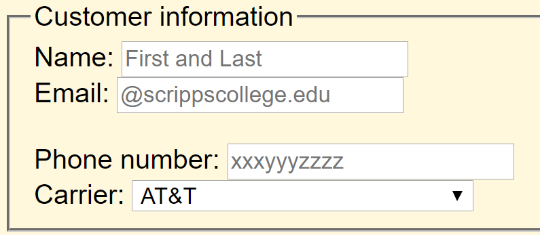
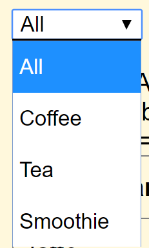
Coffee2Go

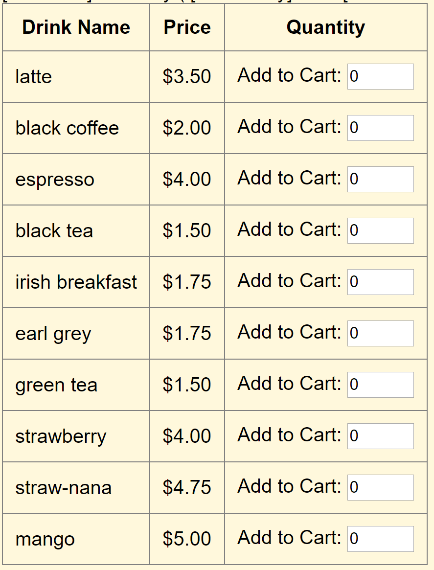
By Lindsey Cleary, Barbara Ko & Jason Barbarosh

**1.1 Introduction**

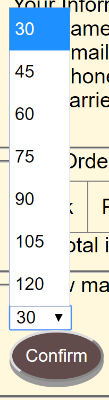
Our idea started with a passion for coffee, more specifically a way to get coffee more efficiently. Inspired by Starbucks’ mobile app and the girl scout homework, we decided to create a web order form for the Motley coffeehouse. A customer simply pulls up the site, and will fill in the fields with their information (name, phone #, email and carrier):



Following this, they will select which drink they want to order. The display can be changed to make navigation of the menu easier. All the drink options are displayed per default, but the dropdown navigation allows the user to narrow down their selection with 1 of 3 criterion: coffee, smoothie or tea.

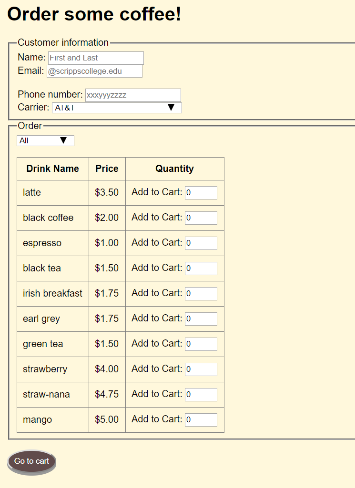


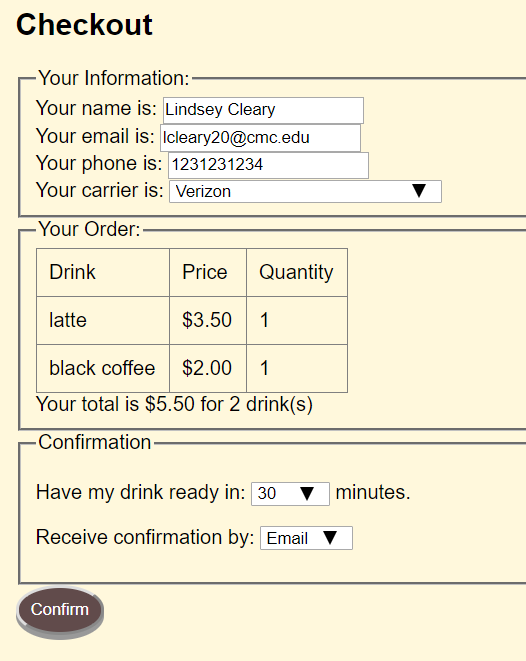
The customer selects his or her drink by entering (or incrementing) the quantity of the drink. They can order a maximum of 5 of any drink.

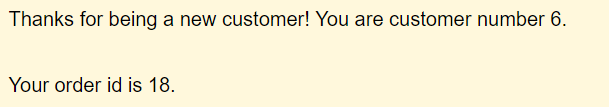
After he or she has made her selection, the customer clicks the “go to cart” button, which takes them to the checkout page to confirm their order. On this page, their information is displayed, their drink, quantity and price as well as the total quantity of drinks ordered and the total price. They may edit their customer information if they need to make any corrections. The user selects the time which they would like to pick up their drink(s) in (with a min of 30 minutes) and whether they would like to receive their confirmation by text, email or both. Then ‘confirms’ the order.

After the order has been confirmed, the page then displays ‘thank you for your order’ and they see their order id and customer id. They are emailed or texted (or both) a response to confirm their order based on the preference they selected.

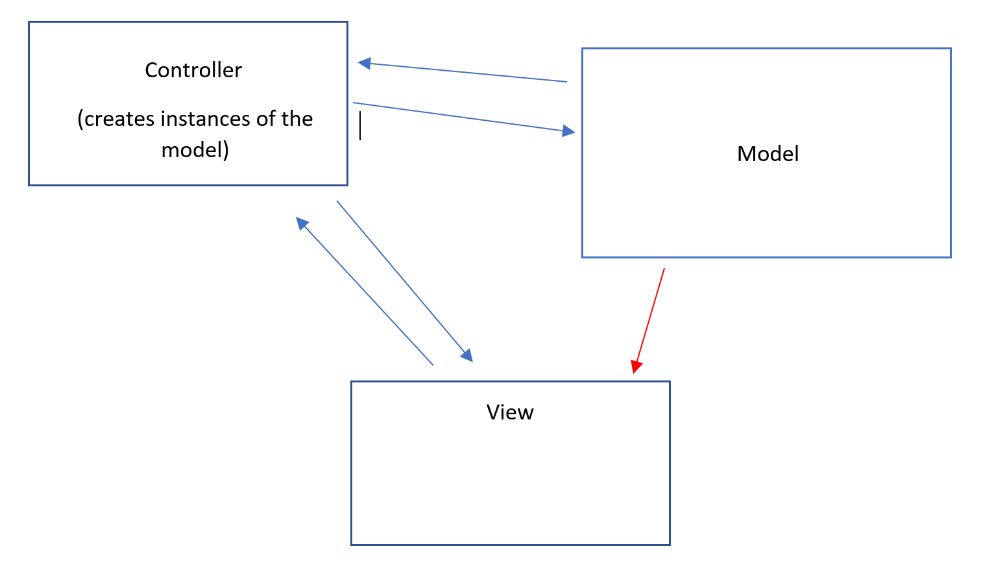
Full page renderings:

The order page: The checkout page:

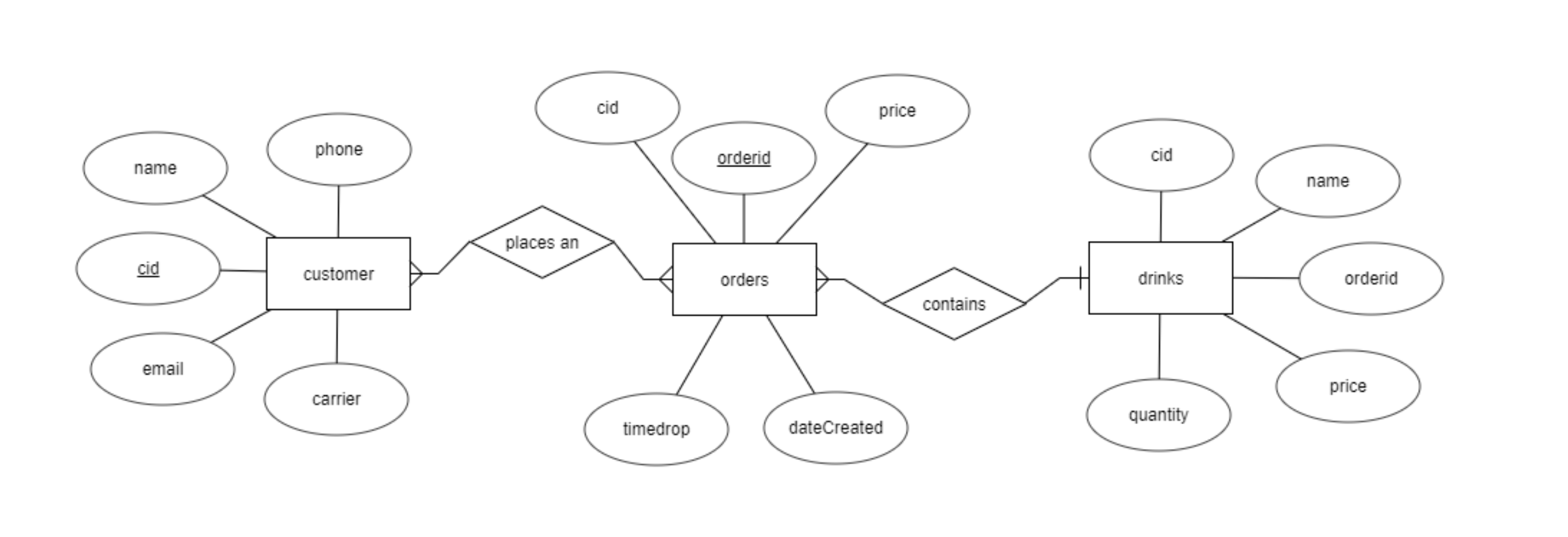


Confirmation page:

**1.2 Software Design**



We worked to implement the model-view-controller method with our project. The index file serves to help navigate via calling the controller’s methods. The view holds the pages that the user sees: the order page, the checkout page, the confirmation page, and of course the view file which renders the necessary files. The model mainly holds the cart, the database, the mail function page, the queries and the model whose functions access the cart and the database. The controller contains the functions that handle the submission of the order page and the checkout page. On the submission of the form of the order page, its post array is received by the invoke() function in the controller. This holds the customer information and the drink order. This information is passed to the model in its relevant function (relating to customer or drink order). The controller then calls the view to render the checkout page which displays the customer’s information and order and total order price. The red arrow indicates a slight break in our MVC design because the cart is accessed by the checkout page as it is implemented right now. Anyways, in this checkout form they may edit their information, select how they wish to be notified and the time duration until they wish to pickup their order. Once submitted, the controller handles this form in the confirm() method, adding all the elements into the database through the method called on the model: addAlltoDb(…). This model function executes the insert statements to put the customer and drink order as well as the other necessary elements for each table into the database. Then the email or text or both is send through calling the sendMail function on the mail object, and the cart is cleared with the model’s clearCart() function. Afterwards, the controller calls to the view to display the confirmation page.

Key concepts included in our architecture are form validation and executing MySQL statements with php (PDO statements). Furthermore, what we learned about MySQL queries is present in the reports page which generates queries for orders in the last 30 minutes, the most popular drink, and the customer that is the “best” (bought the most). We also used JavaScript to handle the dropdown display for which types of drinks to ‘show’ or ‘hide’ based on their class of coffee, tea or smoothie. Obviously, our knowledge of html and CSS were also utilized to provide a humble aesthetic for our web page that also provides necessary information to the customer about the business they are ordering from. 

Our database, “motley” has three tables: customer, orders and drinks. The ER diagram above shows the attributes that each entity has and the relationships between the entities. The customer has a name, customer id, phone number, email and carrier. Each order has an id, a price, a timedrop attribute (time until customer wants to pick up the drink), a dateCreated attribute (timestamp of the order) and the foreign key of the customer id (cid). The drinks table also references the order id and the customer id as foreign keys, and also has a name (of the drink), the quantity of the drink, and the price of it.

**1.3 Contributions**

Lindsey: I wrote the main parts/function of the controller methods, and the communication between the order page and the controller and the checkout page and the controller. I debugged the model so that it correctly adds the drinks into the drink table and the orders into the order table. I contributed to the html forms in the order and checkout pages and wrote the confirmation page. Also, I contributed to writing the progress reports and the final report.

Jason: I wrote the tables for the database and the PDO queries, as well as handled functions to insert customer information and drink order into the database and their respective tables. I constructed the reports.php page.

Barbara: I wrote the front-end design, the JavaScript for dropdown menu, and the mail functionality. Also I did the validation, and allowing the customer information to be updated in checkout page.

**1.4 Future Work**

**Limitations**:

When it comes to inserting information into the database, the MVC design works as it should. The model is the side that interacts with the database, through what it receives from the controller. However, the view does access the model at times, which is not a good practice. The reason for this is because our site takes in the customer’s information with the drink in the first $\_POST array submitted from the order page, it was difficult to find a way to input this into the database, only to be loaded back onto the next page for confirmation and revising of the information (and eventually the order too). The checkout page accesses the $\_SESSION[‘cart’] array, which is the model’s information. This leads to our program not being as dynamic as we would like it since updates to code in one area will result in the necessity of updating other pages in the view as well as possibly the controller. Obviously, a solution would be to remove this access and load the information through the controller instead of directly from the model, that could be done if we had more time.

In addition, the order page does not load the menu from a database of existing drinks, instead the values are hardcoded in an array that is in the cart, along with their prices. This was what we intended to use until the database issues we had were resolved but unfortunately, they took longer than expected so we have been forced to finish with this result. With time, the idea to load the drinks into the database of the menu from an excel spreadsheet and displayed with a query could be achieved.

Each section of our program addresses different concerns, such as the view receiving the user input, the controller taking in the post for submission and the post for confirmation in separate functions and accordingly inserting the elements from the post into the model which puts them in the database. The separation of functions and function of each page provides a good style for our programs overall functioning.

**Future work**:

If we had more time, the first things we would do would be to take away the communication that occurs between the model and the view. This is to allow our program to have single responsibility principles and high cohesion (low coupling). Off of that adjustment, we would use the excel spreadsheet to create a table of the menu and display this in the order page instead of having hardcoded values that come from the model’s cart.

We added in the customer being able to edit their personal information but with more time we would allow the quantity of drinks to be edited (or be deleted from the order). In addition, we would allow add-ons for the drink selections (whip cream, extra espresso shots, etc.) which would increase the price of a drink. Thus, our diagram would adjust to our original plan where we had a standard drink entity who had a relationship “is a” with a customized drink entity that includes an array of add-ons that can be added to the current drink and placed into the order along with their respective prices.

Lastly, we would implement the employee side of the site. This way the employees at the Motley can log in with only a password to view a page that is a list of all the submitted orders. The orders displayed have options that include: in progress, finished, and canceled (custom message). Which when selected and updated will send an update text or email to the customer that a) their drink is being made, b) their drink has been made and waiting for pickup/payment, or c) their drink has been cancelled with a personal message as to why (perhaps they are out of the necessary ingredients). The reason the menu will be loaded from an Excel sheet is so that employees are able to access it and update the menu items and prices in the most efficient way.

**1.5 Reflection**

From the get-go we would all have started with a more elaborate understanding of the MVC design. This would have eliminated a lot of confusion about what communication is allowed between the model and view and controller (only through the controller, if you were wondering😉). At first, splitting up the work was hard because we weren’t all sure what we were supposed to be doing to work with the design, but by the end we feel that we have sufficient understanding of how it works in theory. Splitting up work became easier after the groundwork

Another important item would have been to focus more on functionality rather than ‘extra’ ideas or the design. The database (inserting and retrieving information from it) is probably the most important part of the project since it allows the information to be stored and then accessed. So instead of starting with the layout of the order form and focusing time on getting the drop-down to display correctly we could have all worked on setting up the database. But that comes hand in hand with a better understanding of how the MVC design works.

**1.6 Libraries / Themes / etc.**

N/A