Putt Putt Golf Site Development

Eighth Wonder

Lane Barnes - Authentication and Testing
Evelyn Teeples - Scrum Master
Josh Williams - Site Design and Documentation
Eathan Hodgkinson - Site Design and Database Models
Carter Parks - Database Modeling and Developer

Overview

This project was created to help run and manage a Putt Putt Golf tournament. The four types of users that can use our application are players, managers, sponsors, and drinkmeisters.

A player can play in a tournament and the application will help track their hole and score.

A manager verifies users as well as edits tournaments and the drink menu.

A sponsor can donate money to sponsor a tournament and that tournament will be named by their company.

A drinkmeister makes and delivers drinks. The application helps manage orders, drink recipes, and delivery locations for the drinkmeister.

All users can order drinks and add money to their accounts.

Major Design Decisions

The major design decisions for our project were: what platform to build our application in, how to share our code, how we should communicate as a group, what coding languages to use for development, and how the application should look when it was finished.

We decided to build our application using Django since it was relatively easy to learn and most of our team had used it before. Django also has very professional and easy to read documentation which was another major reason for choosing Django.

For sharing our code between group members we decided to use Git because it was strongly suggested by the project owner and most of our team was familiar with it. Git allowed all of us to work on the project at the same time without issue which allowed us to break our project down into manageable pieces.

Major Design Decisions

To communicate with each other we decided to use Discord because all of us knew how to use it and Discord is fantastic for communication. We could text or call in Discord and if someone had a coding problem they could share their screen and get help.

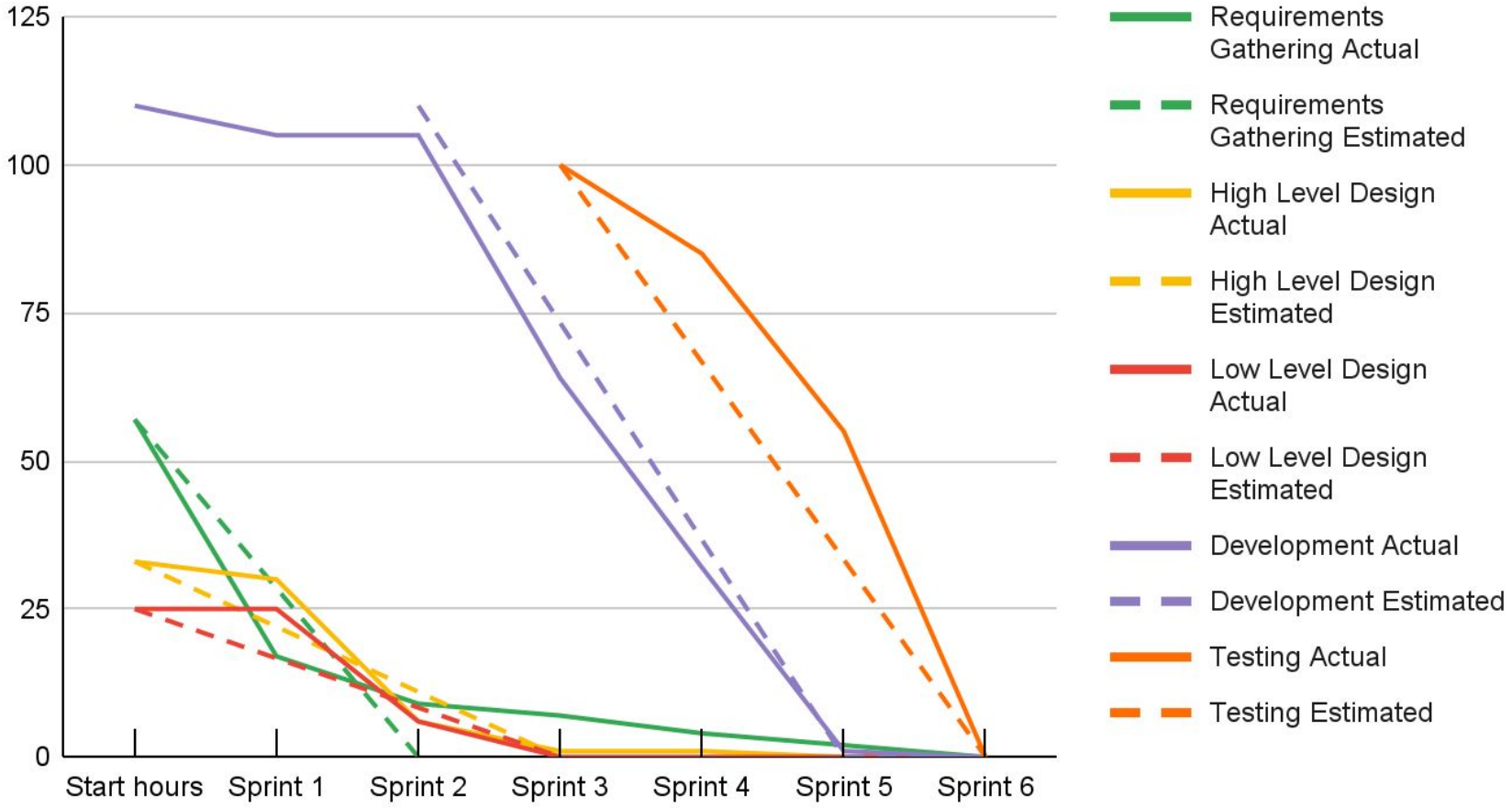
We decided to write our code using Python, HTML, and CSS because our team collectively decided those were our strongest languages and Django uses python so it made it an easy decision for us.

The last design decision we made was about how our application was going to look when it was finished. We decided to go with a green and orange color scheme because it looked nice and had a Putt Putt Golf feel to it.

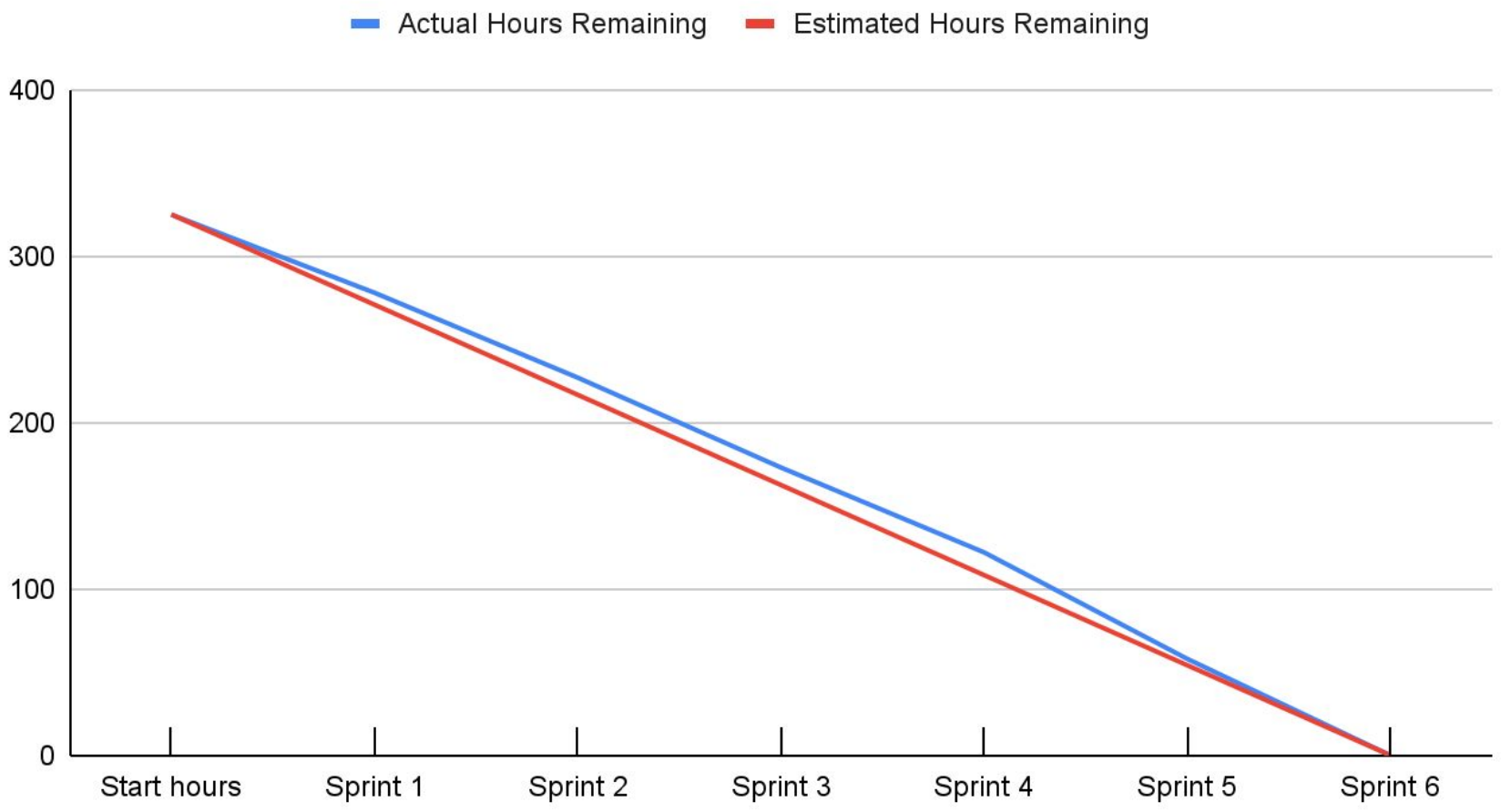
Agile Practices

Our first week as a team was a bit of a disaster because we did not have any agile practices put in place but once we did our productivity began to increase. We started meeting almost every week day and we started breaking down large tasks into smaller pieces so multiple people could work on them. We found that meeting more often helped keep us on track since we were constantly reporting to each other. Breaking down these large tasks helped lessen the anxiety associated with them. We also started using Trello to assign tasks to different group members and that allowed us to constantly know what we had to get done and what everyone else was working on.

5 Color Burndown Chart



Overall Burndown Chart



Deployment

The deployment process for the site is as follows:

- After all pull requests are merged, and testing is complete, the server is launched from the AWS EC2 console.
- After connecting to the server, which runs Ubuntu 20.04.2 LTS, changes to the software are manually pulled from the Github repo where they are stored.
- A series of tools provided by Django are used to perform the following steps.
    - Changes to the database schema are implemented by running a command which compiles the various changes to the Django models into a migration (a slice of the history of the schema), then by migrating these slices to the database.
        - The database is a MySQL database handled entirely through Django's tools.
    - Static files (Javascript, CSS, images) are collected into a single location for easier access by Django.
- Nginx is launched as a system service, along with gunicorn_start, a shell script to run Gunicorn with a series of presets.
    - Gunicorn is a Web Server Gateway Interface HTTP server written in Python, which allows for multiple worker processes (4 in our case) to serve requests, unlike Django's development mode.
    - Nginx is a web server with a variety of features, similar to Apache, but with better performance. It interfaces with Gunicorn, which in turn interfaces with Django, to provide security and scalability.

After these steps have been completed, the site is up and running on port 80 at whichever URL points to the server.

Testing

The majority of our testing was done by us as the developers through manual testing. When we wrote any new feature we would test it on the website immediately to make sure it was working correctly. If the new feature did not work as intended we knew right away and began fixing it.

The other type of testing we did was unit testing for our models and to do this we used Django's built in testing library. We were able to program Django to create our models with very specific information and then we had Django query the database to ensure those models were stored just as intended. Our testing showed that our models were being created and saved to the database correctly.

Requirement - User Authentication

FURPS

- Functionality
  - Each user must have a unique username; although, duplicate emails are allowed
  - The user must be able to select their own user type while creating an account
  - When a user logs in they will be able to see information relevant to their user type.
- Usability
  - Anyone visiting the application is able to make an account.
  - Anyone can log in to an existing account if they have the correct information
- Performance
  - Creating or Logging in to an account should not affect the overall performance of our application
- Supportability
  - A user should be able to create or log in to an account on virtually any device that can access our application

Audience-oriented

- Business Requirements
  - A user has to be authenticated to buy anything.
- User Requirements
  - Logging in and creating an account are simple straightforward processes
- Non-functional Requirements
  - The account creation page will tell the user when they are missing required information.
  - The login page will tell the user if the information they input is incorrect.
- Functional Requirements
  - A user must be added to the database when an account is created
- Implementation Requirements
  - The application must be connected to the database for any authentication to occur

MOSCOW

- Must
  - A user must be able to log in to an existing account
  - A user must be able to create an account.
  - Each account must have a user type
- Should
  - A user should be able to view pieces of the website without being authenticated
- Could
  - A user could have a different account creation process based on the user type they selected.
- Won't
  - A user won't be able to make an account if their username is already taken
  - A user won't be able to retrieve their password if they forget it.

Requirement - User Authentication

To create an account the user needs to input all of their relevant information and click create account.

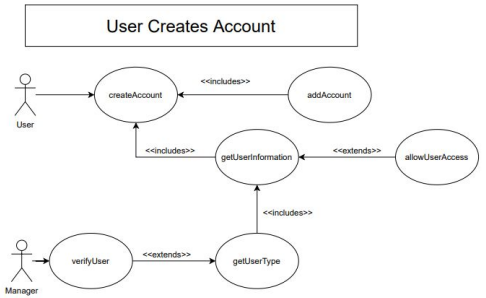
If any needed information was left blank then the application will tell the user what needs to be filled in.

If account creation was successful then the user can login with their username and password. The user's account is now stored in the database

Any managers will now be able to verify the user's new account if the account type was a drinkmeister or sponsor.

Prototype Design

Use Case Diagram



Participating Actor: User, Manager

Entry Conditions: User wants to create an account

Exit Conditions: User creates an account

Event Flow:

1. User goes to the Website and clicks create account.
2. User enters in their Username, Password, Name, Email, and Phone Number
3. User selects what user type they want
4. System adds their account to the database
5. Manager verifies the user account
6. The user gains access to the website

Requirement - User Authentication

The main tasks associated with developing this feature were creating an account and being able to log in to an account.

These two tasks were completed by Lane and testing for this feature was done with unit testing and manual testing.

The unit tests created models with known information and then queried the database to ensure those values were correct.

The manual tests involved creating users with varying information on the application and then logging in to the database to ensure that everything was saving correctly.

```
class PlayerTestCase(TestCase):
    def setUp(self):
        admin = models.User.objects.create_user(username="admin", id=0,
                                                email="admin@admin.com",
                                                password="asdf",
                                                user_type="1",
                                                phone_number="1234567899")
        models.Player.objects.create(user=admin)

    def test_player(self):
        player = models.Player.objects.get(user=0)
        self.assertEqual(player.user.username, "admin")
        self.assertEqual(player.user.phone_number, "1234567899")


class UserTestCase(TestCase):
    def setUp(self):
        models.User.objects.create_user(username="admin", id=0,
                                        email="admin@admin.com",
                                        password="asdf",
                                        user_type="1",
                                        phone_number="1234567899")

    def test_user(self):
        user = models.User.objects.get(id=0)
        self.assertEqual(user.username, "admin")
        self.assertEqual(user.user_type, 1)
        self.assertEqual(user.email, "admin@admin.com")
        self.assertNotEqual(user.phone_number, "12345")
```

Requirement - Drink Meister

FURPS

- Functionality
  - There may be multiple Drink Meisters, Drinks, and Orders.
  - The manager must be able to add, edit, or delete drinks from their account.
  - Any user may order a drink if they have enough money in their account.
  - The Drink Meister must be able to see all the orders from all users and be able to deliver and finish the orders.
- Usability
  - The drinks may be able be ordered by any user.
  - The Drink Meister can deliver the orders.
- Performance
  - The site is hosted on AWS and there isn't likely any performance issues affecting the Drink Meister.
- Supportability
  - This app should be accessible to virtually any device that has internet and browsing capabilities.

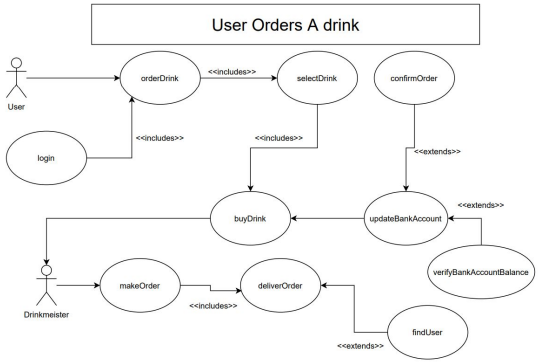
Audience-oriented

- Business Requirements
  - The Manager can control the drinks and orders.
  - Drink Meisters have their own employee profiles and privileges.
- User Requirements
  - The app is simple and user friendly for users to order drinks.
  - Managers and Drink Meisters are also easily able to accomplish their tasks.
- Non-functional Requirements
  - The interface for the drinks and Drink Meister is simple and easy to use.
- Functional Requirements
  - There may be multiple Drink Meisters, Drinks, and Orders.
  - The manager must be able to add, edit, or delete drinks from their account.
  - Any user may order a drink if they have enough money in their account.
  - The Drink Meister must be able to see all the orders from their account and be able to deliver and finish the orders.
- Implementation Requirements
  - Drinks and locations must be created before drinks may be ordered.

MOSCOW

- Must
  - The manager must be able to add, edit, or delete drinks from their account.
  - The Drink Meister must be able to see all the orders from all accounts and be able to deliver and finish the orders.
- Should
  - The users should be able to order any of the drinks in any quantity that they want given that they have the funds.
  - The orders should include a delivery location and personalized instructions.
- Could
  -
- Won't
  -

Use Case Diagram



Participating Actors: User, Drinkmeister

Entry Conditions: User wants to order a drink

Exit Conditions: Users gets their drink
User does not have enough money to pay for their drink.

Event Flow:

1. User logs into the system
2. User orders a drink from the Drink Menu
3. The System Confirms the Drink Order
4. The System verifies the User has enough money to buy the drink
5. The User Pays for the drink
6. The Drinkmeister Receives the Order and makes the drink
7. The Drinkmeister locates the user and delivers the drink

Requirement - Drink Meister

The drink system implements multiple features and classes. And so the design was broken up into many different parts. The designs related to the Drink Meister were discussed and planned out to fit with rest of the requirements. Assignments were broken up into the following:

Class Diagrams:

- Evelyn: Drinks Class, Drink Menu Class, Bank Account Class
- Carter: Drink Meister Class
- Eathan: Manager Class(Create and edit Drinks)

Activity Diagram:

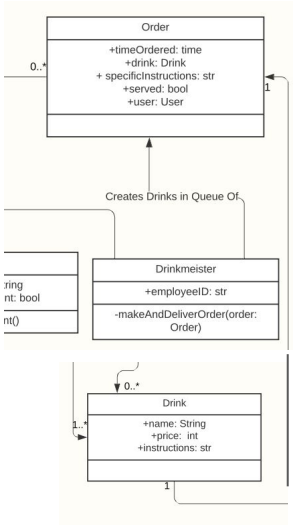
- Evelyn: User Orders a Drink

Prototype Wireframe:

- Josh: Drink Edit Page, Drink Meister Page
- Lane: Drinks Page, Order Confirmation Page

The prototypes of the drink pages were made to be user friendly. The user can have a reference of the available drinks and then choose among them. The pages were also designed so that the Drink Meister can easily access their employee pages from their account and deliver orders. Once the designs were completed in diagrams and prototypes, they were used and generally followed in the implementation of the Drinks and Drink Meister pages.

Class Diagrams



Prototype Designs

Requirement - Drink Meister

The basic templates for the Drink pages and the django models that they would use, were broken up into assignments for the group. The templates were designed to have some basic functionality first, and then the models were connected when they were completed. The models created a user child class for the Drink Meister. They also had to create drink and order objects that can be created and deleted. These objects had to be able to be used, modified, and stored in the database.

Functional Templates:

- Evelyn: Drink Meister Page
- Lane: Drinks Page
- Josh: Drinks Edit Page, Order Confirmation Page

Model Implementation:

- Eathan: Drink Menu Class, Drinks Class, Drink Meister Class, User Class
- Carter Manager Class, Bank Account Class

Connecting to Models:

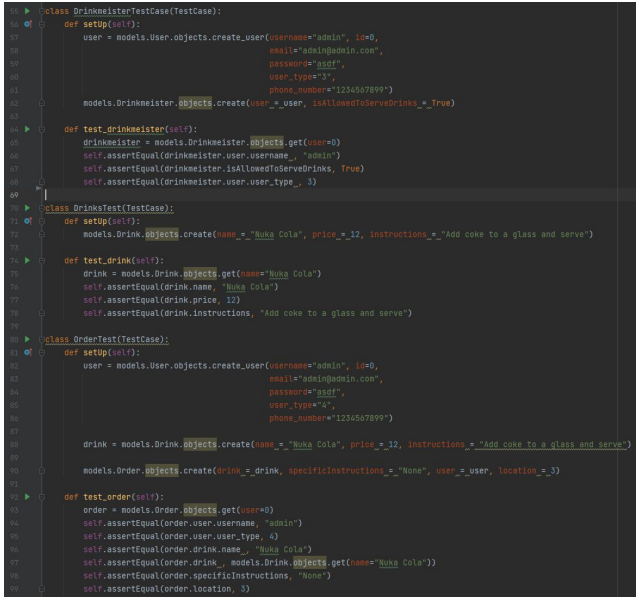
- Josh: Drinks Page, Drinks Edit Page, Drink Meister Page

Unit Tests:

- Lane: Drink, Order, Drink Meister

The pages were then extensively tested to make sure all the requirement were met. Unit tests were created to test the models and objects. The pages were also tested functionally as part of the development. The Manager has to be able to create and edit drinks. The users have to be able to order drinks if they have the money. The Drink Meister has to be able to see and deliver the orders that were sent. Bug fixes were also part of the development. Examples include fixing when the site couldn't handle orders from many people or when there were no orders.

Unit Tests



```python
class DrinkmeisterTestCase(TestCase):
    def setUp(self):
        user = models.User.objects.create_user(username="admin", id=0,
                                               email="admin@admin.com",
                                               password="asdf",
                                               user_type="3",
                                               phone_number="1234567899")
        models.Drinkmeister.objects.create(user_=_user, isAllowedToServeDrinks_=_True)

    def test_drinkmeister(self):
        drinkmeister = models.Drinkmeister.objects.get(user=0)
        self.assertEqual(drinkmeister.user.username_, "admin")
        self.assertEqual(drinkmeister.isAllowedToServeDrinks, True)
        self.assertEqual(drinkmeister.user.user_type_, 3)

class DrinksTest(TestCase):
    def setUp(self):
        models.Drink.objects.create(name_=_"Nuka Cola", price_=_12, instructions_=_"Add coke to a glass and serve")

    def test_drink(self):
        drink = models.Drink.objects.get(name="Nuka Cola")
        self.assertEqual(drink.name, "Nuka Cola")
        self.assertEqual(drink.price, 12)
        self.assertEqual(drink.instructions, "Add coke to a glass and serve")

class OrderTest(TestCase):
    def setUp(self):
        user = models.User.objects.create_user(username="admin", id=0,
                                               email="admin@admin.com",
                                               password="asdf",
                                               user_type="4",
                                               phone_number="1234567899")

        drink = models.Drink.objects.create(name_=_"Nuka Cola", price_=_12, instructions_=_"Add coke to a glass and serve")

        models.Order.objects.create(drink_=_drink, specificInstructions_=_"None", user_=_user, location_=_3)

    def test_order(self):
        order = models.Order.objects.get(user=0)
        self.assertEqual(order.user.username, "admin")
        self.assertEqual(order.user.user_type, 4)
        self.assertEqual(order.drink.name_, "Nuka Cola")
        self.assertEqual(order.drink_, models.Drink.objects.get(name="Nuka Cola"))
        self.assertEqual(order.specificInstructions, "None")
        self.assertEqual(order.location, 3)
```

Requirement - Manager Class

Requirements Definition

FURPS

- Functionality
  - There should only be one manager in the system
  - The manager must be able to verify sponsors and drinkmeisters after their accounts have been created
  - The manager must receive all the money from tournaments and drink purchases
  - The manager must be able to edit drinks
- Usability
  - This class will be used by the manager to run the golfing establishment
- Performance
  - Barring poor internet connection, or issues with the AWS server, there should be no performance issues affecting the manager as they use the app
- Supportability
  - This app should be accessible to virtually any device that can use the internet and can display HTML web pages

Audience-oriented

- Business Requirements
  - The manager should be able to run the golfing establishment using this app
- User Requirements
  - The app should be intuitive and require little to no training for new managers to use it
- Non-functional Requirements
  - The interface for the manager should look neat and simple
- Functional Requirements
  - The manager should be able to verify users, receive proceeds from all purchases and tournaments, and be able to modify the selection of drinks available
- Implementation Requirements
  - App should be fully functional locally before it is ported to the AWS server

MOSCOW

- Must
  - The manager must be able to receive money spent on tournaments and on drinks
  - The manager must be able to edit the selection of drinks available to customers
- Should
  - The manager should be able to verify the accounts of new sponsors and drinkmeisters
- Could
  - The manager could have access to a private drinks menu
- Won't
  - There won't be more than 1 manager for the establishment

Requirement - Manager Class

Class Diagram

Manager
- int yearsWorked - double mostMoneyHeld - int drinksSold - int totalTournamentsMade
- createTournament(): bool - editTournament(tournament: Tournament) - verifySponsor(sponsor: Sponsor): bool - verifyDrinkmeister:(dirnkmeister: Drinkmeister): bool - editDrink(drink: Drink) - addDrink(): bool

Prototype Drinks Edit Page

Requirements Definition (Continued)

Excerpt of Manager Implementation

```python
class Manager(models.Model):
    user = models.OneToOneField(get_user_model(), on_delete=models.CASCADE, primary_key=True
    yearsWorked = models.IntegerField()
    mostMoneyHeld = models.IntegerField()
    drinksSold = models.IntegerField()
    totalTournamentsMade = models.IntegerField()
    communityPosts = models.TextField(max_length=2000, default="")

    @staticmethod
    def createTournament(name, startTime, endTime, sponsor, approved, completed):
        tournament = Tournament(name, startTime, endTime, sponsor, approved, completed)
        tournament.save()

    @staticmethod
    def editTournament(tournament, name, startTime, endTime, sponsor, approved, completed):
        tournament.name = name
        tournament.startTime = startTime
        tournament.endTime = endTime
        tournament.sponsor = sponsor
        tournament.approved = approved
        tournament.completed = completed
        tournament.save()

    @staticmethod
    def verifySponsor(sponsor):
        sponsor.canSponsorTournament = True
        sponsor.save()
```

Design Choices

- All actions specific to the manager can be found on the manager's account page.
- When the manager picks an action, such as verifying users or editing the drinks menu, the manager is taken to a different page with further options. We did this to avoid having too much content displaying on the account page, making it look cluttered.
- Confirmation of whether or not an action is successful is given to the manager for transparency. For example, the manager is notified that they aren't able to schedule multiple tournaments on the same day if they attempt to do so.
- All the manager-specific pages have the same layout and appearance as the rest of the site, keeping the interface simple and intuitive for the manager.

Requirement - Manager Class

Scrum Tasks

- Developing this feature began with gathering requirements, and these requirements were recorded by Evelyn Teeples.
- Next, Eathan Hodgkinson created the class diagram for the manager class based on the requirements
- Lane Barnes created the activity diagram for the manager class, and Carter Parks and Eathan Hodgkinson implemented the manager model in Django based on the requirements, class diagram, and activity diagram
- Evelyn Teeples then created the prototype account and tournament edit pages for the manager, Josh Williams made the prototype page for editing drinks, and Eathan Hodgkinson made the prototype user verification page
- Next, the prototype pages for the manager were linked to the Django models, giving them added functionality. Josh Williams linked the drinks edit page and Eathan Hodgkinson linked the manager account page and the user verification page.
- Eathan Hodgkinson and Lane Barnes added the CSS code needed to improve the appearance of all the manager pages
- Finally, Eathan Hodgkinson tested the manager page to confirm that all the manager-specific pages work
- As of now, all manager-specific pages work except the user verification page, which will be finished at a later date
- We tested every feature as it was added

Scrum Tasks Itemized List

- Lane Barnes
  - Created activity diagram for manager
  - Wrote part of the CSS code for manager pages
  - Wrote unit tests for the manager class
- Eathan Hodgkinson
  - Created class diagram for manager
  - Implemented part of the Django manager model
  - Made prototype user verification page
  - Linked the manager account and user verification pages to the Django manager model
  - Wrote part of the CSS code for manager pages
  - Tested the manager-specific pages for quality assurance
- Carter Parks
  - Implemented part of the Django manager model
- Evelyn Teeples
  - Gathered requirements for manager
  - Created the prototype account and tournament edit pages
- Josh Williams
  - Made the prototype drinks edit page
  - Linked the drinks edit page to the Django manager model

Resources

- Trello. (2019). https://trello.com
- Figma. (2019). https://www.figma.com
- Django documentation. (n.d.). https://docs.djangoproject.com/en/3.2/
- GitHub. (2013). https://github.com
- Lucidchart. (2019). https://www.lucidchart.com
- Google Drive. (2019). https://drive.google.com
- Discord. (n.d.). https://discord.com/
- Amazon Web Services (AWS). (2015). Amazon Web Services, Inc. https://aws.amazon.com
- Stack Overflow. (2019). https://stackoverflow.com
- Harshvijaythakkar. (2020, December 19). Dajngo [sic] with Nginx, Gunicorn. Analytics Vidhya. https://medium.com/analytics-vidhya/dajngo-with-nginx-gunicorn-aaf8431dc9e0

Questions?