controllers take user input (passed from the view) and decide how to handle it: usually by modifying a data structure and/or telling the view to display something else.

It would be nice to establish a general protocol for logic controller and view interaction. This is discussed in this section and the next ("View Behavior").

Drew proposes the following model for logic controllers. Note that each component (see "Components" above) will have one or more logic controller class.

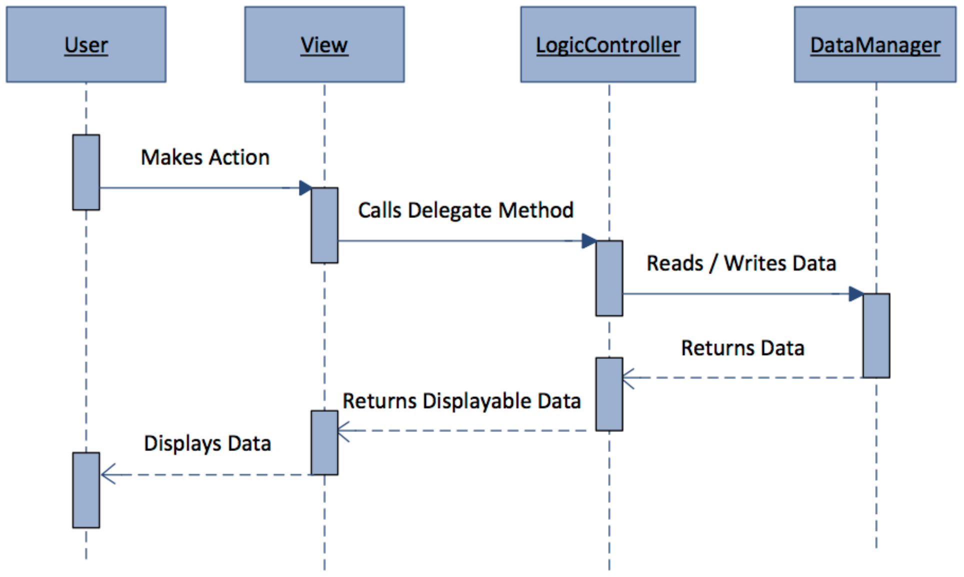
- A logic controller has a set of "states". These will indicate the current status of the controller. For example, the "Order Creation" controller will have states like:
    - "Prompting For Customer Phone Number"
    - "Prompting For Food Items"
    - "Forwarding Order to Kitchen"
- A controller occupies exactly one state at all times.
- When the controller receives messages about actions from the view, it acts depending on its current state. In general, this will involve modifying internal data and subsequently advancing to a new state.

With this model, each logic controller can be easily described as a finite state machine. For instance, the relevent FSM for the simplified description of the "Order Creation" controller is shown below.



The implementation of a controller follows very easily from this description as well: each controller implements a method in which it responds to input from the view. In this method, it checks its current state and, according to the FSM and the input, performs an action and advances to a successor state.

Another advantage of the FSM model is that the controller is only active immediately proceeding input from the user. This is to say that there is no part of the controller that is constantly active and waiting for input. The logic controller lies dormant until it is needed to respond to some event. The following simple sequence diagram illustrates the life cycle of a single action event and how information resulting from it travels through the MVC hierarchy.



(The “delegate method” referred to here is simply the method in which the view notifies the controller about input.)

View Behavior

It would be optimal to have a system for conveying information between the controller and view in a way that does not require the logic controller to be modified for each type of view (command line and GUI). This requires some sort of interface for input and output between the logic controller (abbreviated LC) and the view(s). The thought process behind this interface will be recorded here.
The means of input and output for the command line and graphical user interfaces are outlined in the following.

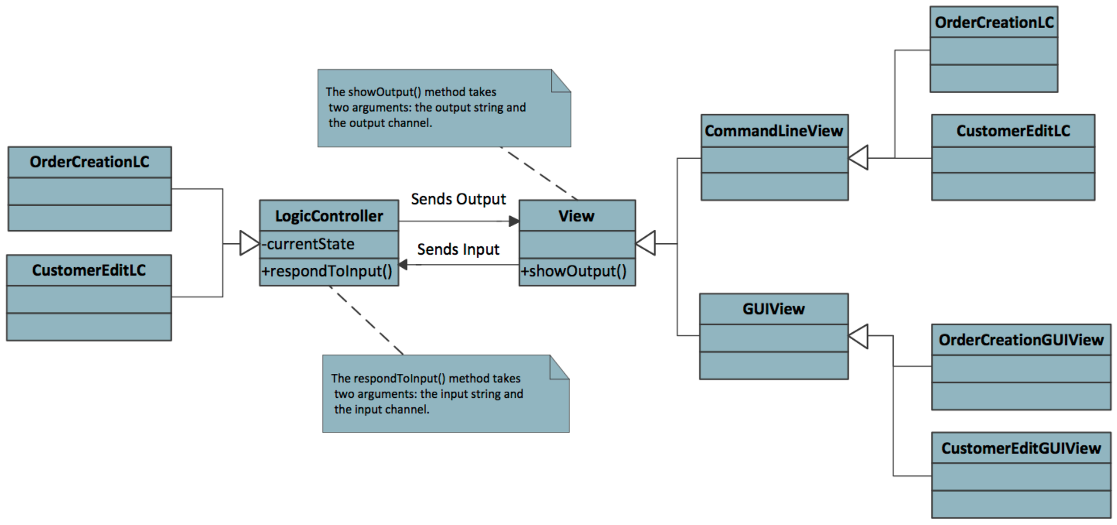
	Command Line	**GUI**
Input	● Text (via System.out)	● Labels ● Textboxes
Output	● Menu Selection (by number) ● String Input (via System.in)	● Buttons ● Textboxes

Note that a textbox in the GUI input might be the same textbox that must receive output. (For example, a string entry form might need pre-loaded or filler text.) This motivates the following:

Each view class will maintain what will be called input and output “contexts,” which are abstractions of different methods of inputs and outputs. Each context will correspond to a single method of receiving input or sending output. For example, a GUI view will generally have textboxes for input contexts and labels for output contexts. For the command line, the input and output channels could be standard input and output, but the context could vary depending on the type of input or output that is being conveyed through standard input or output.

The advantage here is that the communication between the view and the logic controllers is very general. A logic controller sends output (as strings) to the view and specifies where it should be displayed by indicating an output context. It need not know any specifics about how the information will be displayed (be it command line or GUI). Also, upon receiving input, the view invokes the "respond to input" method of the LC (discussed in the previous section) and passes it the input string and which input context it came in.

It is important to note, however, that this does not eliminate the need for specific implementations of a view for each component or function; this necessity remains. Each component will have its own logic controller implementation as well as two views (GUI and command line) that are constructed with the component in mind. The generality presented here is not meant to eliminate the distinction between view or controller implementations for different components, but instead to allow simple interchanging of the command line and the GUI views as the GUI is constructed. The following class diagram illustrates the relationships described. The key things to note are the LogicController class, which is the parent of every component-specific LC class, and the View interface, which is the general link between the specific view and the LCs.



Note that, as mentioned previously, each component will have its own LC, CLView, and GUIView; the two shown are meant only to illustrate that there will be multiple children for LogicController, CommandLineView, and GUIView.

"Ninjas"

Once an order has been finalized by the cashier it is sent to the kitchen to be processed automatically by the program. This part of the system simulates what in real life would be the cooks, ovens, and drivers that take an order from preparation to delivery. These "ninjas" do all of the work behind the scenes and advance the orders from state to state.

There are 5 ninjas: kitchen, cook, oven, driver, and time. The kitchen ninja coordinates the cook, oven, and driver ninjas. The cook ninja manages receiving orders for preparation, holding them in the preparation stage for the correct period of time, and then sending them to wait to be placed in an oven. The oven ninja then takes these orders from the queue and places them in an oven (if there is space). After the proper amount of time has elapsed, the oven ninja removes the order from the oven and sends it to wait for delivery. Finally, the driver ninja picks up the finished orders and changes the state to en route until the correct amount of time has passed, at which point the order is labeled as finished and written to the database as a past order. While all of this occurs, the time ninja is in charge of keeping a count of the time that has passed. This time counter counts one real time second as a minute within the program. It is this fictional time that the other ninjas base their timing upon.

The simultaneous nature of the actions of each of the ninjas requires threading to operate properly. This allows each section of the kitchen to manage their orders all at the same time while also allowing the user to input new orders by interacting with the controllers.

For R1, each ninja is only in charge of one corresponding cook, oven, or driver. In the future this will change as the ninjas will need to manage the flow of orders through multiple terminals.

The idea of the ninjas is that they operate completely without user input and automatically handle each order, thus informing the naming convention.

Current Issues

This section is meant to keep track of the current issues that have come up during the design process. It will be updated as new issues are encountered and as existing issues are resolved.

Issue	Description	Resolved?
01	*How will data be stored persistently?* *Th*e options are using Object serialization or writing a more general "hand-rolled" method. While the serialization method would require less thought, the hand-rolled method would allow us to use custom decoders to read stored data into an object without requiring the original stored object to still exist. Furthermore, the generality of the hand-rolled method would make a very simple data-handler possible.	**Yes. See the "Data Structure" discussion.**

Missing Requirements

- The ability to edit the price of pizzas.
- Support for phone number extensions for customers.
- The ability to set individual prices for toppings.
- Manager Report calculates values based on all past orders, not just orders placed on the current day.
- Manager Report does not display order ID numbers for max values.
- Toppings do not have individual prices.
- Past orders for a particular customer are not displayed with the customer's information.