CS4227- Software Design and Architecture

Car Valet System

Arnas Juravicius - 18257305
Dylan Kearney - 18227023
Cyiaph McCann – 17233453
Oisin McNamara - 18237398

Table of Contents

Business Scenario

This application is based on the project submitted in CS4125. We used this project as a baseline to implement the new requirements that CS4227 required. The business is largely the same with some additional features to accommodate the new design patterns.

Our application is a platform for the management and processing of a vehicle valet chain. This system will allow the business to manage and operate the business by tracking and monitoring the status of different services. Information such as customer subscription and financial reports will all be available.

Customers will be able to create an account. They will be given options to book the different services that this chain offers, for example, a full car wash with wax and an interior clean. There will also be a loyalty program implemented for returning customers giving them discounts on their orders. Customers will also be able to cancel any service booked up to 24hrs before the appointment or they can reschedule it if needed. If a particular store is experiencing high volumes of customer traffic, customers will have the option to be redirected to the nearest store with less traffic to reduce wait times for their service. A subscription model will be available with various levels that can be purchased all with several types of rewards. Customers will also be able to avail themselves of seasonal promotions which will help increase business.

From a store point of view, the staff will have the ability to view all existing bookings and their relevant details, such as the type of service ordered and the date of booking. Staff will have access to all admin functionality which will allow them to reschedule or cancel an appointment. Staff will have the ability to apply discounts to customer orders. Each store will be able to view their own profit margins, peak hours, and incoming traffic.

From a franchise point of view, users will be able to see several factors throughout the chain. Traffic flow, profit margins, customer reviews, customer subscriptions and operating statuses of each franchise.

The application will have a simple user interface for both the customers and staff. The UI will provide a user-friendly experience to make creating a booking as seamless as possible. There will also be information pages about the different services that are available and store locations throughout the country.

Functional Requirements

Listing of Requirements

- Creation of user accounts for customers.
- The ability to create a booking, 10% deposit fee must be paid.
- The ability to cancel a booking for free up until 24 hours, otherwise a 20% fee will be incurred.
- The ability to update a booking, the change must be approved by staff.
- User ability to view order summary/history.
- Staff ability to update job status.
    - Notification sent to customer on completion of each job in the valet.
- Staff can apply percentage discounts to customer orders.
- Staff get a 30% discount when creating a booking for themselves.
- Payment of booking.
    - Every 8th car wash is free.
- Email notifications of order confirmation and reminder notification 24 hours before scheduled appointment.
- Submission of reviews by customers.
- After the vehicle is collected, and payment is completed. An auto-generated invoice is sent to the customer.
- When creating a booking, if you book more than one car, a 50% discount will be applied.
- Subscription Model
    - For Bronze Subscriptions, a free car wash once a week
    - For Silver Subscriptions, a free car wash and interior clean once a week
    - For Gold Subscriptions, a free car wash, interior clean and car waxing once a week.

Use Case Diagram

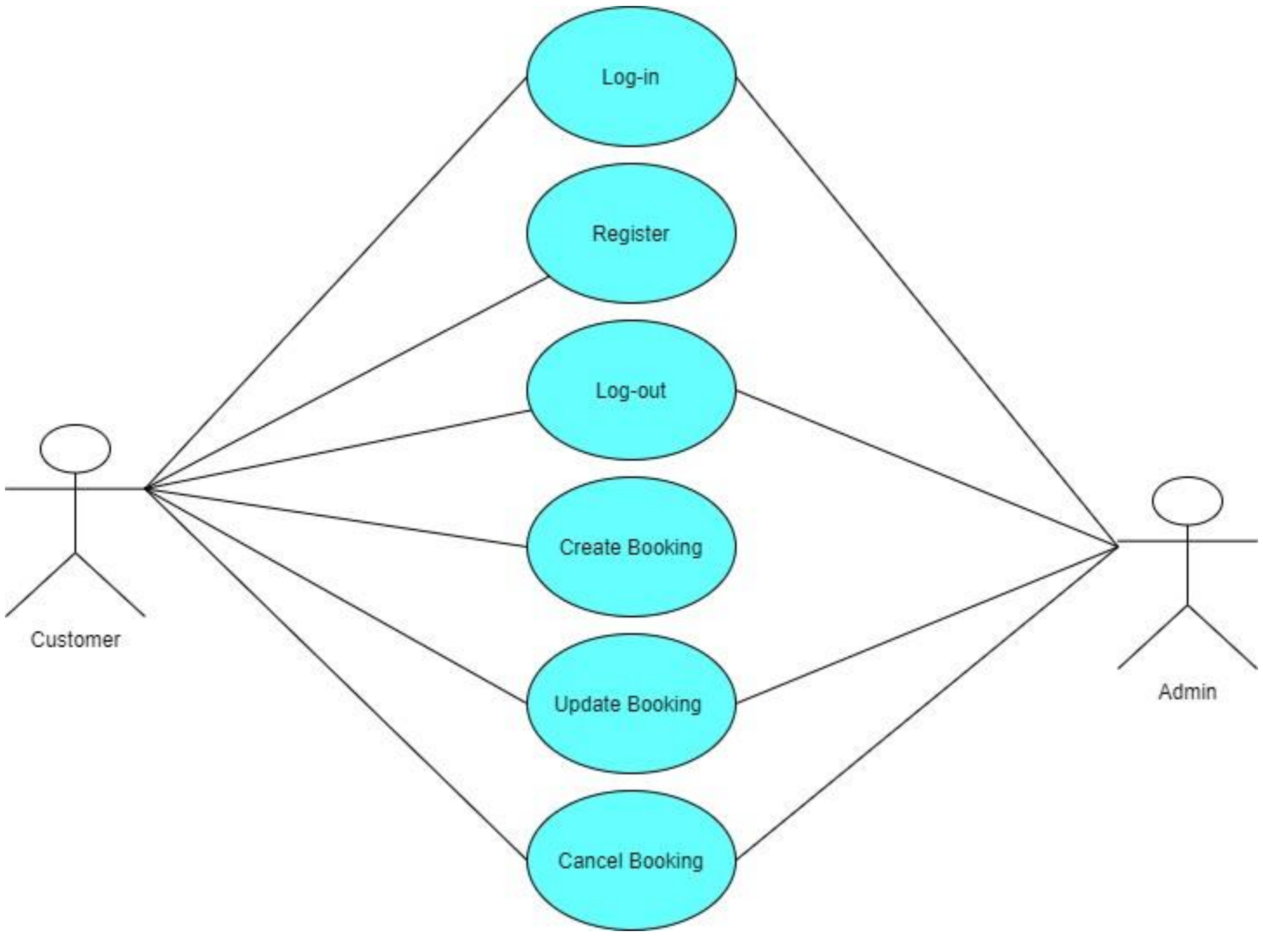


Figure 1 Use Diagram

Use Case Descriptions

Use case description: User Registers an Account	
Actor Action	**System Response**
1. User enters information and clicks register	**2.** System creates a new user account and logs user in

Use case description: User Logs In	
Actor Action	**System Response**
1. User Login.	2. System logs user into account.

Use case description: User Logs Out	
Actor Action	**System Response**

1. User Logout	**2.** System logs user out.

Use case description: User Creates a Booking	
Actor Action	**System Response**
1. Users select the type of service they want to book.	**2.** System returns a list of dates to select.
3. User selects the date they want.	4. System creates a successful booking.

Use case description: User Updates a Booking	
Actor Action	**System Response**
1. User updates an existing booking.	2. System returns information about changes made to existing booking.
3. User selects alternative booking time.	4. System updates booking to newly selected time.
5. User selects additional service	6. System updates booking and adds newly selected service.

Use case description: User Cancels a Booking	
Actor Action	**System Response**
1. User selects a booking to cancel.	2. System removes the booking.

Use case description: Staff Add Store to the Chain	
Actor Action	**System Response**
1. Staff enter details about a new store.	2. System creates a new store

Use case description: Staff Applies Discount to Booking	
Actor Action	**System Response**
1. Staff apply percentage discount to selected booking.	2. System updates total booking price with percentage reduction.

Use case description: User Writes a Review	
Actor Action	**System Response**
1. When a valet is finished, the user is prompted to create a review about the service/store.	2. The system logs the review, which can be used in analytics in the chain

Use case description: User can select a subscription	
Actor Action	**System Response**
1. Users can select a subscription model for the chain.	2. System logs customer as a subscriber to the service.

Detailed Use Case Descriptions

USE CASE 1	Book Valet	
Goal in context	Customer selects service that they want to book, then select an available time slot. Additionally, if a user books a service and they are a subscription holder they will receive a discount depending on the type of subscription they are holding	
Scope and level	Company, Summary	
Pre-conditions	Customer has account	
Success End Conditions	Customer successfully books appointment	
Failed End Condition	Customer is unable to book appointment	
Primary, Secondary, Actors	Customer, System	
Triggers	Customer selects book appointment	
Description	Step	Action
	1.	Customer selects valet service they want
	2.	Customer selects time and date for valet
	3.	Customer confirms booking
	4.	Customer pays deposit of 10% of booking price
Variations		**Branching Actions**
		Booking can pay using: Credit card Debit Card
Related information	Use case diagram – account creation	
Priority	High – directly affects sales of business if this side of the system is down	
Performance	Booking is made within 30 seconds to prevent a user booking the same slot	
Frequency	Depending on the number of sales in a day	
Channel to actors	n/a	
Open Issues	n/a	

USE CASE 2	Customer account creation
Goal in context	Customer wants to create an account to book valet services. They will be offered to purchase a subscription upon account creation
Scope and level	Company, Primary task
Pre-conditions	N/A
Success End Conditions	Customer creates account and gets access to the list of different valet locations and can book a service
Failed End Condition	Customer unable to book an account

Primary, Secondary, Actors	Customer, system	
Triggers	Click sign-up	
Description	**Step**	**Action**
	1.	Customer fills in details related to the account
	2.	Customer sets new password
	3.	Customer selects if they want to purchase a subscription based on bronze to gold.
	4.	Customer clicks accept Terms and Conditions and confirm details/pay
	5.	Customer brought to home screen
Variations		**Branching Actions**
		Phone Use web booking service

USE CASE 3	Valet stores Traffic monitoring	
Goal in context	When a user creates a booking at a store, if that store is maxed out for the hour, the user will be redirected to the closest store that is free for that hour.	
Scope and level	Company, Customer	
Pre-conditions	Must be a customer type user on the system	
Success End Conditions	Customer redirected to available, closest store	
Failed End Condition	Customer cannot be redirected.	
Primary, Secondary, Actors	Customer, system	
Triggers	Customer clicks to create booking	
Description	**Step**	**Action**
	1.	Customer selects a store.
	2.	Customer clicks on make booking.
	3.	System returns the closest available store if the one they selected is booked up.
	4.	Customer can still select the same store if they are willing to wait the extended time wait.
Variations		**Branching Actions**
		Phone
		Use web booking service

Non-functional requirements (NFR).

NFRs are constraints imposed on the system

Security

It is essential for us to address the security risks related to our web application. These risks include account passwords and user information. We must ensure that all sensitive data is encrypted and that user passwords are not visible at any time. User account and booking details must not be visible to other users on the platform. Restrictions will be placed on user accounts to ensure that said accounts do not have access to restricted API calls. Checks must be in place to ensure that inputs by the user are not some forms of malicious code to prevent injection.

Usability

The application developed should be straightforward for the customer to use. It should be quickly learnable and easy to navigate through the interface. The user should be able to book the desired service with ease and without confusion. Investing in the user experience should deliver a usable and accessible system.

Availability

The time span during which the system operates normally and without failure is referred to as availability. The applications proportion of total application downtime over a specific time is used to calculate availability. Failures, bugs, malicious code, upgrades all have an impact on availability.

Maintainability

Maintainability refers to an application's ability to tackle changes and updates with minimal effort. This is defined as the degree of versatility in which the application may be altered, whether to add new features or bug patches. Improved maintainability can boost availability and cut down on runtime errors. The entire quality of an application determines its maintainability.

Quality Tactics

Quality tactics will discuss how we will try implement the NFRs in our system.

Security

To implement the security features, we will have to consider several factors when designing the system. OWASP top ten security risks was a big inspiration to us when developing the quality tactics. All passwords in the system should be salted and hashed to provide a prominent level of security. On the user side, all passwords and sensitive data should be hidden or asterisked out. Other details regarding users such as address, and phone numbers should not be visible to other users (owasp, 2021) (cloudflare, 2021).

Usability

For the project we will implement a simple user interface for the customer / staff user. The UI will be easy to navigate and easy to understand. A simple login will bring either the customer or staff to their relevant view which will display their options. Staff will be able to view the different setting and to be able to access and edit information relevant to any customer or reservations they have. Customer will see the options for booking different services and also the available date/times for said services.

Availability

When creating the application, extensive code reviews and refactoring should be carried out when implementing features. Code reviews and refactoring could reduce the possibility of failures and bugs appearing in the code. A reduction in these will increase the availability of the application.

Maintainability

The application should be able to tackle change easily. This will be done by creating software with high cohesion and minimal dependency. Furthermore, the application should have modularity with defined interfaces. This makes the software easier to maintain and upgrade. It will also speed up time-to-market.

Architectural and Design Patterns

We implemented a total of 8 design patterns. We have already implemented both composite, visitor and factory in CS4125 so we could apply these to this project also. We found that implementing the patterns gave us a better idea of how the patterns function within the code. The patterns also made it easier to understand how we could implement the new features into our project. All code related to the patterns is found down below in the interesting elements of code section.

Interceptor Architectural pattern

The interceptor architectural pattern enables services to be introduced to a framework discreetly and dynamically started when specific events occur (Eichberg, M, 2022). The interceptor architecture pattern uses context objects, dispatchers, and concrete interceptors in orchestration to carry out out-of-band services. For our project we done a per event interceptor. This means that it has one interface and many context objects which are registered with the dispatcher. The concrete interceptor can then query these context objects via the dispatcher to retrieve information from the concrete framework. We used the interceptor to retrieve information about customer emails and store names via the context object.

Command

Command is a behavioural design pattern that converts a request into a stand-alone object with all the request's data. You can pass requests as method arguments, postpone, or queue a request's execution, and support undoable operations with this transformation (Refactoring Guru, 2022). For the project we used the command design pattern for creating buttons for the admin which could change the state of a booking using the command execute method. We also included an undo method which would revert the state back to the original state.

Pluggable Adapter

The adapter design pattern is a structural design pattern that allows items with mismatched interfaces to work together such as 3rd party services working with your system (Refactoring Guru, 2022). To create a pluggable adapter the adapter must define a narrow interface for the adaptee and use abstract operations in the target. For the project, we used the pluggable adapter with the visitor to get vasts amount of information which we then could export to a csv file.

Composite

The composite is a structural design pattern that allows you to group elements into tree structures and manipulate them as if they were distinct, individual objects (GeeksforGeeks, 2021). Here our thinking was that the base wash would be the top of the composite tree. The base wash could have a basic time duration of 10 minutes. We then branch off two composites, interior, and exterior. The interior composite would have vacuum, leather, and steam clean nodes. The exterior composite would have the polish, wax, and wash nodes. Each node would add duration in minutes to the composite. The base wash would add duration of both composites into a final duration in minutes.

Builder

Builder is a design pattern that allows you to build complicated objects in stages. Using the same building code, you can create different types and representations of the same object. (Refactoring Guru, 2022) For the project, we used the builder to build complex valet objects using the decorator and composite design patterns. Depending on the user's membership type such as gold, the builder would build a different valet object for each membership type. This could be used to seasonal discounts.

Factory

The Factory Method design pattern is a creational design pattern that offers an interface for producing objects in a superclass while allowing subclasses to choose the type of objects generated (Refactoring Guru, 2022).
In our project, the user is created using the signup form. This form creates a User in the database. Here a user factory design pattern was used to create the customer or staff depending on the email provided on registration.

Visitor

The Visitor pattern is a behavioural design pattern that allows you to decouple algorithms from the objects they act on (Refactoring Guru, 2022). Instead of attempting to integrate new functionality into existing classes, the Visitor pattern advises creating a new class named visitor. The original object that had to execute the action is now supplied as an argument to one of the visitor's methods, giving the method access to all the object's data. For example, in our project, we used the visitor as an ability to export data from the database to a CSV file. This enabled for extensibility and modification easily. The visitor works with the pluggable adapter to write to a CSV file.

Prototype

Prototype is a design pattern that allows you to clone existing objects without having to rely on their classes in your code. (Refactoring Guru, 2022). The cloning operation is delegated to the objects that require cloning using the Prototype pattern. An abstract method clone () is defined in the Prototype parent class and is implemented by the subclasses.

In our project, we implemented the prototype so that we could make clones of stores. We have ChainStores that could be either a LargeStore or a MiniStore where a LargeStore has one or more automatic valeting devices and MiniStores are basic manual valet stores. ChainStore has the abstract method clone () that is implemented in the subclasses LargeStore and MiniStore. The clone method returns a copy of the object it was called on

System Architecture

Discussion

For the application, we decided to opt for the Django Framework as the development environment.Django is open source, and it is a high-level Python web framework. Django uses its own implementation of MVC known as an MVT, Model-View-Template.

The system is split into three layers. The user interface layer, a database layer, and the business layer.Each layer runs independently which allows for better maintainability for the system. The UI (User Interface) layer is handled by a Django Template and URLs. A Django template is a presentation file which can be rendered using an HTML file. A Django view file is the controller for the business logic.

The business layer computes all the business logic for the system. The UI layer interacts with the business layer and the business layer can communicate with the database layer. The UI layer can nevercommunicate with the database layer. The business layer will perform all the business logic and is the layer accountable for most of the performance.

Architecture

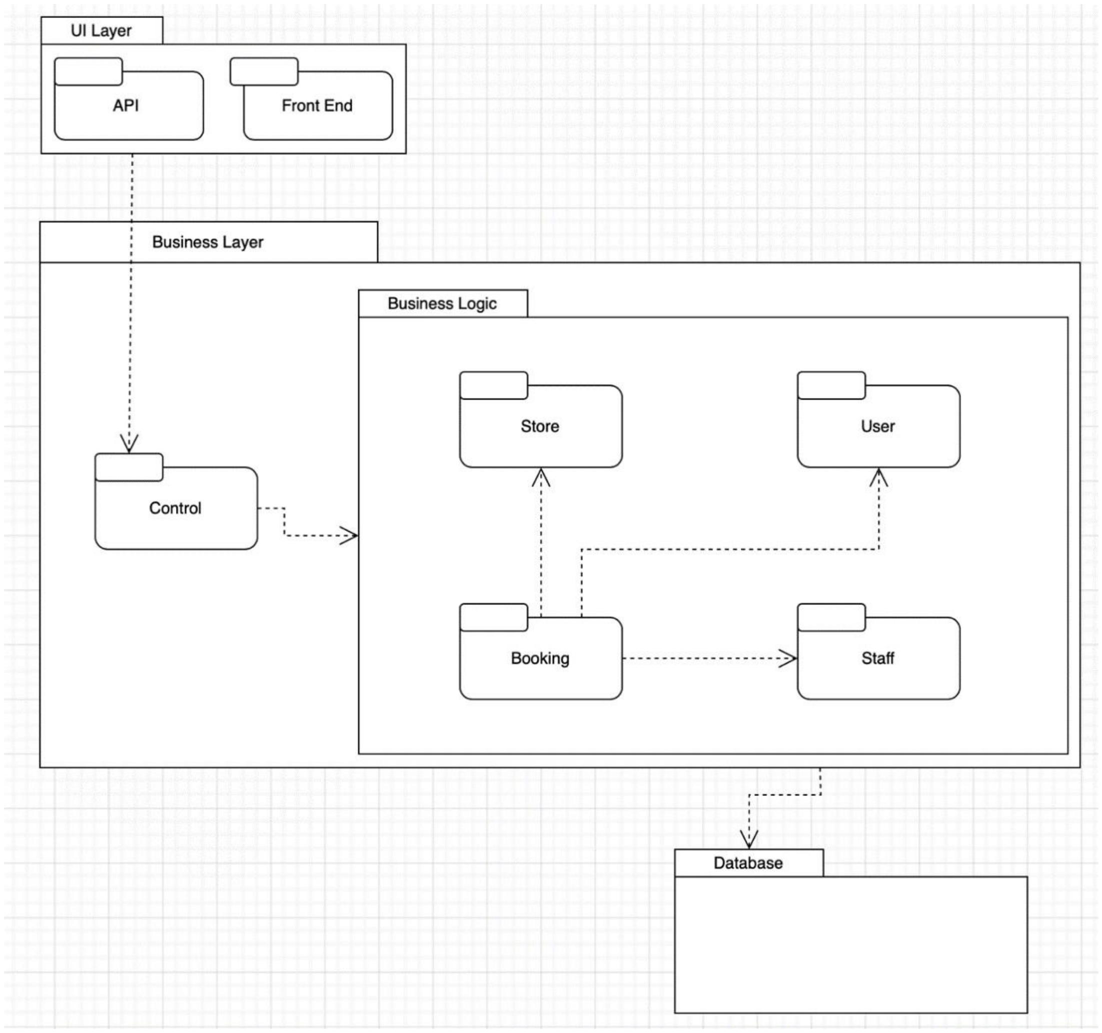


Figure 2 System Architecture

Structural and Behavioural Diagrams

Class Diagram

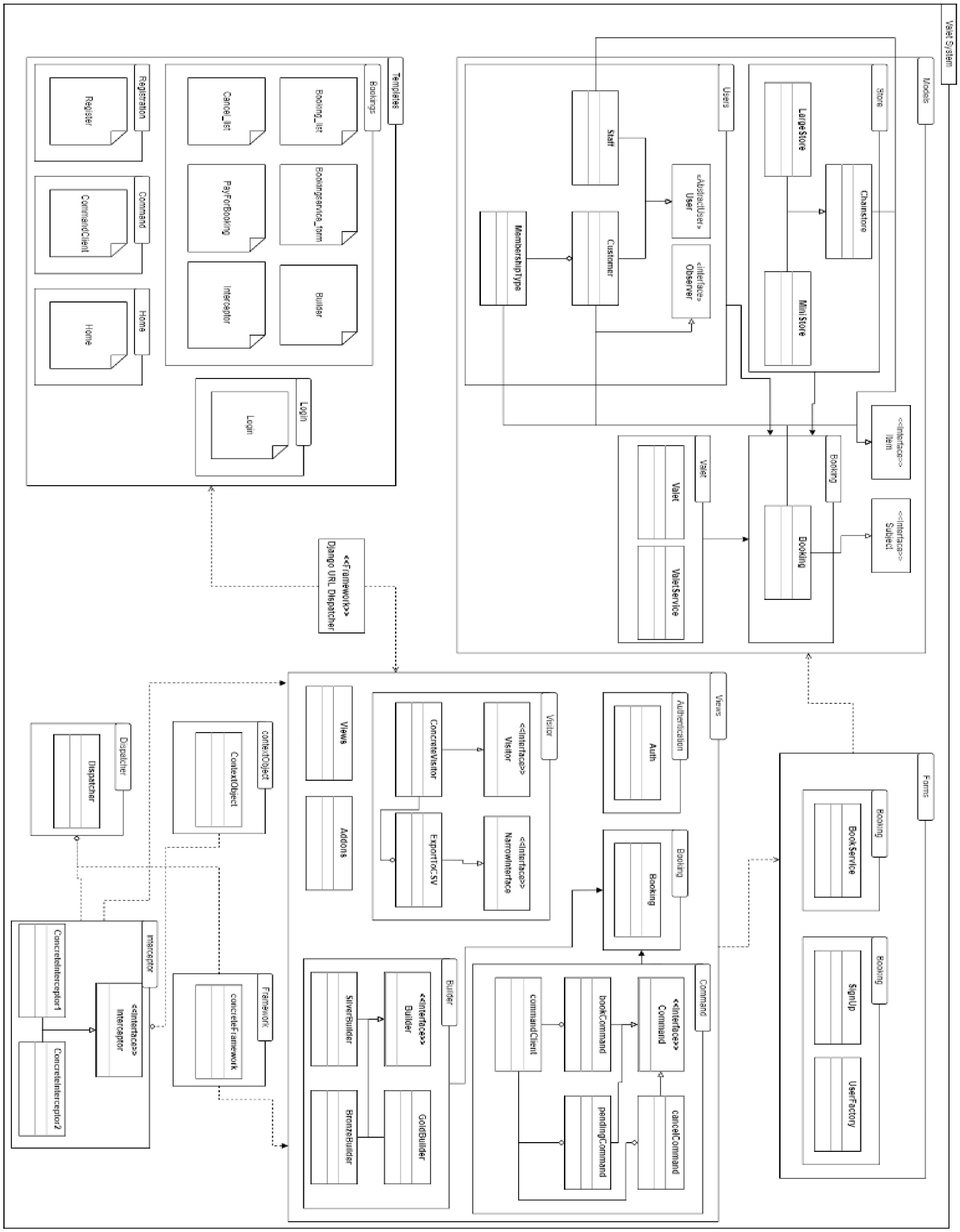


Figure 3 System Class Diagram

Interaction Diagram

Sequence Diagram for User Creation use case

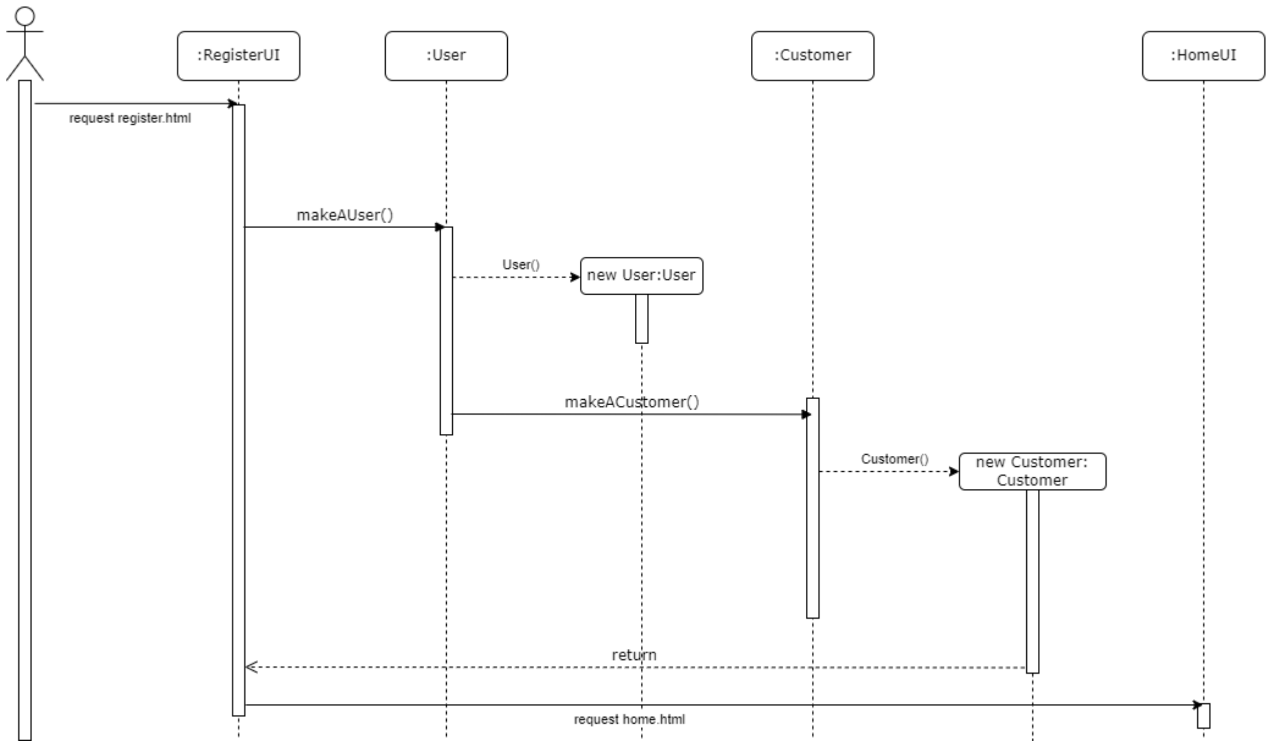
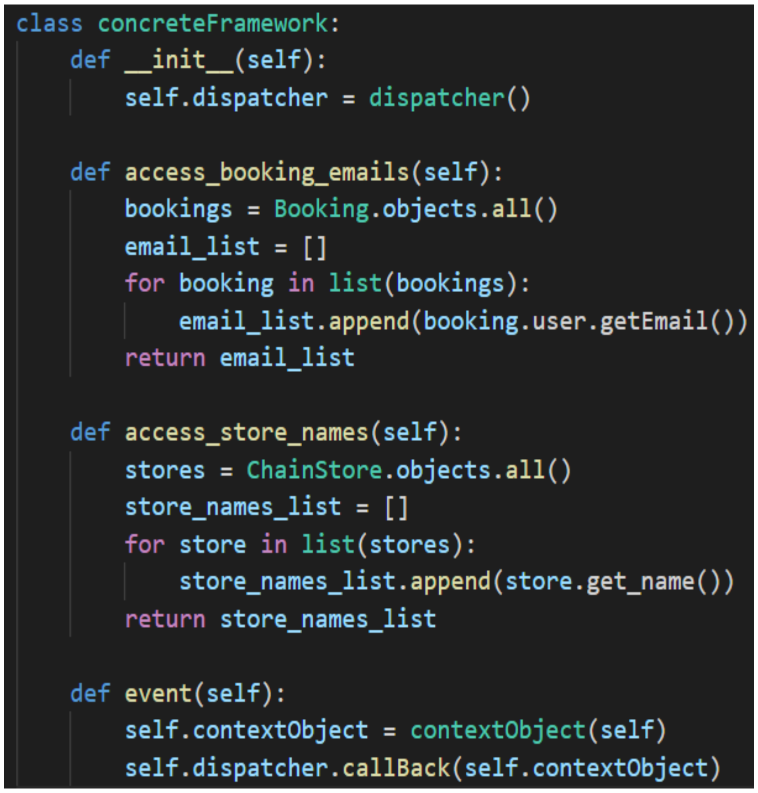


Figure 4 Use Case Sequence Diagram

Interesting Implementation Elements

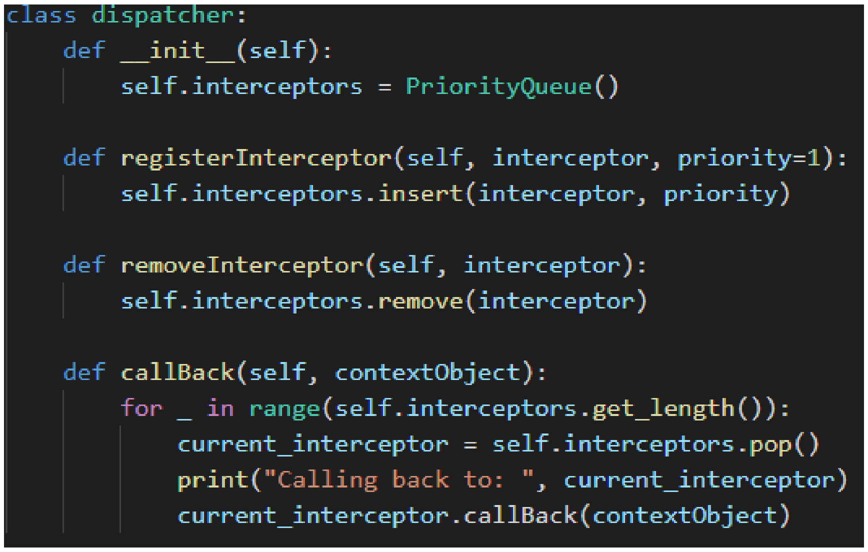
Interceptor Architectural Pattern

Here we can see the concrete framework and its out-of-band services. One is to get customer emails and the other returns all the store names. Lastly, since our interceptor is in response to an event, the concrete framework passes the context object to the dispatcher. The dispatcher then iterates through the list of concrete interceptors calling back to the concrete interceptor passing reference to the context object.



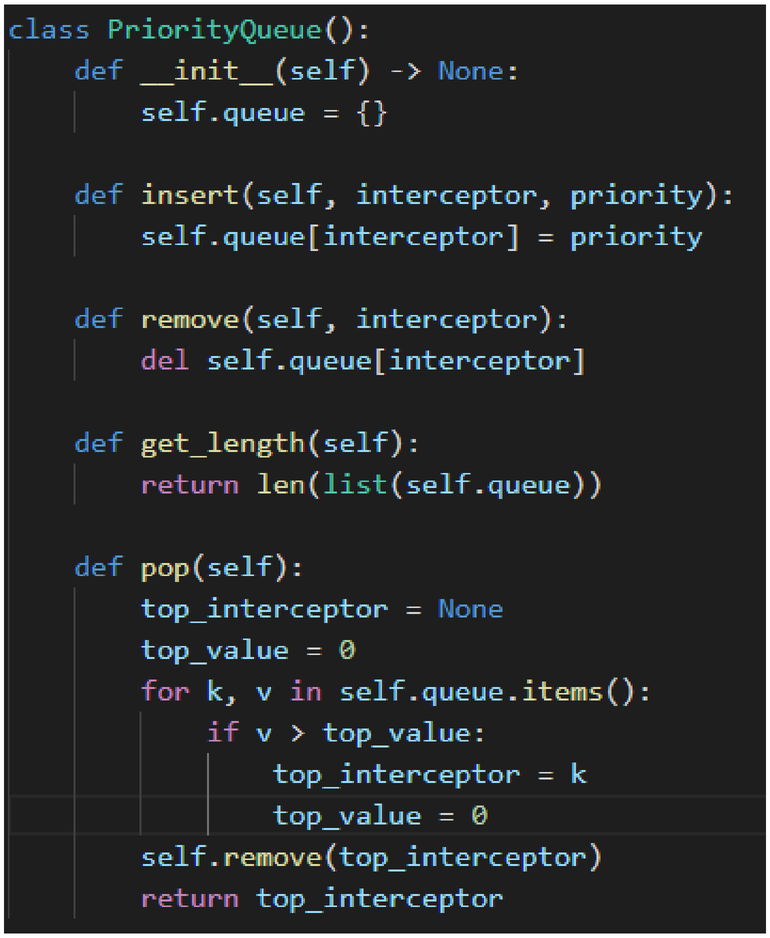
```python
class concreteFramework:
    def __init__(self):
        self.dispatcher = dispatcher()

    def access_booking_emails(self):
        bookings = Booking.objects.all()
        email_list = []
        for booking in list(bookings):
            email_list.append(booking.user.getEmail())
        return email_list

    def access_store_names(self):
        stores = ChainStore.objects.all()
        store_names_list = []
        for store in list(stores):
            store_names_list.append(store.get_name())
        return store_names_list

    def event(self):
        self.contextObject = contextObject(self)
        self.dispatcher.callBack(self.contextObject)
```

Figure 5 Concrete Framework Code



```python
class dispatcher:
    def __init__(self):
        self.interceptors = PriorityQueue()

    def registerInterceptor(self, interceptor, priority=1):
        self.interceptors.insert(interceptor, priority)

    def removeInterceptor(self, interceptor):
        self.interceptors.remove(interceptor)

    def callBack(self, contextObject):
        for _ in range(self.interceptors.get_length()):
            current_interceptor = self.interceptors.pop()
            print("Calling back to: ", current_interceptor)
            current_interceptor.callBack(contextObject)
```

Figure 6 Dispatcher Code

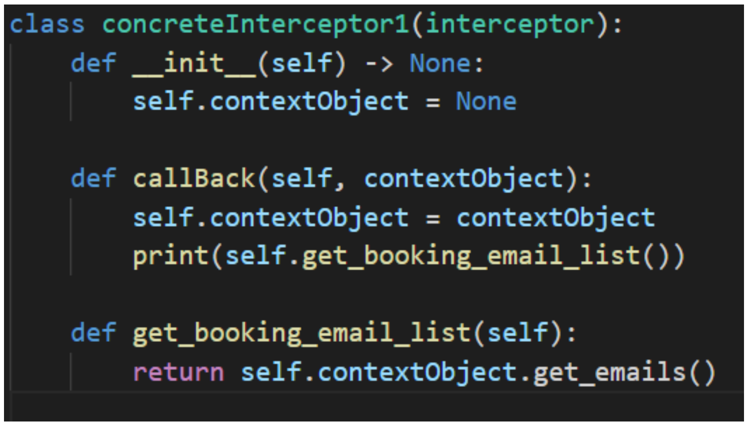


```python
class PriorityQueue():
    def __init__(self) -> None:
        self.queue = {}

    def insert(self, interceptor, priority):
        self.queue[interceptor] = priority

    def remove(self, interceptor):
        del self.queue[interceptor]

    def get_length(self):
        return len(list(self.queue))

    def pop(self):
        top_interceptor = None
        top_value = 0
        for k, v in self.queue.items():
            if v > top_value:
                top_interceptor = k
                top_value = 0
        self.remove(top_interceptor)
        return top_interceptor
```

Figure 7 Dispatcher Priority Queue Code

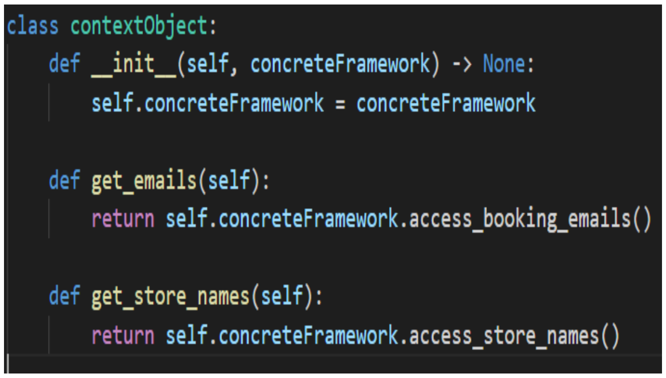
The dispatcher keeps the list of interceptors as a prioritised list. We created the priority queue class with methods to insert, remove and pop interceptors. When registering the interceptor with the dispatcher, it is given a priority number. When calling back to the interceptors, the interceptor with the highest priority is popped off the queue until all interceptors are popped off.

This concrete interceptor calls back to the context object querying the concrete framework via the context object. The concrete interceptor can now get the list of emails produced by the concrete framework. This is a fully working interceptor querying the framework via the context object with the context object acting as a middleman between the concrete interceptors and the concrete framework.



```python
class concreteInterceptor1(interceptor):
    def __init__(self) -> None:
        self.contextObject = None

    def callBack(self, contextObject):
        self.contextObject = contextObject
        print(self.get_booking_email_list())

    def get_booking_email_list(self):
        return self.contextObject.get_emails()
```

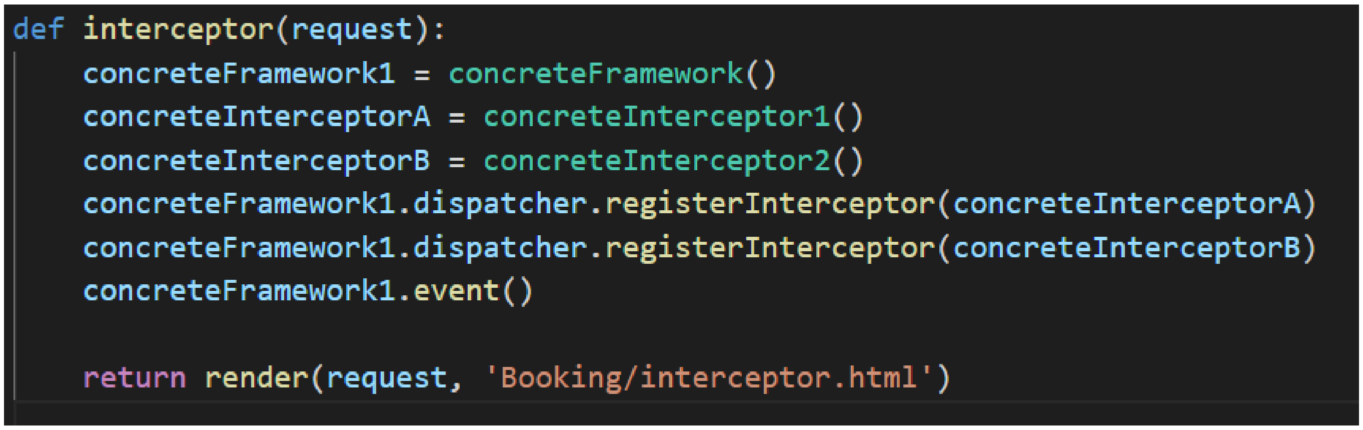
Figure 9 Concrete Interceptor



```python
class contextObject:
    def __init__(self, concreteFramework) -> None:
        self.concreteFramework = concreteFramework

    def get_emails(self):
        return self.concreteFramework.access_booking_emails()

    def get_store_names(self):
        return self.concreteFramework.access_store_names()
```

Figure 8 Context Object

Lastly, we see the interceptor in action. We first create the concrete framework and create two concrete interceptors. We then register both with the dispatcher and trigger the event. This prints out the emails and store names.

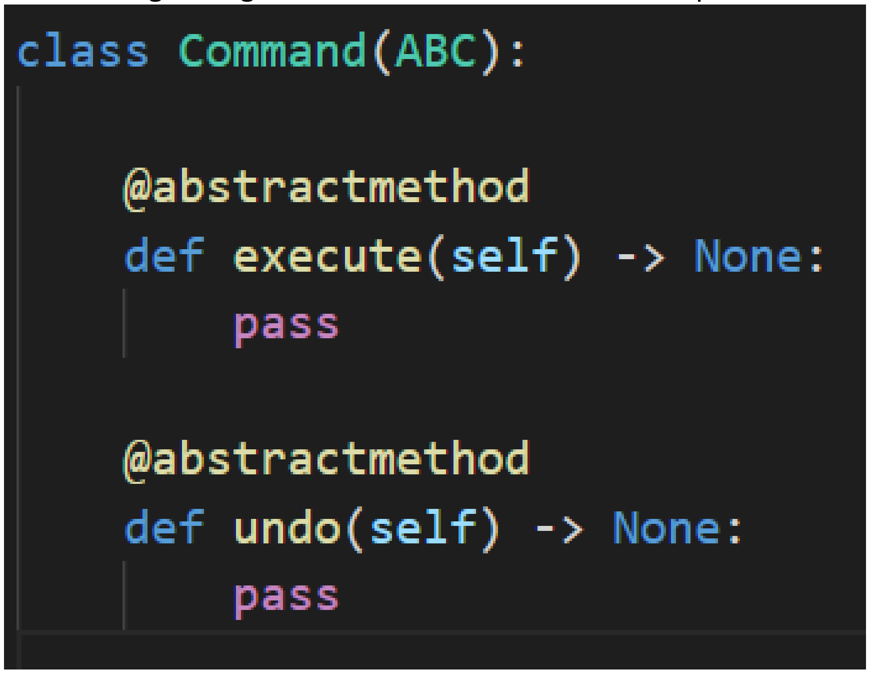


```python
def interceptor(request):
    concreteFramework1 = concreteFramework()
    concreteInterceptorA = concreteInterceptor1()
    concreteInterceptorB = concreteInterceptor2()
    concreteFramework1.dispatcher.registerInterceptor(concreteInterceptorA)
    concreteFramework1.dispatcher.registerInterceptor(concreteInterceptorB)
    concreteFramework1.event()

    return render(request, 'Booking/interceptor.html')
```

Figure 10 Interceptor Invocation

Command

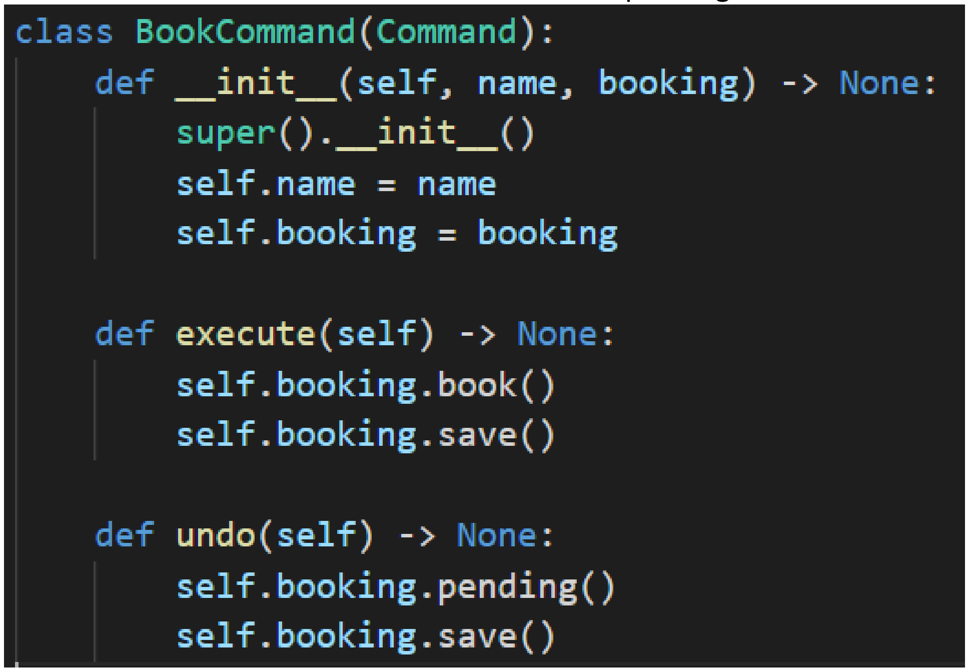
For the command, we created an admin page which could move a booking to the pending, cancelled and booked stage using buttons. The command interface provides two methods, execute and undo.



```python
class Command(ABC):

    @abstractmethod
    def execute(self) -> None:
        pass

    @abstractmethod
    def undo(self) -> None:
        pass
```

Figure 11 Command Interface

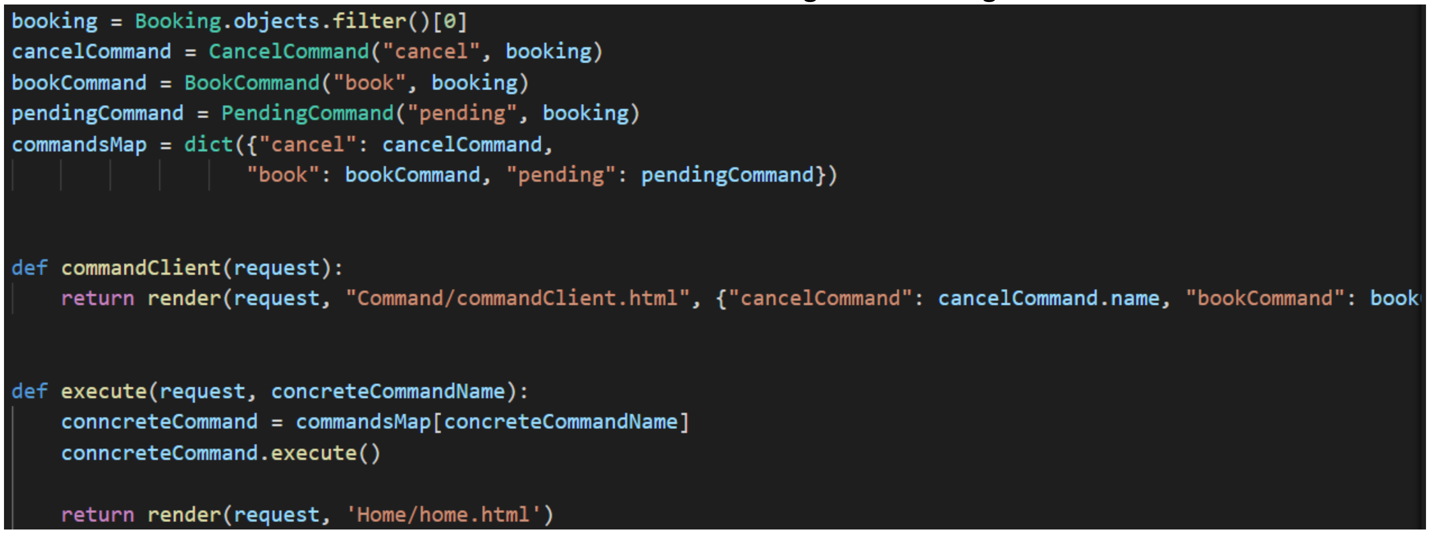
There we three commands that would inherit the command interface. BookCommand, cancelCommand and pendingCommand. Below is an example of the bookCommand. The execute changes that bookings state to booked and the undo reverts it back to pending.



```python
class BookCommand(Command):
    def __init__(self, name, booking) -> None:
        super().__init__()
        self.name = name
        self.booking = booking

    def execute(self) -> None:
        self.booking.book()
        self.booking.save()

    def undo(self) -> None:
        self.booking.pending()
        self.booking.save()
```

Figure 12 Concrete Command

Below, we create three concrete commands. We then store the three commands in a dictionary/hash map. Depending in the html which button was clicked it would pass the name of the button, i.e., book to the execute method. This method would then look up the command in the commandsMap and perform that execute method of that command. This would then change the bookings state to booked.

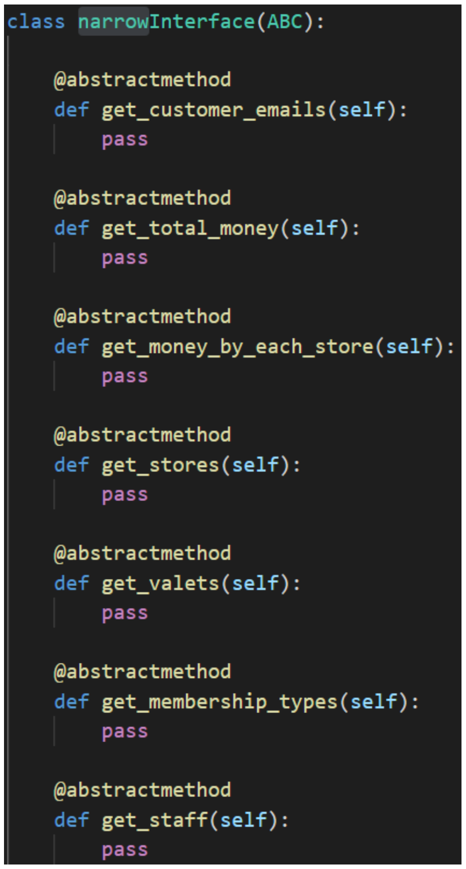


```
booking = Booking.objects.filter()[0]
cancelCommand = CancelCommand("cancel", booking)
bookCommand = BookCommand("book", booking)
pendingCommand = PendingCommand("pending", booking)
commandsMap = dict({"cancel": cancelCommand,
                    "book": bookCommand, "pending": pendingCommand})


def commandClient(request):
    return render(request, "Command/commandClient.html", {"cancelCommand": cancelCommand.name, "bookCommand": book


def execute(request, concreteCommandName):
    conncreteCommand = commandsMap[concreteCommandName]
    conncreteCommand.execute()

    return render(request, 'Home/home.html')
```

Figure 13 Command Invocation

Pluggable Adapter

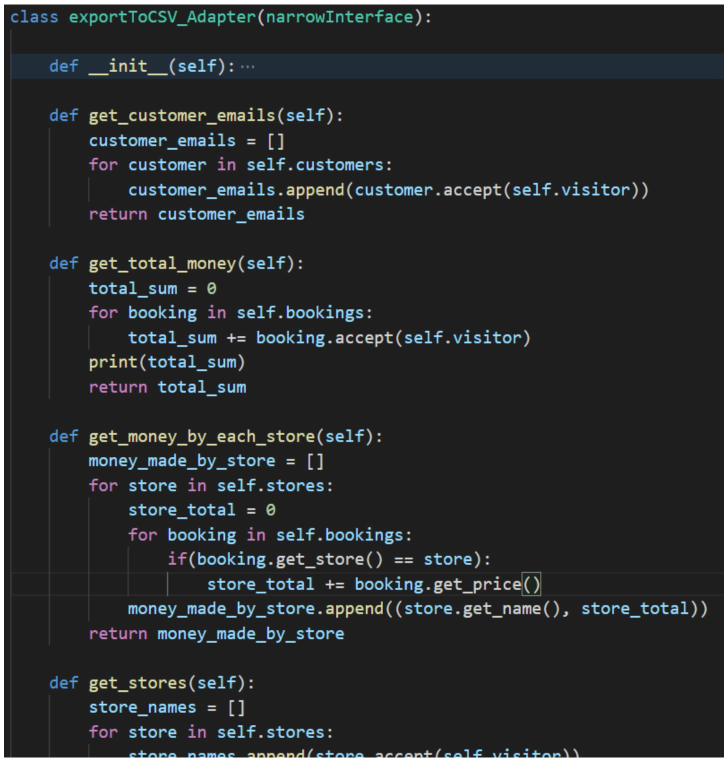
This was one of the most interesting design patterns we implemented. We built on last year's visitor design pattern which will be explained below. First, we start by creating an abstract interface. This is all the information which we will want to export from the database to a CSV file for an external system.



```
class narrowInterface(ABC):

    @abstractmethod
    def get_customer_emails(self):
        pass

    @abstractmethod
    def get_total_money(self):
        pass

    @abstractmethod
    def get_money_by_each_store(self):
        pass

    @abstractmethod
    def get_stores(self):
        pass

    @abstractmethod
    def get_valets(self):
        pass

    @abstractmethod
    def get_membership_types(self):
        pass

    @abstractmethod
    def get_staff(self):
        pass
```

Figure 14 Adapter Interfacef

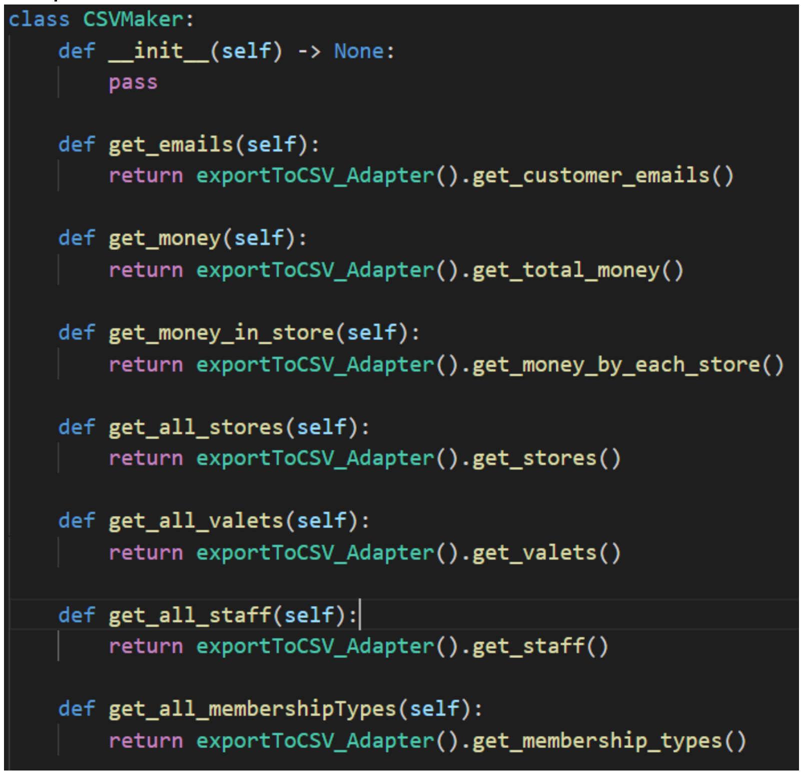
We know this is a pluggable adapter since the adapter inherits from the narrow interface using abstract operations. The adapter uses the visitor to return the information to the adapter. Below is an example of some of the methods. For each customer we visit the vistor using the accept method which returns the email of the customer. Code on next page.



```python
class exportToCSV_Adapter(narrowInterface):

    def __init__(self):...

    def get_customer_emails(self):
        customer_emails = []
        for customer in self.customers:
            customer_emails.append(customer.accept(self.visitor))
        return customer_emails

    def get_total_money(self):
        total_sum = 0
        for booking in self.bookings:
            total_sum += booking.accept(self.visitor)
        print(total_sum)
        return total_sum

    def get_money_by_each_store(self):
        money_made_by_store = []
        for store in self.stores:
            store_total = 0
            for booking in self.bookings:
                if(booking.get_store() == store):
                    store_total += booking.get_price()
            money_made_by_store.append((store.get_name(), store_total))
        return money_made_by_store

    def get_stores(self):
        store_names = []
        for store in self.stores:
            store_names.append(store.accept(self.visitor))
```

Figure 15 Pluggable Adapter

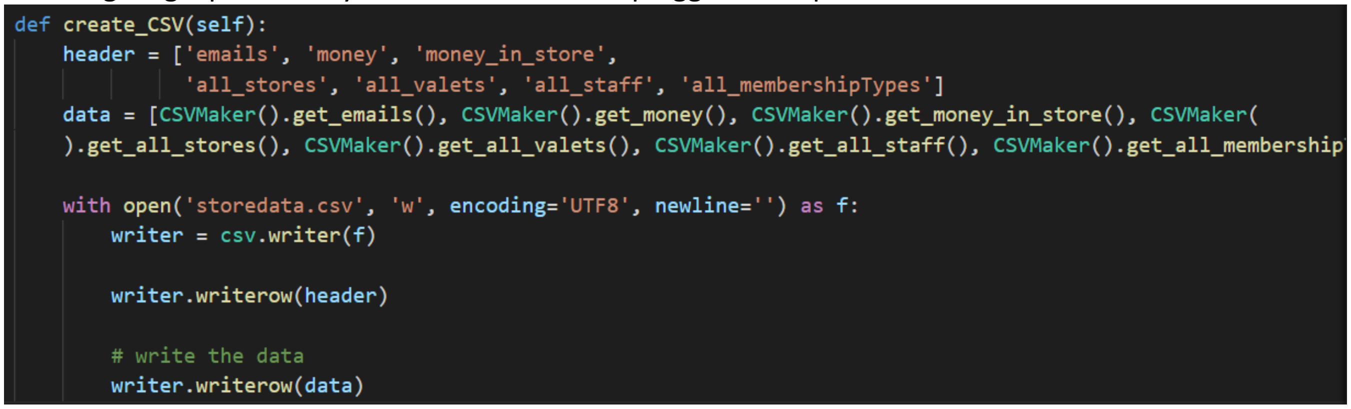
We then use the adapter in the CSVMaker. This gives the CSV maker all the information retrieved by the adapater.



```python
class CSVMaker:
    def __init__(self) -> None:
        pass

    def get_emails(self):
        return exportToCSV_Adapter().get_customer_emails()

    def get_money(self):
        return exportToCSV_Adapter().get_total_money()

    def get_money_in_store(self):
        return exportToCSV_Adapter().get_money_by_each_store()

    def get_all_stores(self):
        return exportToCSV_Adapter().get_stores()

    def get_all_valets(self):
        return exportToCSV_Adapter().get_valets()

    def get_all_staff(self):
        return exportToCSV_Adapter().get_staff()

    def get_all_membershipTypes(self):
        return exportToCSV_Adapter().get_membership_types()
```

Figure 16 Adapter Target

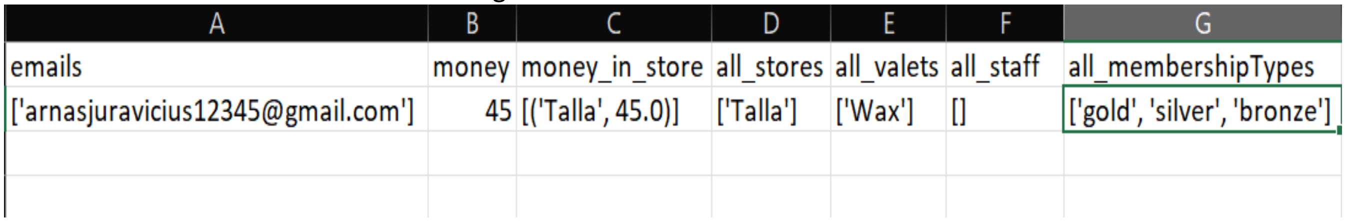
We can then call on all these methods and write to a CSV file. This CSV file can then be used in a 3rd party service giving a particularly effective use for the pluggable adapter.



```
def create_CSV(self):
    header = ['emails', 'money', 'money_in_store',
              'all_stores', 'all_valets', 'all_staff', 'all_membershipTypes']
    data = [CSVMaker().get_emails(), CSVMaker().get_money(), CSVMaker().get_money_in_store(), CSVMaker(
    ).get_all_stores(), CSVMaker().get_all_valets(), CSVMaker().get_all_staff(), CSVMaker().get_all_membership

    with open('storedata.csv', 'w', encoding='UTF8', newline='') as f:
        writer = csv.writer(f)

        writer.writerow(header)

        # write the data
        writer.writerow(data)
```

Figure 17 CSV Writer

Here we can see the code in effect writing to a CSV file.



A	B	C	D	E	F	G
emails	money	money_in_store	all_stores	all_valets	all_staff	all_membershipTypes
['arnasjuravicius12345@gmail.com']	45	[('Talla', 45.0)]	['Talla']	['Wax']	[]	['gold', 'silver', 'bronze']

Figure 18 CSV File

Composite

The composite design pattern was implemented last semester. Here we created a composite tree for valet services with the base wash being the top of the tree branching into interior and exterior.

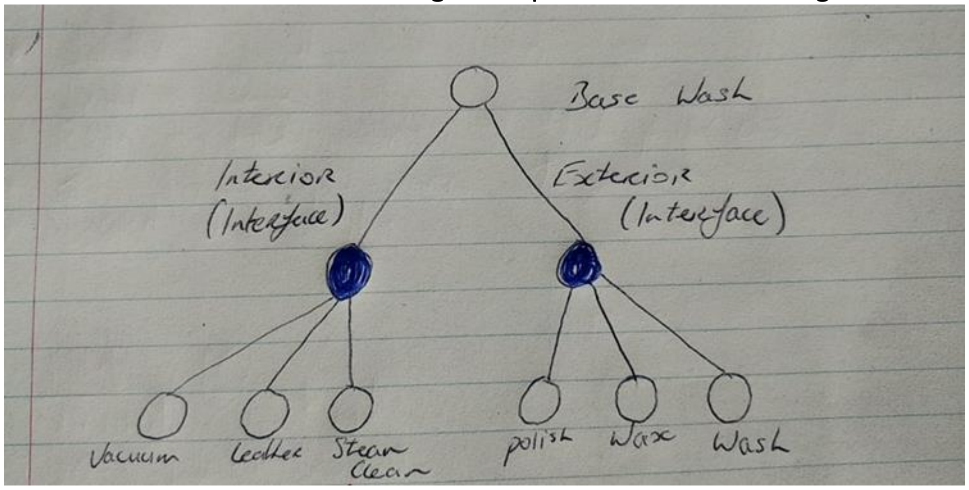
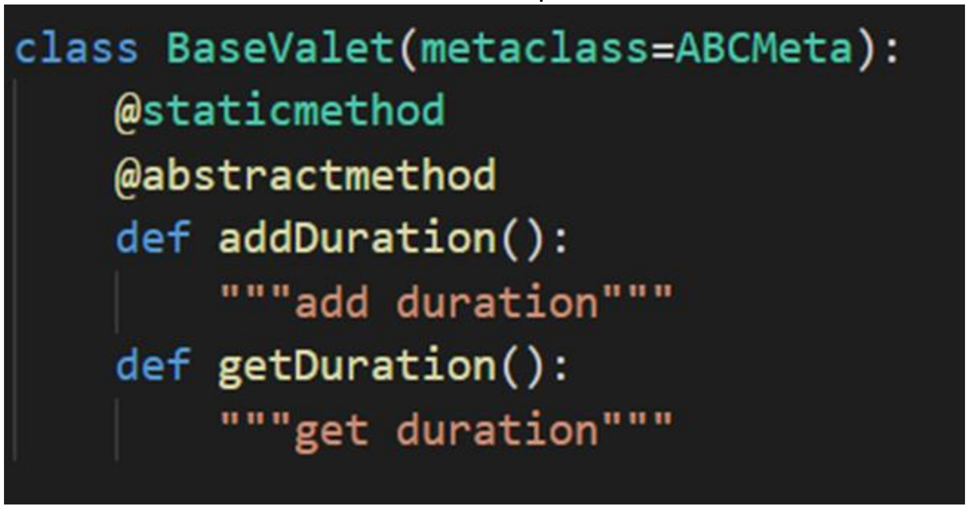


Figure 19 Composite Structure

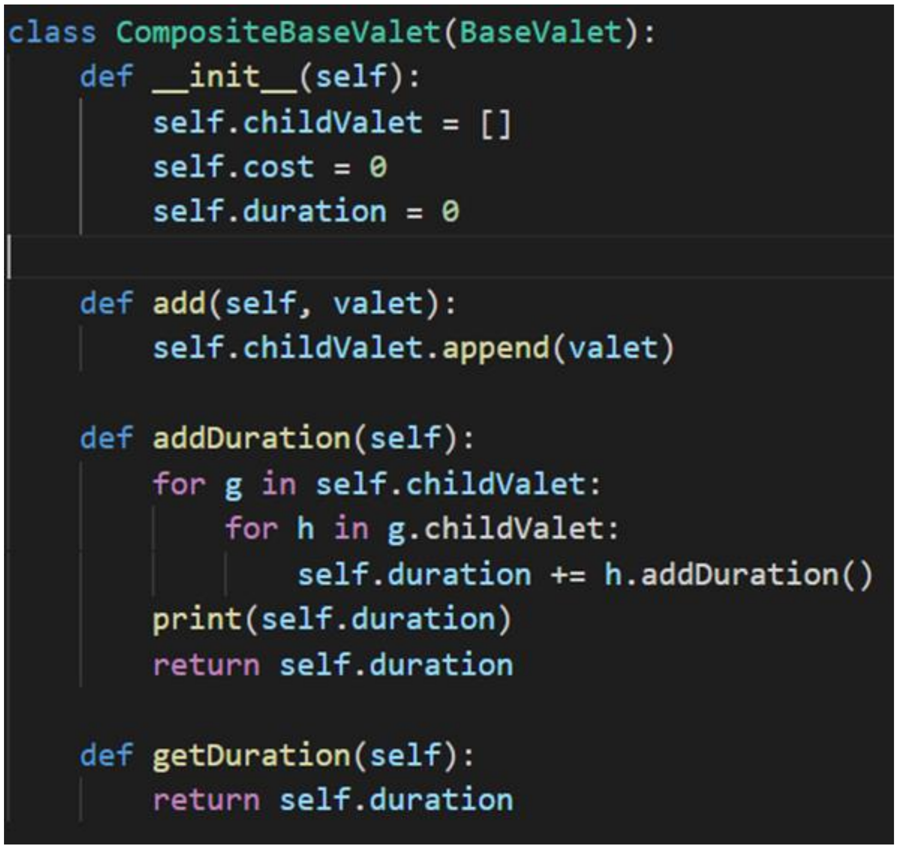
We first create an interface for each composite node to inherit.



```
class BaseValet(metaclass=ABCMeta):
    @staticmethod
    @abstractmethod
    def addDuration():
        """add duration"""
    def getDuration():
        """get duration"""
```

Figure 20 Composite Interface

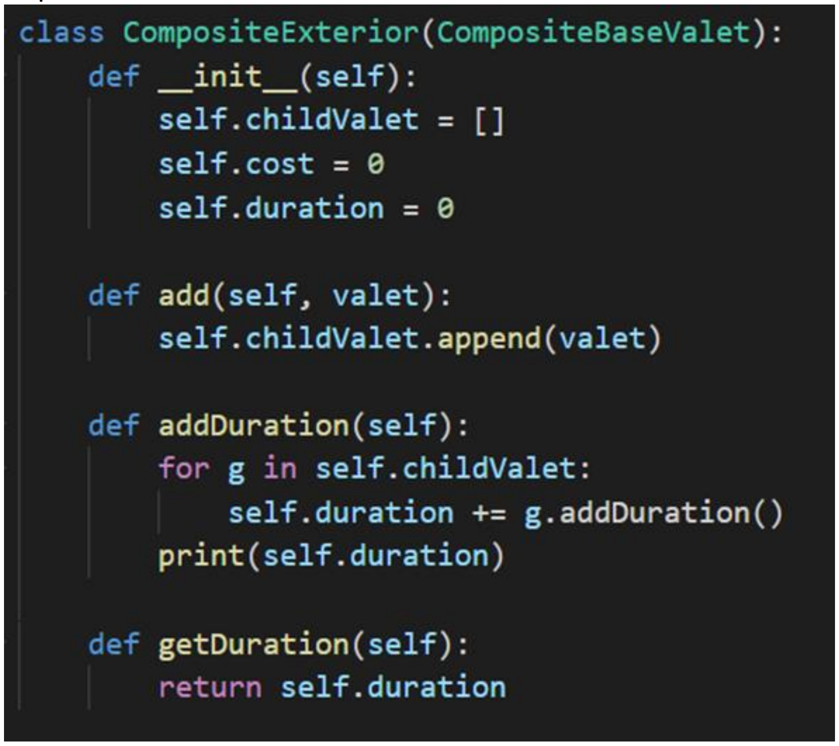
Here is an example of the base node implementing them methods adding the time for each node in its composite node.



```python
class CompositeBaseValet(BaseValet):
    def __init__(self):
        self.childValet = []
        self.cost = 0
        self.duration = 0

    def add(self, valet):
        self.childValet.append(valet)

    def addDuration(self):
        for g in self.childValet:
            for h in g.childValet:
                self.duration += h.addDuration()
        print(self.duration)
        return self.duration

    def getDuration(self):
        return self.duration
```

Figure 21 Concrete composite

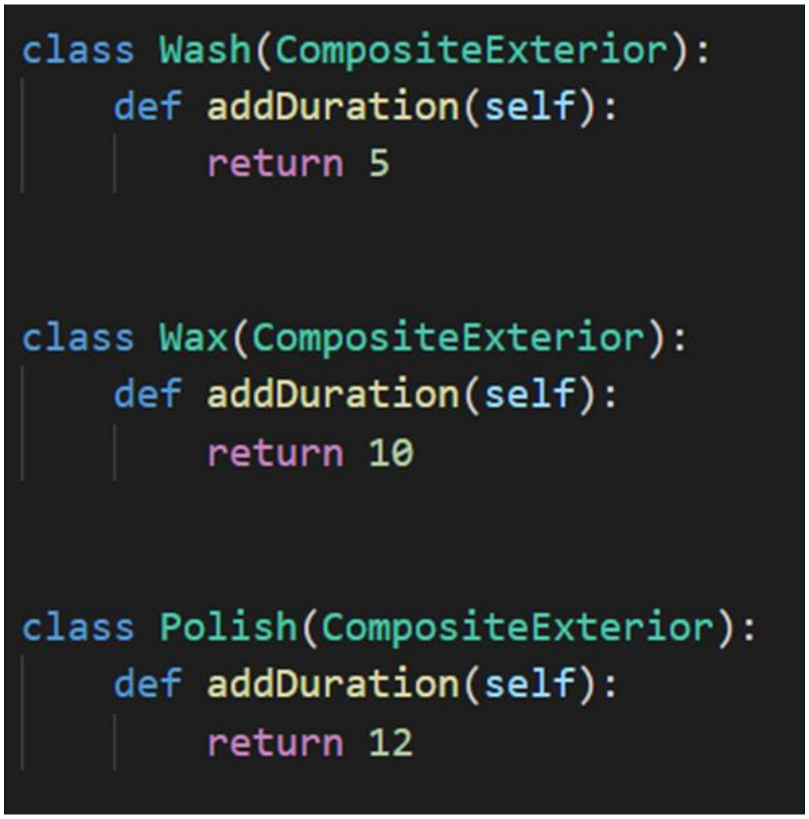
Here is the next composite node of the base used for the exterior valets. The methods are also implemented.



```python
class CompositeExterior(CompositeBaseValet):
    def __init__(self):
        self.childValet = []
        self.cost = 0
        self.duration = 0

    def add(self, valet):
        self.childValet.append(valet)

    def addDuration(self):
        for g in self.childValet:
            self.duration += g.addDuration()
        print(self.duration)

    def getDuration(self):
        return self.duration
```

Figure 22 Concrete composite

This is the bottom of the tree from exterior composite node. Each node adds different amounts of time to the duration. The total duration of the tree is then calculated at the top giving a value of time to the user.

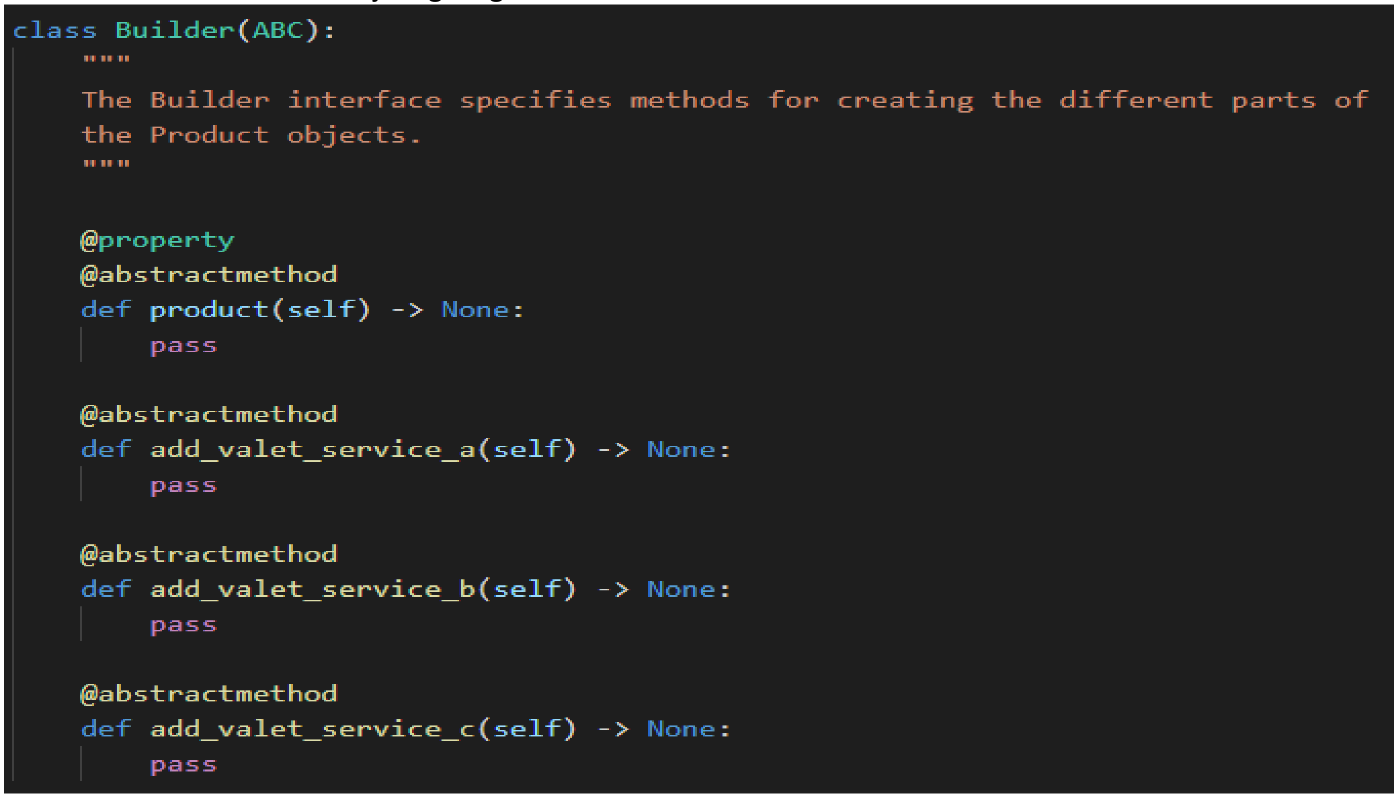


```python
class Wash(CompositeExterior):
    def addDuration(self):
        return 5


class Wax(CompositeExterior):
    def addDuration(self):
        return 10


class Polish(CompositeExterior):
    def addDuration(self):
        return 12
```

Figure 23 Concrete Node

Builder

For the builder we got it working with the decorator design pattern and with the composite design pattern. The thought for this was depending on the user's membership type the builder would build them a different seasonal valet object giving the user seasonal discounts.

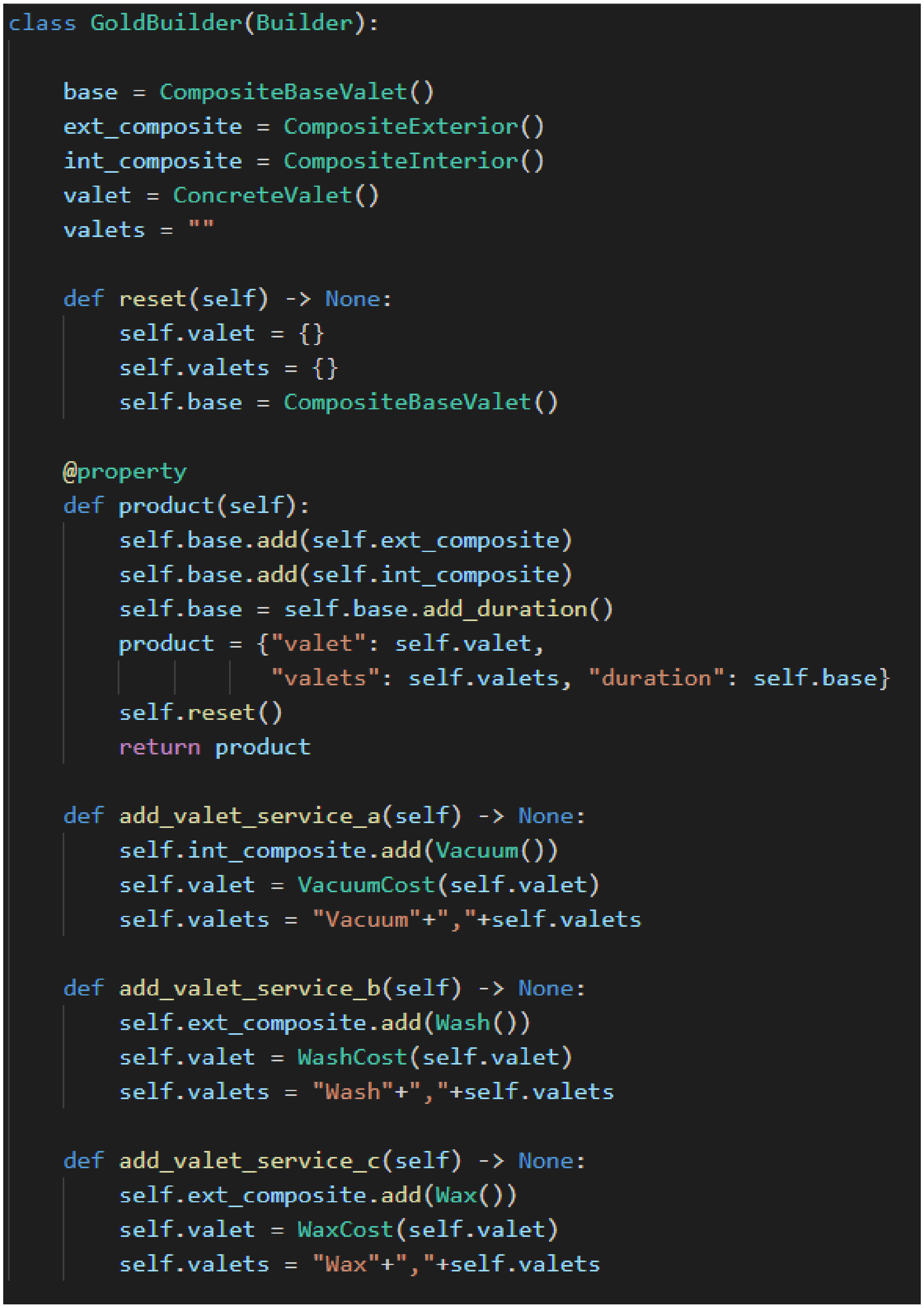


```python
class Builder(ABC):
    """
    The Builder interface specifies methods for creating the different parts of
    the Product objects.
    """

    @property
    @abstractmethod
    def product(self) -> None:
        pass

    @abstractmethod
    def add_valet_service_a(self) -> None:
        pass

    @abstractmethod
    def add_valet_service_b(self) -> None:
        pass

    @abstractmethod
    def add_valet_service_c(self) -> None:
        pass
```

Figure 24 Builder interface

First, we create an interface for the builder which every concrete builder will incorporate. We then create gold, silver, and bronze concrete builders, these represent the user membership type. The user's membership type will trigger which builder to build.

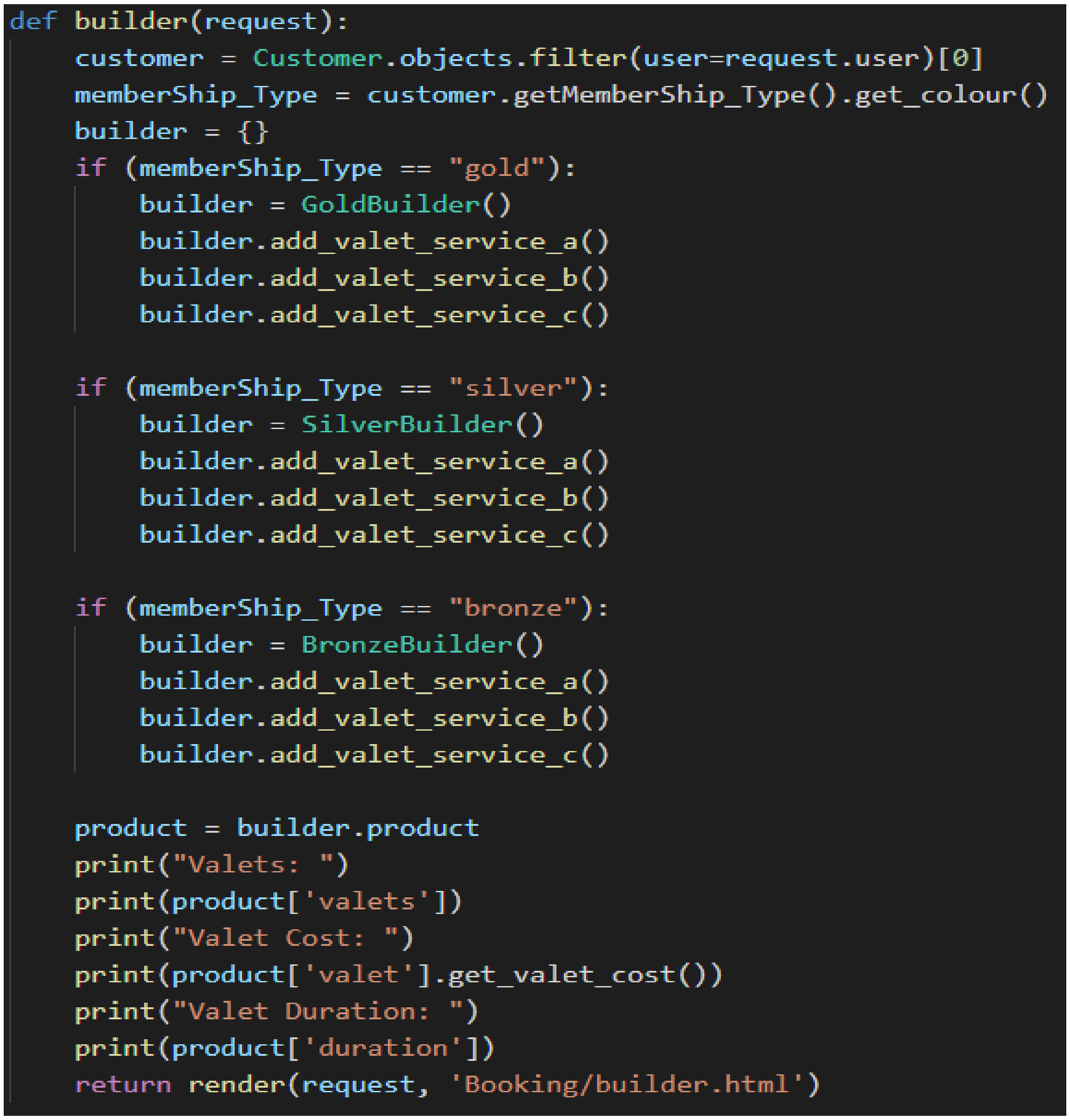
Below we can see the GoldBuilder. This builder creates a base Valet Object using the composite design pattern where it has exterior and interior composites. Aswell as a Valet decorator Object. For instance, in add_valet_service_a it adds Vacuum to the interior composite which adds time duration to the valet and Vacuum cost adds to the cost of the valet object. The product then returns the name, cost and duration of the Valet Object which is used for seasonal offers.



```python
class GoldBuilder(Builder):

    base = CompositeBaseValet()
    ext_composite = CompositeExterior()
    int_composite = CompositeInterior()
    valet = ConcreteValet()
    valets = ""

    def reset(self) -> None:
        self.valet = {}
        self.valets = {}
        self.base = CompositeBaseValet()

    @property
    def product(self):
        self.base.add(self.ext_composite)
        self.base.add(self.int_composite)
        self.base = self.base.add_duration()
        product = {"valet": self.valet,
                   "valets": self.valets, "duration": self.base}
        self.reset()
        return product

    def add_valet_service_a(self) -> None:
        self.int_composite.add(Vacuum())
        self.valet = VacuumCost(self.valet)
        self.valets = "Vacuum"+","+self.valets

    def add_valet_service_b(self) -> None:
        self.ext_composite.add(Wash())
        self.valet = WashCost(self.valet)
        self.valets = "Wash"+","+self.valets

    def add_valet_service_c(self) -> None:
        self.ext_composite.add(Wax())
        self.valet = WaxCost(self.valet)
        self.valets = "Wax"+","+self.valets
```

Figure 25 Concrete builder

Here we can see invoking the different builders depending on the user's membership type. This then prints out the valets on offer, the cost and the duration of the valet.

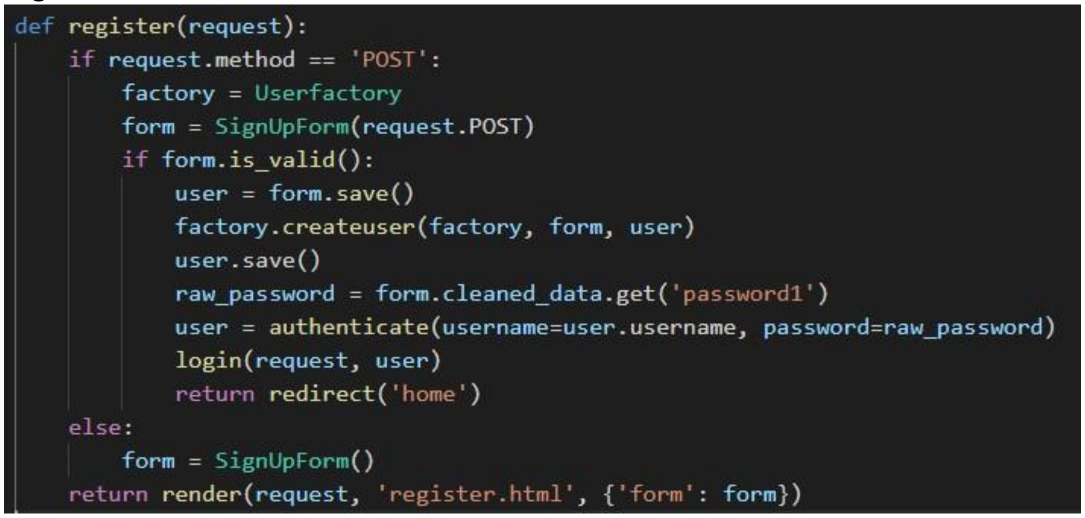


```python
def builder(request):
    customer = Customer.objects.filter(user=request.user)[0]
    memberShip_Type = customer.getMemberShip_Type().get_colour()
    builder = {}
    if (memberShip_Type == "gold"):
        builder = GoldBuilder()
        builder.add_valet_service_a()
        builder.add_valet_service_b()
        builder.add_valet_service_c()

    if (memberShip_Type == "silver"):
        builder = SilverBuilder()
        builder.add_valet_service_a()
        builder.add_valet_service_b()
        builder.add_valet_service_c()

    if (memberShip_Type == "bronze"):
        builder = BronzeBuilder()
        builder.add_valet_service_a()
        builder.add_valet_service_b()
        builder.add_valet_service_c()

    product = builder.product
    print("Valets: ")
    print(product['valets'])
    print("Valet Cost: ")
    print(product['valet'].get_valet_cost())
    print("Valet Duration: ")
    print(product['duration'])
    return render(request, 'Booking/builder.html')
```

Figure 26 Builder invocation

Factory

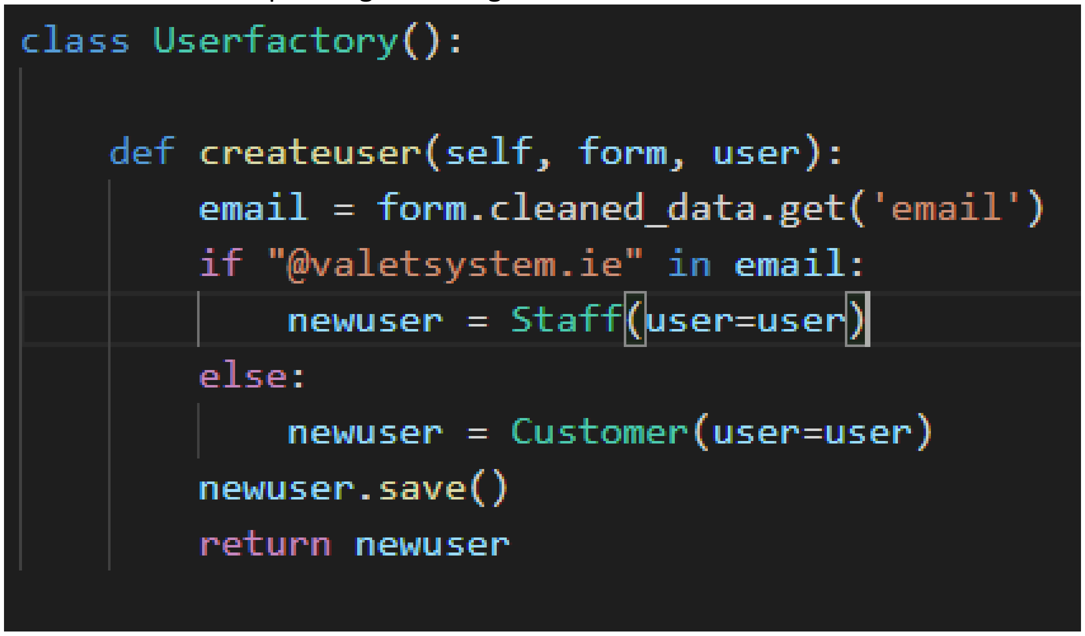
The factory method is a design pattern that provides an interface for creating objects in a superclass, it allows for subclasses to alter the type of objects that will be created.
In our project, the user is created using the signup form. This form creates a User in the database. Here a user factory design pattern was used to create the customer or staff depending on the email provided on registration.



```
def register(request):
    if request.method == 'POST':
        factory = Userfactory
        form = SignUpForm(request.POST)
        if form.is_valid():
            user = form.save()
            factory.createuser(factory, form, user)
            user.save()
            raw_password = form.cleaned_data.get('password1')
            user = authenticate(username=user.username, password=raw_password)
            login(request, user)
            return redirect('home')
    else:
        form = SignUpForm()
    return render(request, 'register.html', {'form': form})
```

Figure 27 Factory invoker

Here the user is sent to the user factory on creation. Inside the user factory, the factory method creates a customer or staff depending on data given.



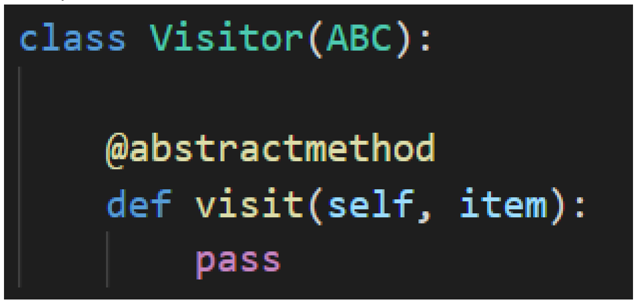
```
class Userfactory():

    def createuser(self, form, user):
        email = form.cleaned_data.get('email')
        if "@valetsystem.ie" in email:
            newuser = Staff(user=user)
        else:
            newuser = Customer(user=user)
        newuser.save()
        return newuser
```

Figure 28 Factory method

Visitor

Instead of attempting to integrate new functionality into existing classes, the Visitor pattern advises creating a new class named visitor. The original object that had to execute the action is now supplied as an argument to one of the visitor's methods, giving the method access to all the object's data. For example, in our project, we used the visitor as an ability to export data from the database to a CSV file. This enabled for extensibility and modification easily.

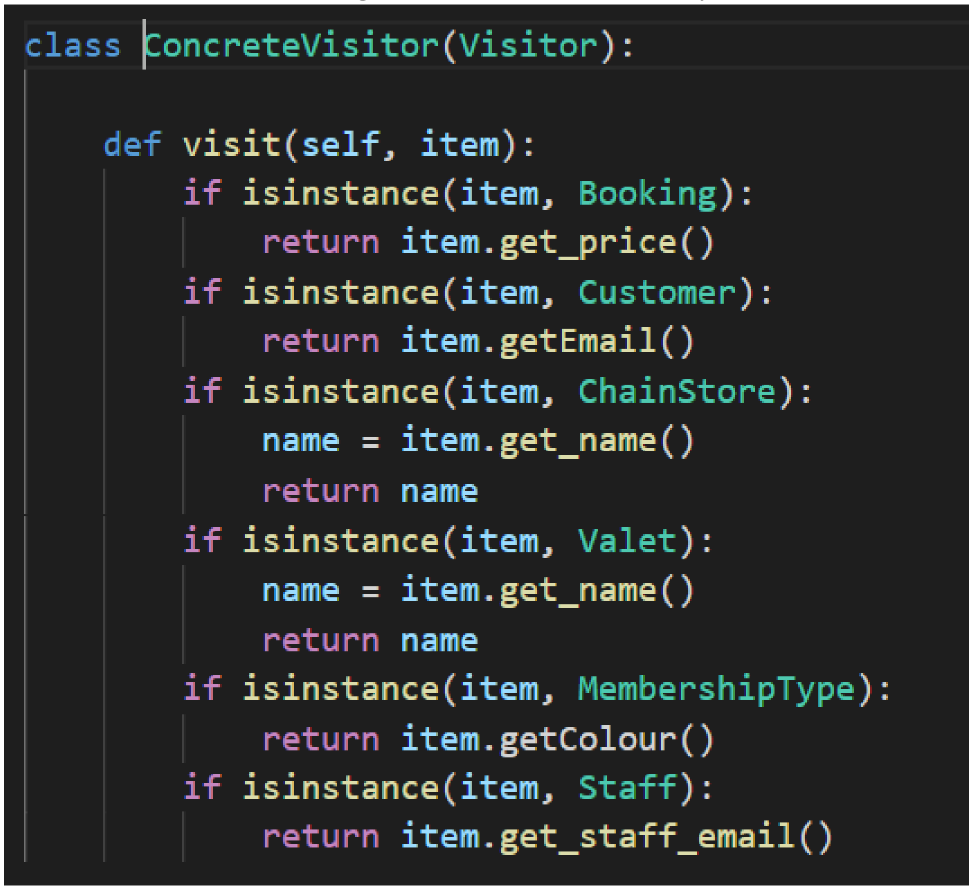
First, we declare an abstract class called the Visitor.



```python
class Visitor(ABC):

    @abstractmethod
    def visit(self, item):
        pass
```

Figure 29 Visitor interface

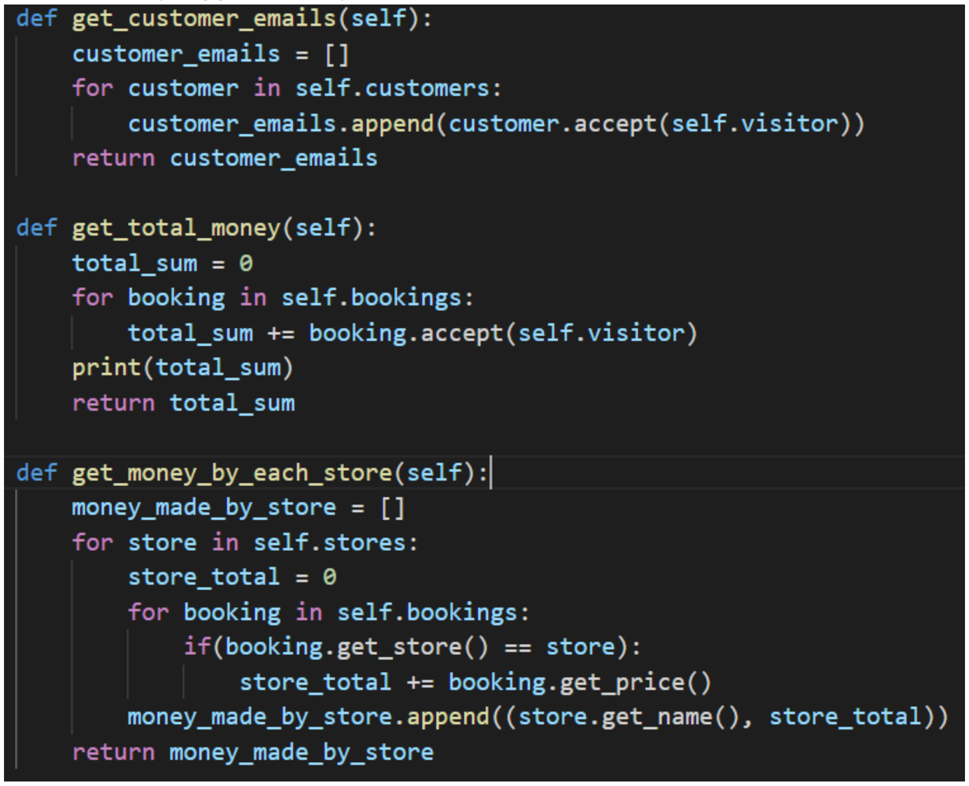
Then we create a concrete visitor which implements the visitor class. Each new instance of object must be added to the conditional logic as there is no other way to do this.



```python
class ConcreteVisitor(Visitor):

    def visit(self, item):
        if isinstance(item, Booking):
            return item.get_price()
        if isinstance(item, Customer):
            return item.getEmail()
        if isinstance(item, ChainStore):
            name = item.get_name()
            return name
        if isinstance(item, Valet):
            name = item.get_name()
            return name
        if isinstance(item, MembershipType):
            return item.getColour()
        if isinstance(item, Staff):
            return item.get_staff_email()
```

Figure 30 Concrete visitor

The visitor then is used with the pluggable adapter as was seen in above. Below is an example of the pluggable adapter. Each object accepts the visitor which in the accepts the visitor returning information back to the pluggable adapter.



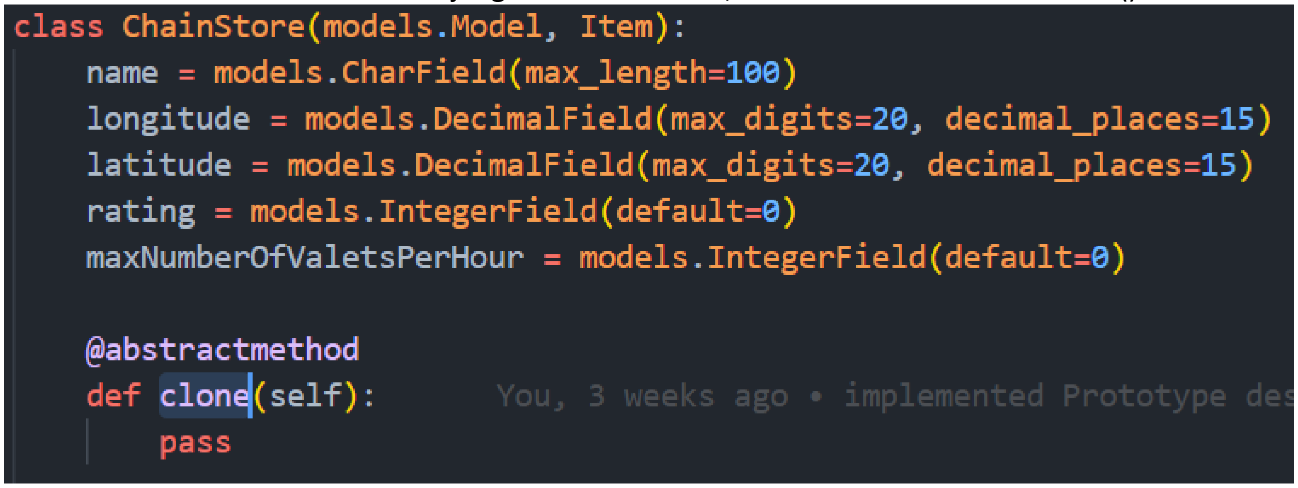
```python
def get_customer_emails(self):
    customer_emails = []
    for customer in self.customers:
        customer_emails.append(customer.accept(self.visitor))
    return customer_emails

def get_total_money(self):
    total_sum = 0
    for booking in self.bookings:
        total_sum += booking.accept(self.visitor)
    print(total_sum)
    return total_sum

def get_money_by_each_store(self):
    money_made_by_store = []
    for store in self.stores:
        store_total = 0
        for booking in self.bookings:
            if(booking.get_store() == store):
                store_total += booking.get_price()
        money_made_by_store.append((store.get_name(), store_total))
    return money_made_by_store
```

Figure 31 Pluggable adapter

Prototype

The prototype was implemented as follows.

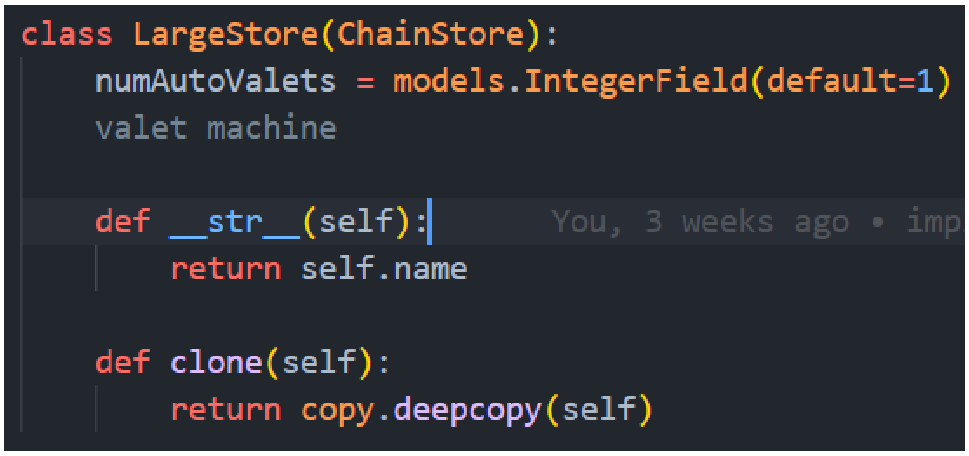
The class ChainStore defines the Django database fields, some methods and the clone() abstract method.



```python
class ChainStore(models.Model, Item):
    name = models.CharField(max_length=100)
    longitude = models.DecimalField(max_digits=20, decimal_places=15)
    latitude = models.DecimalField(max_digits=20, decimal_places=15)
    rating = models.IntegerField(default=0)
    maxNumberOfValetsPerHour = models.IntegerField(default=0)

    @abstractmethod
    def clone(self):        You, 3 weeks ago • implemented Prototype des
        pass
```

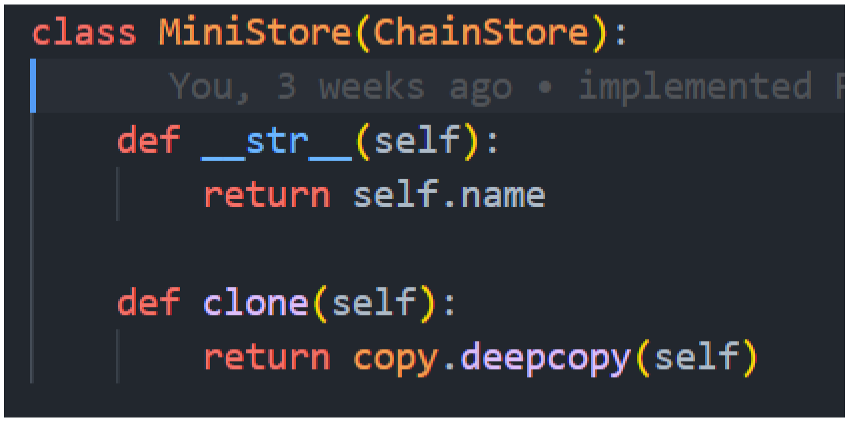
Figure 32 Prototype interface

Subclasses LargeStore and MiniStore implement this clone method. Both return a copy of themselves to the client that calls on them. LargeStore is different to MiniStore as it has at least one automatic valeting machine.



```
class LargeStore(ChainStore):
    numAutoValets = models.IntegerField(default=1)
    valet machine

    def __str__(self):        You, 3 weeks ago • imp
        return self.name

    def clone(self):
        return copy.deepcopy(self)
```

Figure 33 Concrete Prototype class



```
class MiniStore(ChainStore):
        You, 3 weeks ago • implemented F
    def __str__(self):
        return self.name

    def clone(self):
        return copy.deepcopy(self)
```

Figure 34 Concrete Prototype class

Added Values

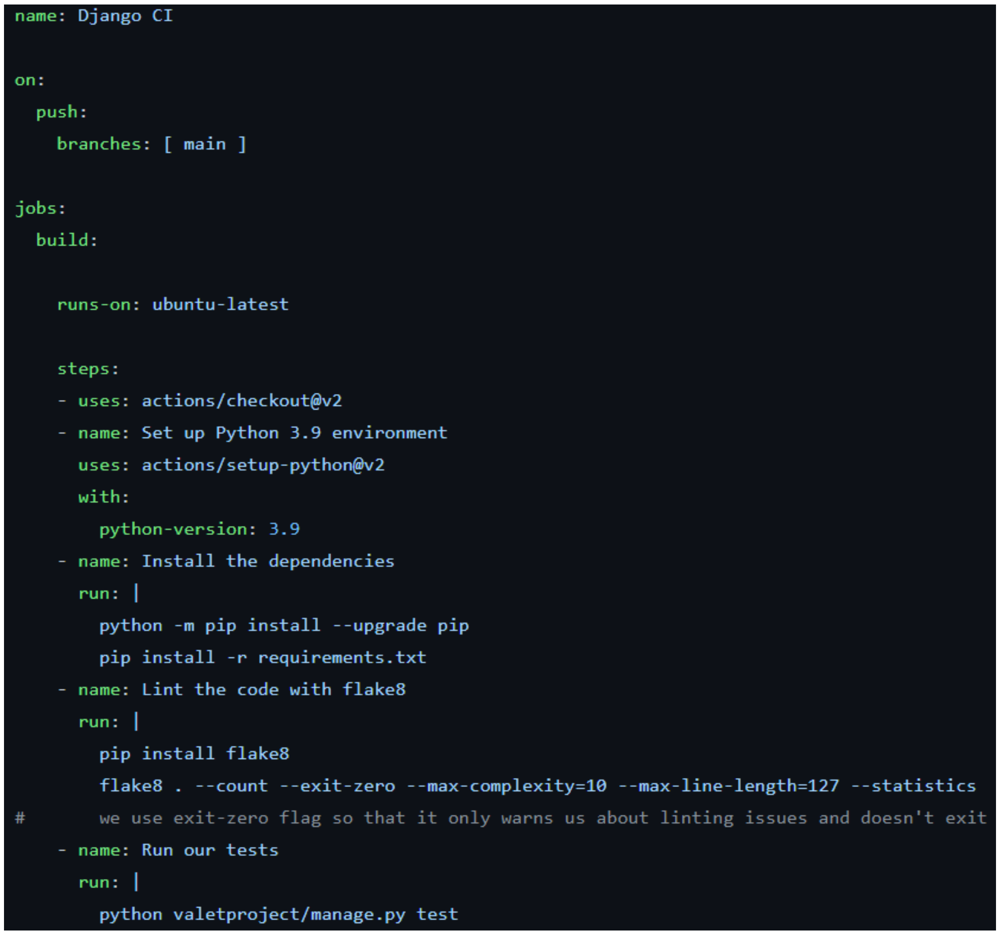
GitHub Actions

For this project, we wanted a way to automatically test the code whenever it is pushed to the main branch of GitHub. We used GitHub actions to make this happen.

Build, Lint and Test the Django Project

Below is the code we wrote in order to perform the GitHub action. We named it “Django CI”. This action runs each time we push code to the main branch. We created one job named “build” that runs on an Ubuntu image and described 4 main steps.

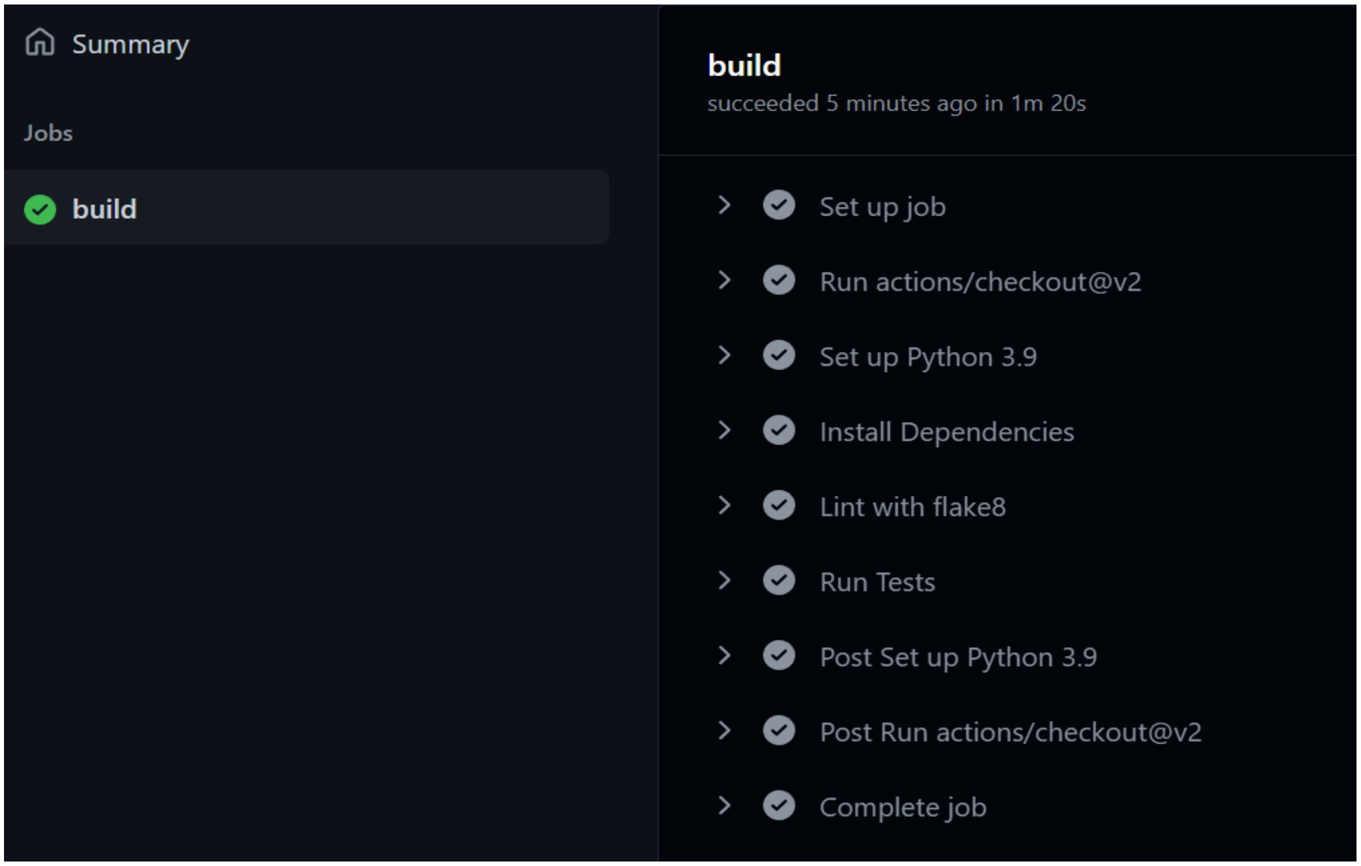
1. Set up Python 3.9 environment
2. Install the required dependencies (these are different depending on the operating system we are using i.e., Ubuntu in this case)
3. Run the linting on the code
4. Run the tests we made in the code



```yaml
name: Django CI

on:
  push:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.9 environment
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
    - name: Install the dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
    - name: Lint the code with flake8
      run: |
        pip install flake8
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
#       we use exit-zero flag so that it only warns us about linting issues and doesn't exit
    - name: Run our tests
      run: |
        python valetproject/manage.py test
```

Figure 35 Workflow django.yml

Below shows a successful run of the GitHub workflow job named “build”. This took 1 minute and 20 seconds.



Summary

Jobs

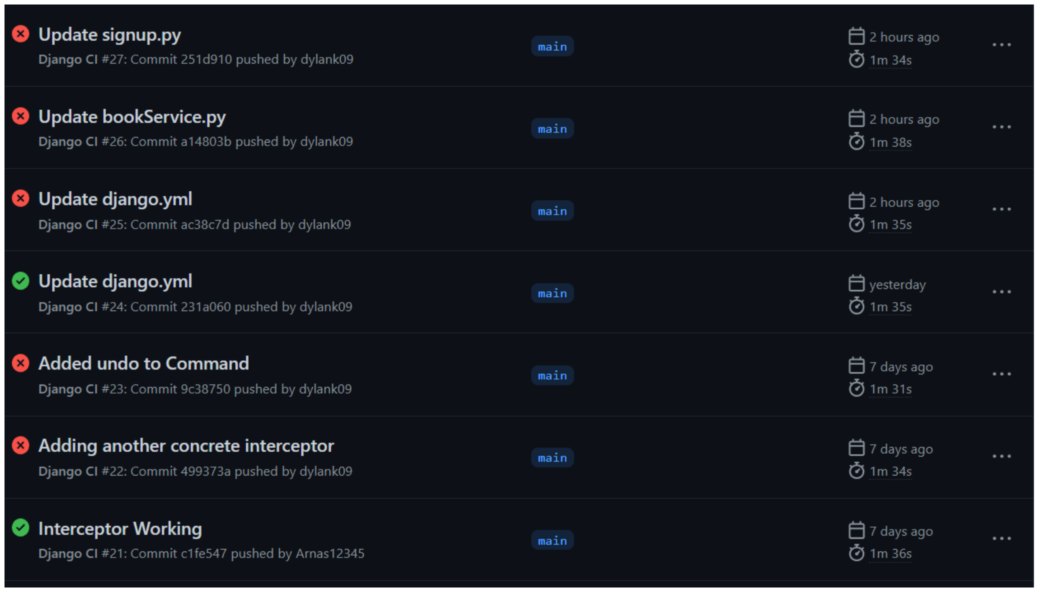
✓ build

build
succeeded 5 minutes ago in 1m 20s

- ✓ Set up job
- ✓ Run actions/checkout@v2
- ✓ Set up Python 3.9
- ✓ Install Dependencies
- ✓ Lint with flake8
- ✓ Run Tests
- ✓ Post Set up Python 3.9
- ✓ Post Run actions/checkout@v2
- ✓ Complete job

Figure 36 Example of successful job on Git Actions

Here is a sample snippet of the 34 workflow runs in the GitHub repository.

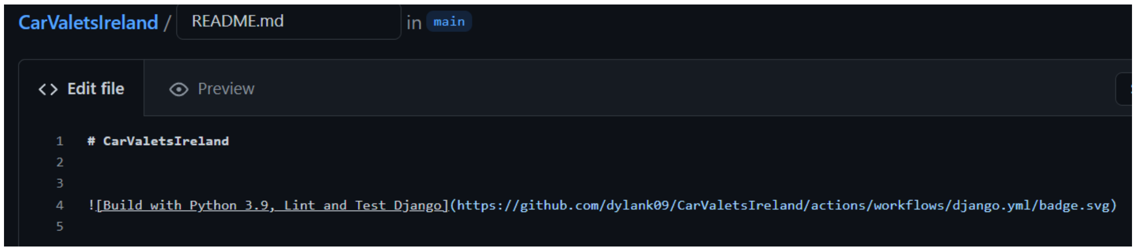


✗	Update signup.py Django CI #27: Commit 251d910 pushed by dylank09	main	2 hours ago 1m 34s
✗	Update bookService.py Django CI #26: Commit a14803b pushed by dylank09	main	2 hours ago 1m 38s
✗	Update django.yml Django CI #25: Commit ac38c7d pushed by dylank09	main	2 hours ago 1m 35s
✓	Update django.yml Django CI #24: Commit 231a060 pushed by dylank09	main	yesterday 1m 35s
✗	Added undo to Command Django CI #23: Commit 9c38750 pushed by dylank09	main	7 days ago 1m 31s
✗	Adding another concrete interceptor Django CI #22: Commit 499373a pushed by dylank09	main	7 days ago 1m 34s
✓	Interceptor Working Django CI #21: Commit c1fe547 pushed by Arnas12345	main	7 days ago 1m 36s

Figure 37 Some sample workflow runs on Github

Status of the Build

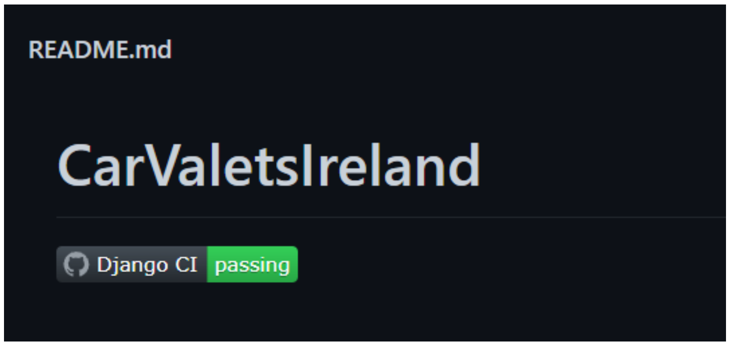
In the README.md file we can add a workflow status badge to observe the status of our GitHub action.


CarValetsIreland / README.md in main

Edit file Preview

```
# CarValetsIreland


![Build with Python 3.9, Lint and Test Django](https://github.com/dylank09/CarValetsIreland/actions/workflows/django.yml/badge.svg)

```

Figure 39 README.md file with Badge to watch status of build


README.md

CarValetsIreland

Django CI passing

Figure 38 Badge on README with status of build

Issues

We had some issues setting up the GitHub action.

Different Operating Systems

Since we were developing the project on our personal Windows PCs, the Django requirements.txt had requirements that were made for the Windows image. When attempting to Install Dependencies on the Ubuntu image, it would fail each time. We had to do some research on Google and Stack Overflow to replace the windows packages with Ubuntu compatible packages. Once all the packages were compatible, this step passed.

Linting Errors Exiting the Build

When first running the job, when we got to the Linting step it would exit if it found any issue at all (such as trailing whitespace on a line). This was not the behaviour we wanted. We simply wanted the linter to warn us of a linting issue rather than exiting the whole build process. To fix this, we added the *–exit-zero* flag seen above in the code image. This instead exits a linting issue with a zero so that it continues executing the lining.

Incorrect Python Version

At first, we tried building the Django project with python 2.7 but this was not compatible with the list of requirements we provided. It was then realised that we were running the Django project on our machines with Python 3.9. To fix this issue, we updated the django.yml code to use python 3.9 instead of 2.7.

Dockerize Django and back-up on an AWS ECR repository

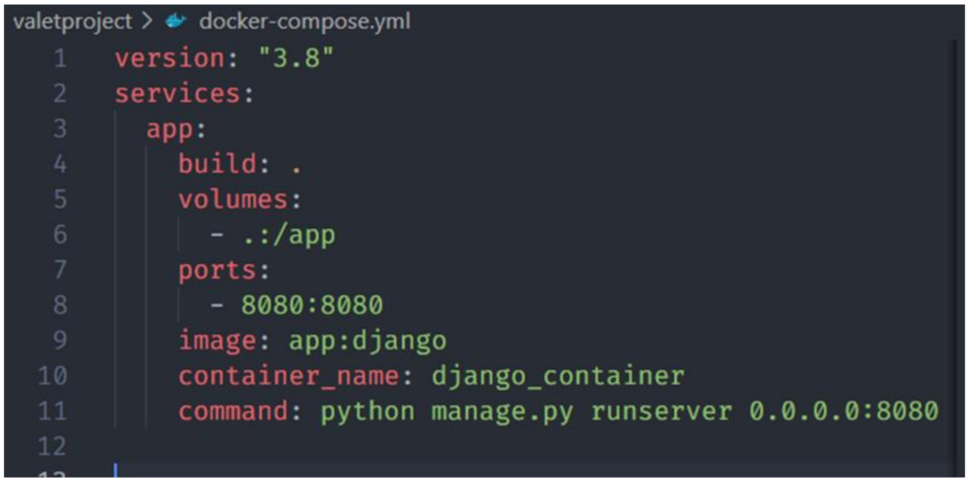
Launching Django through docker and creating an amazon ECR image for launch

Docker is an open-source container platform. The platform enables developers to package the application into containers. Once you have docker installed you can run the container on any operating system. This allows the system to be more flexible.

Containers also serve benefits over virtual machines as containers do not contain an OS. This means that containers have smaller sizes than virtual machines. They are also quicker and faster to start.

We added docker to our Django application so that it can be potentially deployed on AWS EC2 cluster machines. To carry out this process we had to create three new files in our application docker-compose.yml, Dockerfile and requirements.txt.

The docker-compose.yml:

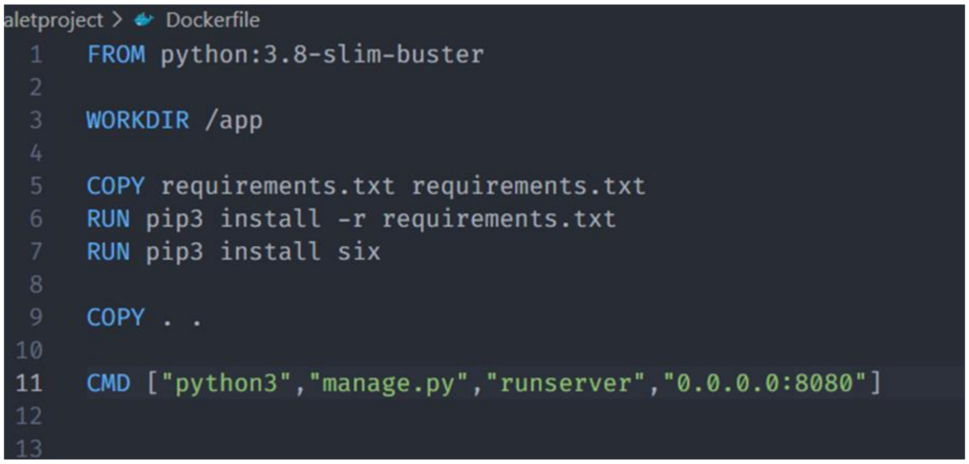


```
valetproject > docker-compose.yml
version: "3.8"
services:
  app:
    build: .
    volumes:
      - .:/app
    ports:
      - 8080:8080
    image: app:django
    container_name: django_container
    command: python manage.py runserver 0.0.0.0:8080
```

Figure 40 docker-compose file with instructions for docker

Defines and builds the docker container:

DockerFile:



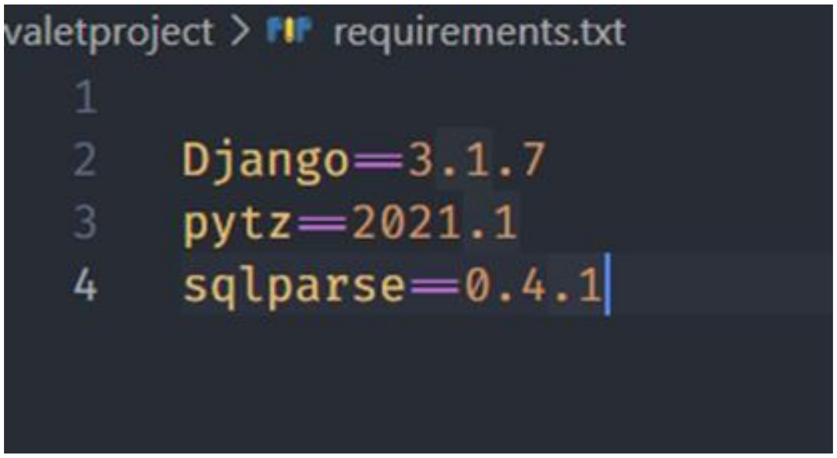
```
aletproject > Dockerfile
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt
RUN pip3 install six

COPY . .

CMD ["python3","manage.py","runserver","0.0.0.0:8080"]
```

Figure 41 Docker file

Specify the commands to run on the container:

requirements.txt:

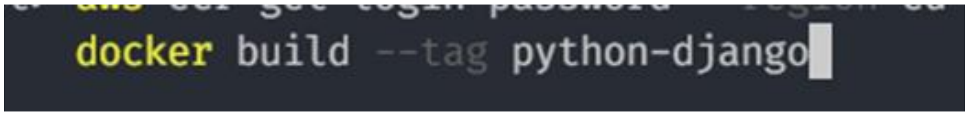
packages needed to be installed in order to run the Django application:



```
valetproject > requirements.txt

Django==3.1.7
pytz==2021.1
sqlparse==0.4.1
```

Figure 42 Docker requirements for building

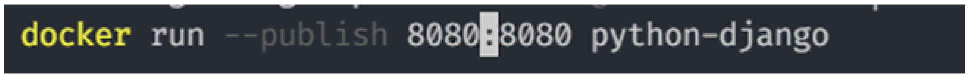
We then needed to build the docker application



```
docker build --tag python-django
```

Figure 43 Docker command to build application

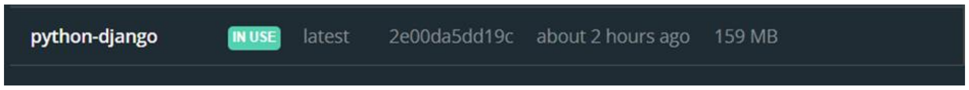
Following this we need to publish the container. The host's computer is mapped to the containers port:



```
docker run --publish 8080:8080 python-django
```

Figure 44 Docker command to run the container

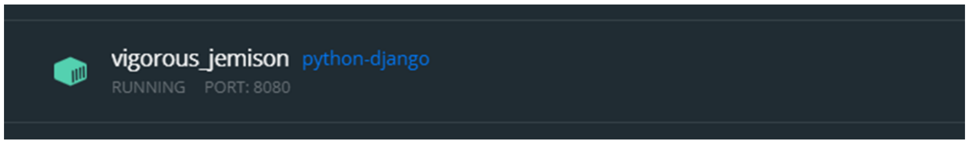
The image file created in docker desktop is as follows:



```
python-django   IN USE   latest   2e00da5dd19c   about 2 hours ago   159 MB
```

Figure 45 Docker image

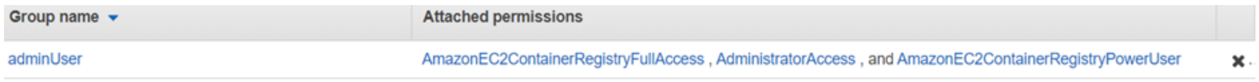
Port container running:



```
vigorous_jemison  python-django
RUNNING  PORT: 8080
```

Figure 46 Container

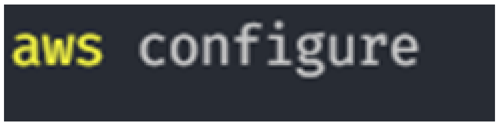
To deploy our docker image to AWS we utilized the AWS CLI. To gain access to AWS services through the CLI a super user first had to be created with permissions to edit and create containers.



Group name	Attached permissions
adminUser	AmazonEC2ContainerRegistryFullAccess , AdministratorAccess , and AmazonEC2ContainerRegistryPowerUser

Figure 47 AWS

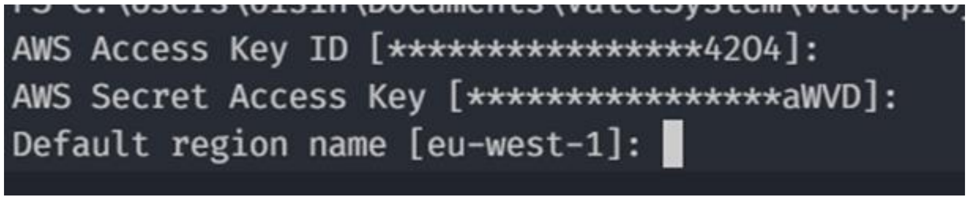
To connect the AWS user with the CLI



```
aws configure
```

Figure 48 AWS configure command

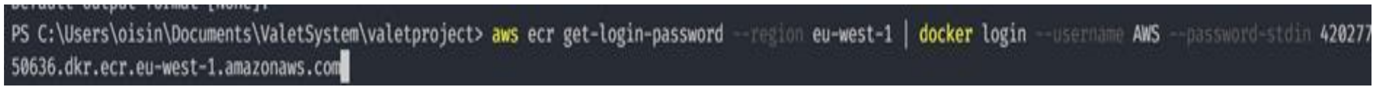
Then the users' keys generated by AWS is entered along with the region type of that user



```
AWS Access Key ID [****************4204]:
AWS Secret Access Key [****************aWVD]:
Default region name [eu-west-1]:
```

Figure 49 Inputting AWS config

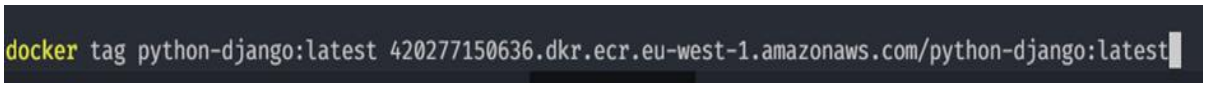
Following this we create an AWS ECR. The AWS ECR service allows you to easily manage, store, share and deploy the container images anywhere.
First, we needed to connect to the AWS services through the CLI using the following command:



```
PS C:\Users\oisin\Documents\ValetSystem\valetproject> aws ecr get-login-password --region eu-west-1 | docker login --username AWS --password-stdin 420277
50636.dkr.ecr.eu-west-1.amazonaws.com
```

Figure 50 Connecting to AWS

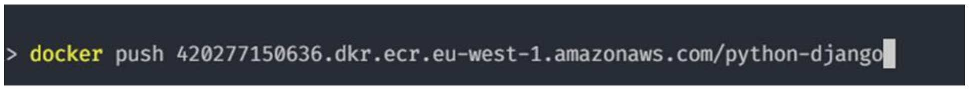
We then uploaded our docker image to the service using the following commands
We tagged our docker image using the command:



```
docker tag python-django:latest 420277150636.dkr.ecr.eu-west-1.amazonaws.com/python-django:latest
```

Figure 51 Uploading docker image to AWS

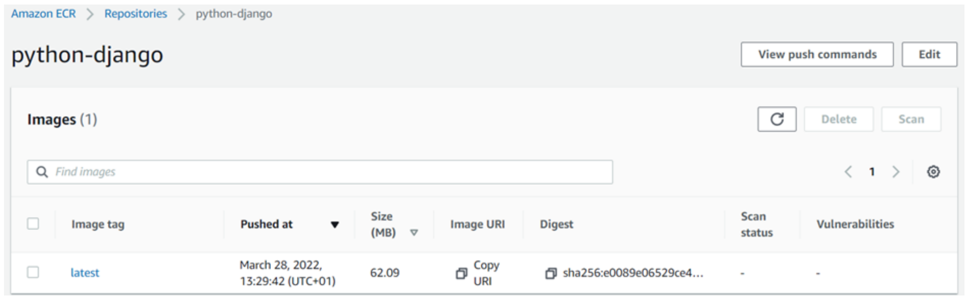
Then it was built using:



```
> docker push 420277150636.dkr.ecr.eu-west-1.amazonaws.com/python-django
```

Figure 52 Pushing image to aws

These successful uploads the image to the AWS ECR repository:



Amazon ECR > Repositories > python-django

python-django

Images (1)

Image tag	Pushed at	Size (MB)	Image URI	Digest	Scan status	Vulnerabilities
latest	March 28, 2022, 13:29:42 (UTC+01)	62.09	Copy URI	sha256:e0089e06529ce4...	-	-

Figure 53 AWS

The next steps of this process would be to set up ec2 AWS clustering machines to run the container services from the web. Due to costs we were unable to carry out this process.
However, we successfully built a docker image with a container running our project on port 8080. This will allow easy testing across all operating systems. Our image is also backed up on AWS ECR repository.

Pluggable Adapter

A wow factor for us was using the visitor design pattern in the pluggable adapter design pattern and writing the information to a CSV file. This is noted above.

Anti-Pattern

Another wow factor for us was the anti-pattern involving the bridge design pattern which is talked below in the issues section.

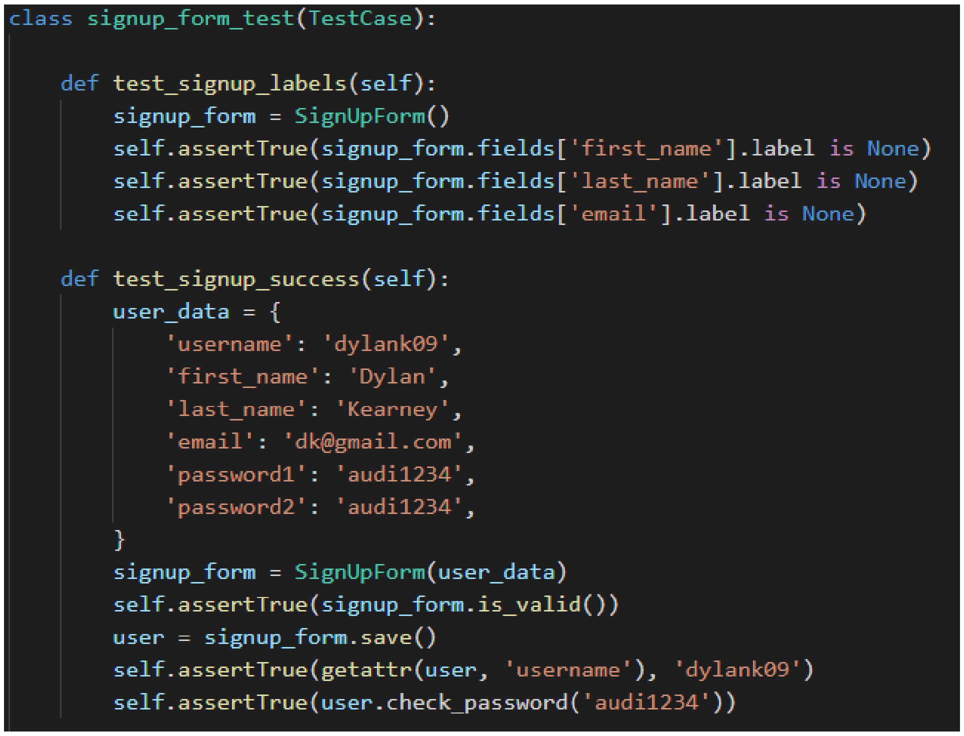
Automated Testing

Unit Tests / TDD

Test-Driven Development (TDD). TDD is a software development methodology in which test cases are created to define and validate what the code will accomplish. To put it another way, test cases for each function are generated and tested first, and if the test fails, new code is written to pass the test, resulting in code that is simple and bug-free.

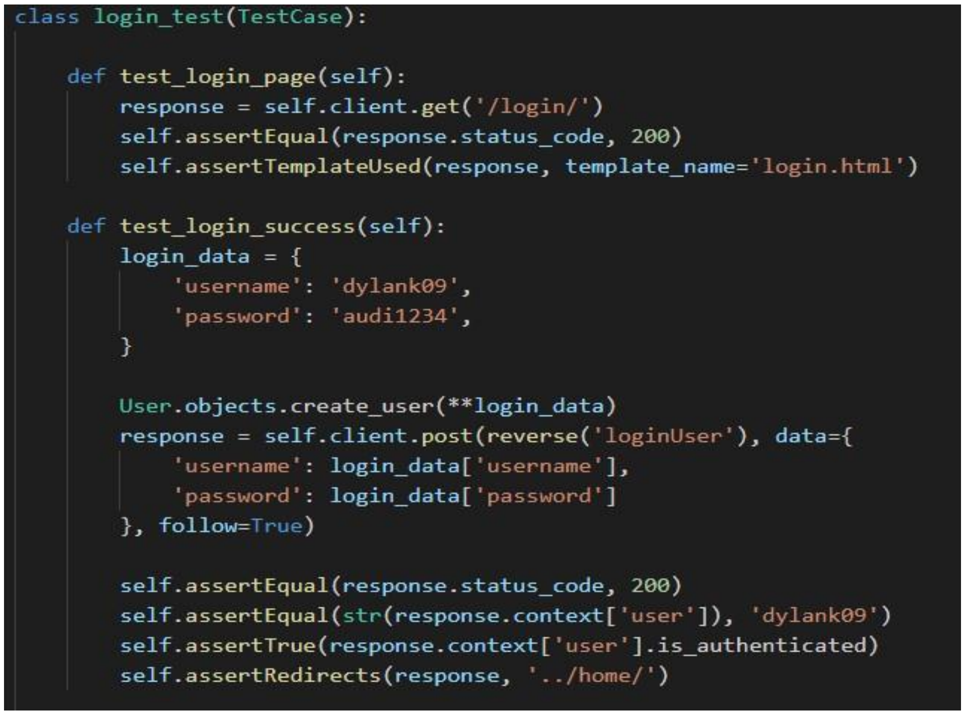
Test-Driven Designing and building tests for each tiny function of an application is the first step in development. Only write new code if an automated test fails, according to the TDD methodology. This prevents code duplication.

For our project, we tested the register, login and booking functionality with all tests passing. Below is a snippet of some of the tests.



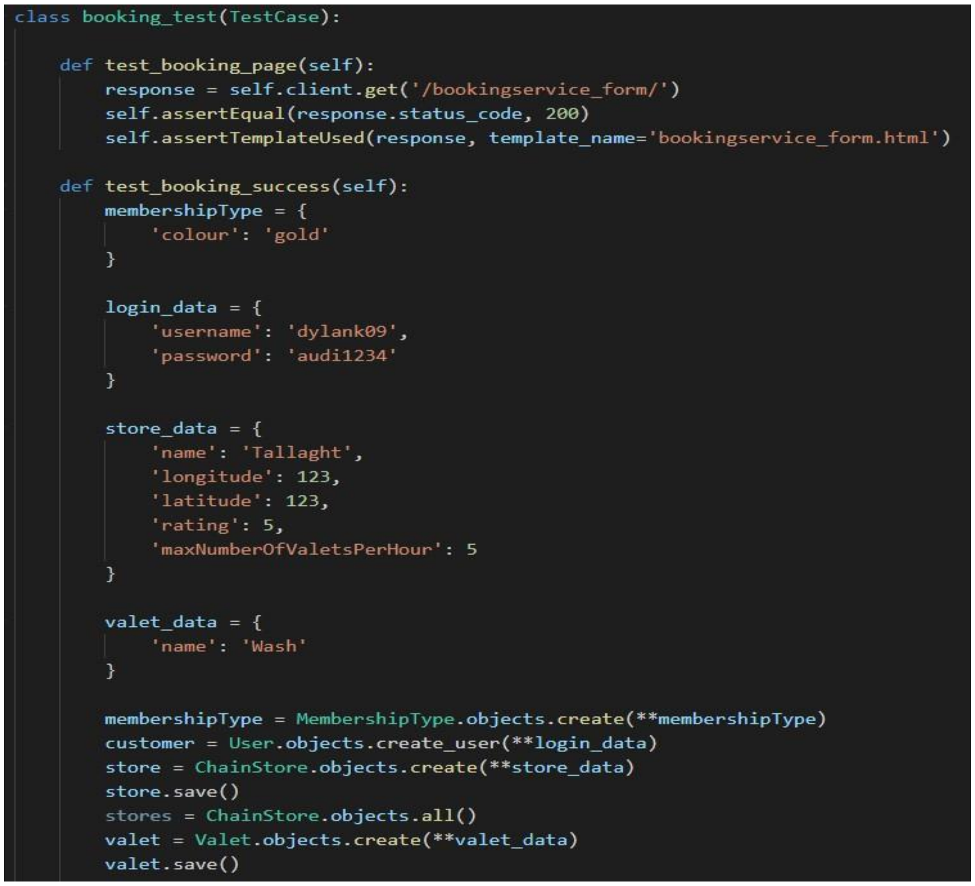
```python
class signup_form_test(TestCase):

    def test_signup_labels(self):
        signup_form = SignUpForm()
        self.assertTrue(signup_form.fields['first_name'].label is None)
        self.assertTrue(signup_form.fields['last_name'].label is None)
        self.assertTrue(signup_form.fields['email'].label is None)

    def test_signup_success(self):
        user_data = {
            'username': 'dylank09',
            'first_name': 'Dylan',
            'last_name': 'Kearney',
            'email': 'dk@gmail.com',
            'password1': 'audi1234',
            'password2': 'audi1234',
        }
        signup_form = SignUpForm(user_data)
        self.assertTrue(signup_form.is_valid())
        user = signup_form.save()
        self.assertTrue(getattr(user, 'username'), 'dylank09')
        self.assertTrue(user.check_password('audi1234'))
```

Figure 54 Testing register



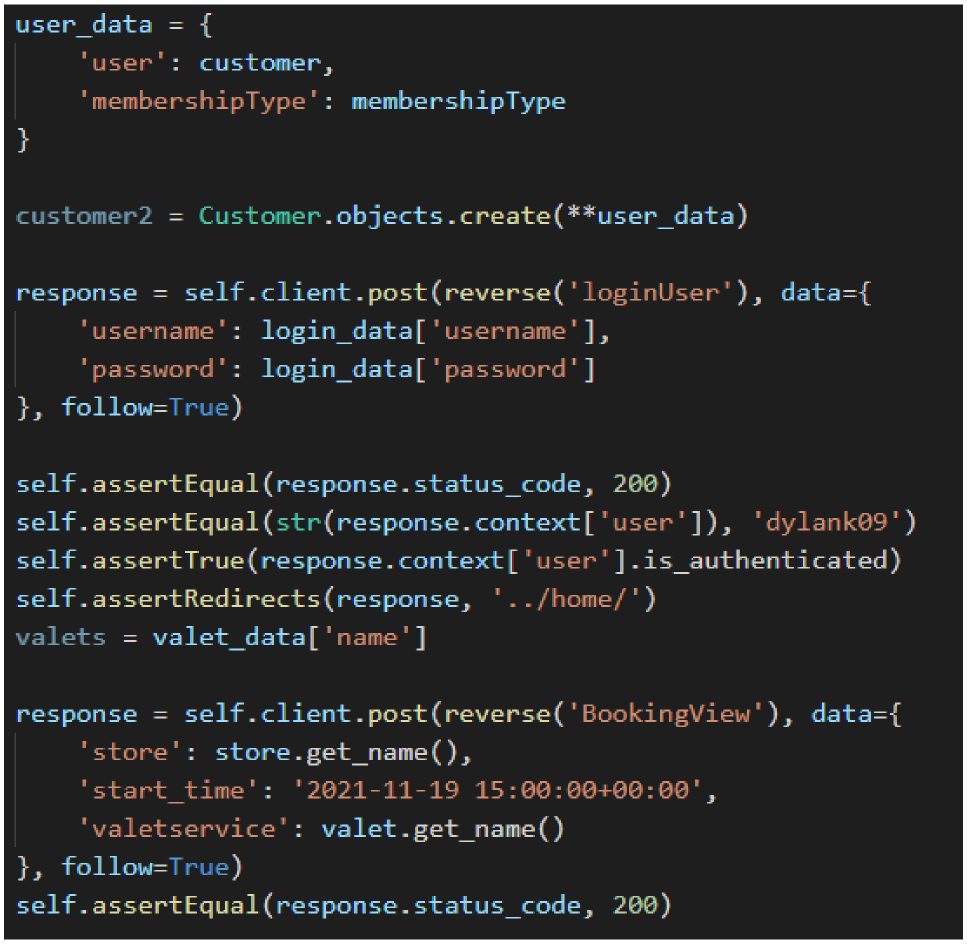
```python
class login_test(TestCase):

    def test_login_page(self):
        response = self.client.get('/login/')
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, template_name='login.html')

    def test_login_success(self):
        login_data = {
            'username': 'dylank09',
            'password': 'audi1234',
        }

        User.objects.create_user(**login_data)
        response = self.client.post(reverse('loginUser'), data={
            'username': login_data['username'],
            'password': login_data['password']
        }, follow=True)

        self.assertEqual(response.status_code, 200)
        self.assertEqual(str(response.context['user']), 'dylank09')
        self.assertTrue(response.context['user'].is_authenticated)
        self.assertRedirects(response, '../home/')
```

Figure 55 Testing login



```python
class booking_test(TestCase):

    def test_booking_page(self):
        response = self.client.get('/bookingservice_form/')
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, template_name='bookingservice_form.html')

    def test_booking_success(self):
        membershipType = {
            'colour': 'gold'
        }

        login_data = {
            'username': 'dylank09',
            'password': 'audi1234'
        }

        store_data = {
            'name': 'Tallaght',
            'longitude': 123,
            'latitude': 123,
            'rating': 5,
            'maxNumberOfValetsPerHour': 5
        }

        valet_data = {
            'name': 'Wash'
        }

        membershipType = MembershipType.objects.create(**membershipType)
        customer = User.objects.create_user(**login_data)
        store = ChainStore.objects.create(**store_data)
        store.save()
        stores = ChainStore.objects.all()
        valet = Valet.objects.create(**valet_data)
        valet.save()
```

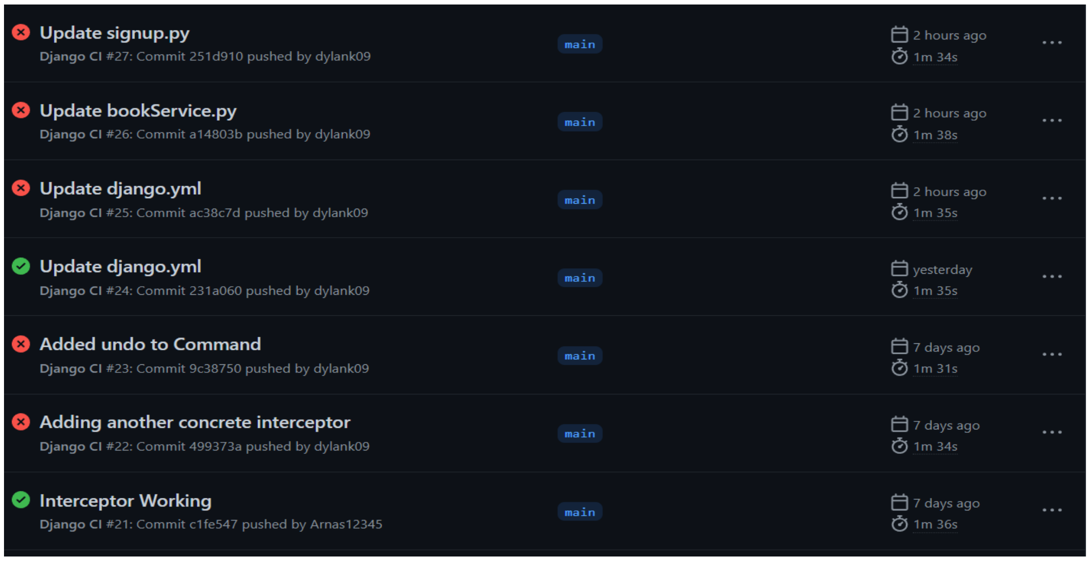
Figure 56 Testing booking



```python
user_data = {
    'user': customer,
    'membershipType': membershipType
}

customer2 = Customer.objects.create(**user_data)

response = self.client.post(reverse('loginUser'), data={
    'username': login_data['username'],
    'password': login_data['password']
}, follow=True)

self.assertEqual(response.status_code, 200)
self.assertEqual(str(response.context['user']), 'dylank09')
self.assertTrue(response.context['user'].is_authenticated)
self.assertRedirects(response, '../home/')
valets = valet_data['name']

response = self.client.post(reverse('BookingView'), data={
    'store': store.get_name(),
    'start_time': '2021-11-19 15:00:00+00:00',
    'valetservice': valet.get_name()
}, follow=True)
self.assertEqual(response.status_code, 200)
```

Figure 57 Testing booking

All the tests are run using the GitHub action described above in the Added Value section. If any test fails, it will fail the workflow. This allowed us to do Automated Testing.



```
Update signup.py                                          main    2 hours ago
Django CI #27: Commit 251d910 pushed by dylank09                  1m 34s

Update bookService.py                                     main    2 hours ago
Django CI #26: Commit a14803b pushed by dylank09                  1m 38s

Update django.yml                                         main    2 hours ago
Django CI #25: Commit ac38c7d pushed by dylank09                  1m 35s

Update django.yml                                         main    yesterday
Django CI #24: Commit 231a060 pushed by dylank09                  1m 35s

Added undo to Command                                     main    7 days ago
Django CI #23: Commit 9c38750 pushed by dylank09                  1m 31s

Adding another concrete interceptor                       main    7 days ago
Django CI #22: Commit 499373a pushed by dylank09                  1m 34s

Interceptor Working                                       main    7 days ago
Django CI #21: Commit c1fe547 pushed by Arnas12345                1m 36s
```

Figure 58 Sample workflow runs on GitHub

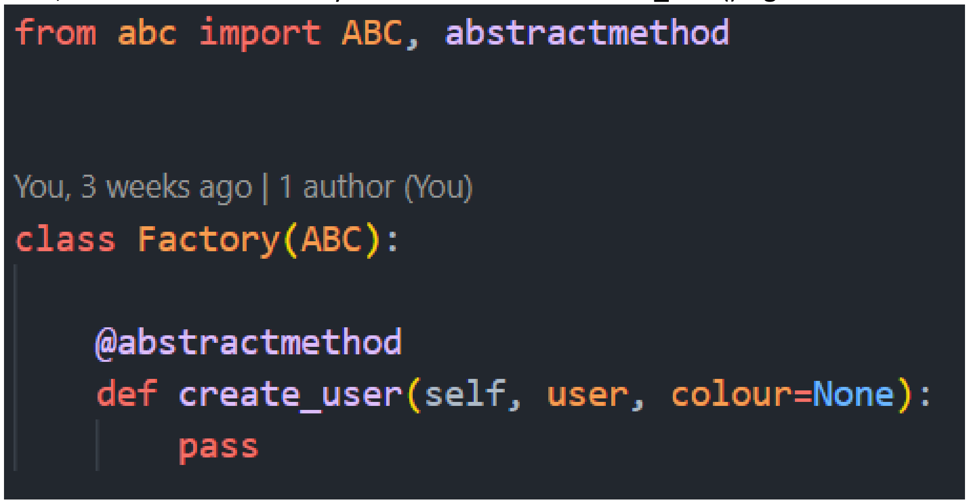
Issues

Bridge Anti-Pattern

During development we attempted to implement the Bridge Design Pattern.

There already was a UserFactory that would add either a Staff or Customer to the database based on if they had a staff email address or not. We wanted to instead make this into an Abstract Factory with a class for each StaffFactory and CustomerFactory.

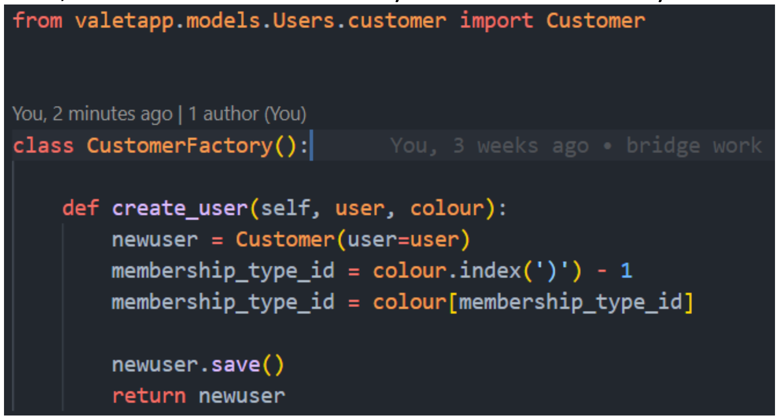
First, we created the Factory abstract class with create_user() signature.



```
from abc import ABC, abstractmethod


You, 3 weeks ago | 1 author (You)
class Factory(ABC):

    @abstractmethod
    def create_user(self, user, colour=None):
        pass
```

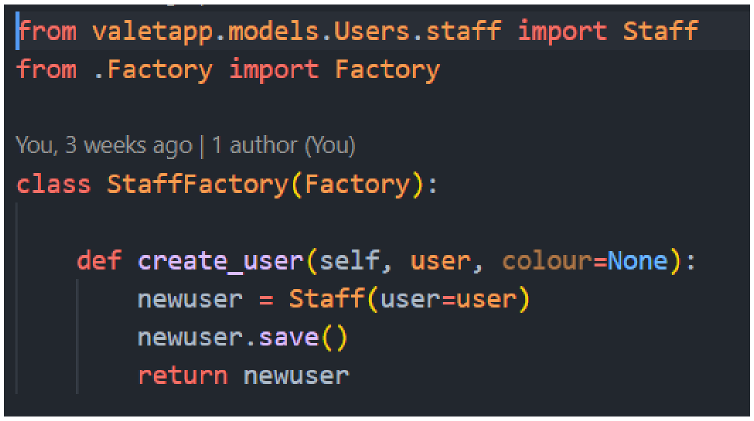
Figure 59 Factory/ Implementation interface

Next, we created the StaffFactory and CustomerFactory classes that implemented the create_user() method



```
from valetapp.models.Users.customer import Customer


You, 2 minutes ago | 1 author (You)
class CustomerFactory():        You, 3 weeks ago • bridge work

    def create_user(self, user, colour):
        newuser = Customer(user=user)
        membership_type_id = colour.index(')') - 1
        membership_type_id = colour[membership_type_id]

        newuser.save()
        return newuser
```

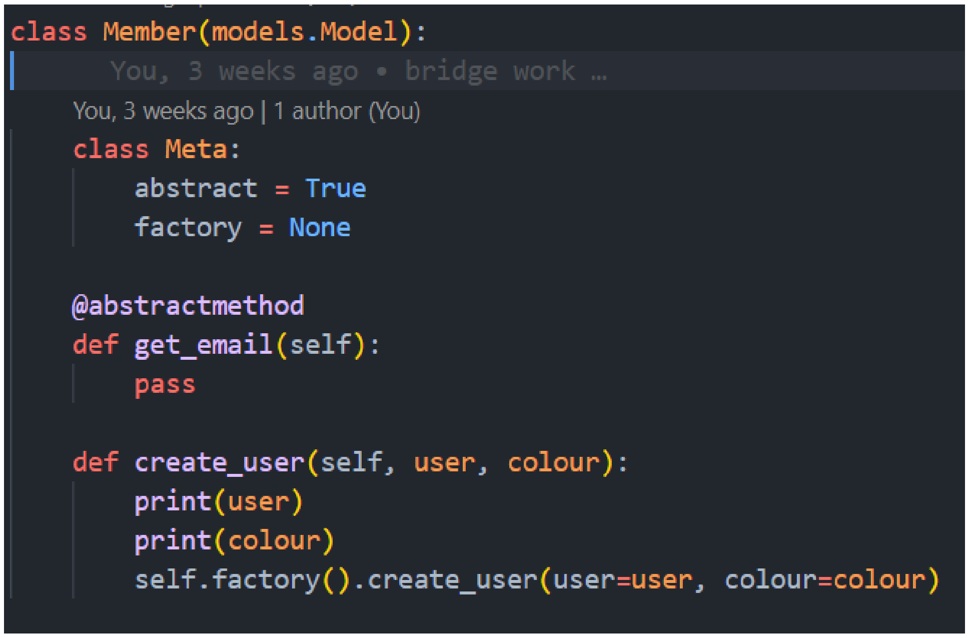
Figure 60 Concrete Implementation for Customer



```
from valetapp.models.Users.staff import Staff
from .Factory import Factory

You, 3 weeks ago | 1 author (You)
class StaffFactory(Factory):

    def create_user(self, user, colour=None):
        newuser = Staff(user=user)
        newuser.save()
        return newuser
```

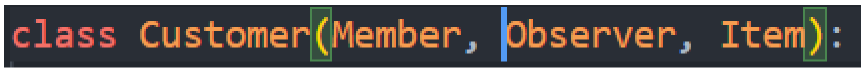
Figure 61 Concrete Implementation for Staff

Now that we had the implementation side of the Bridge design pattern created, we moved on to the abstraction. We created the Member class as an abstraction for Customer and Staff classes. The Member class provides the create_user() method to its subclasses. The concrete abstraction classes are constructed by supplying the relevant factory class (i.e., CustomerFactory or StaffFactory).



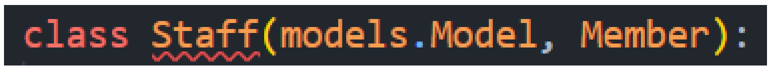
```
class Member(models.Model):
        You, 3 weeks ago • bridge work …
    You, 3 weeks ago | 1 author (You)
    class Meta:
        abstract = True
        factory = None

    @abstractmethod
    def get_email(self):
        pass

    def create_user(self, user, colour):
        print(user)
        print(colour)
        self.factory().create_user(user=user, colour=colour)
```

Figure 62 Abstraction Interface



```
class Customer(Member, Observer, Item):
```

Figure 63 Concrete abstraction

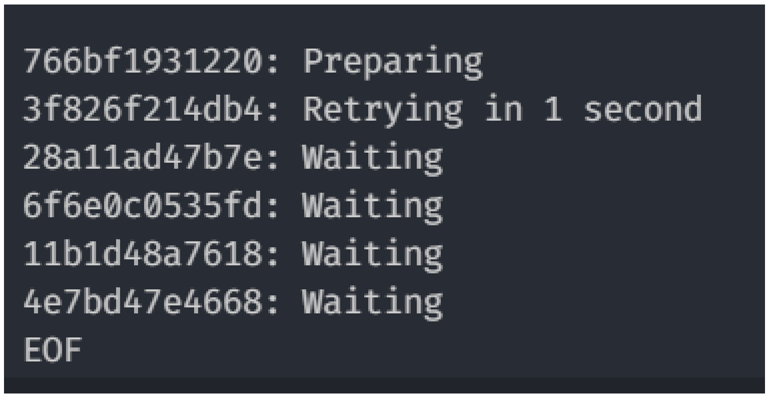


```
class Staff(models.Model, Member):
```

Figure 64 Concrete abstraction

AWS connection issue

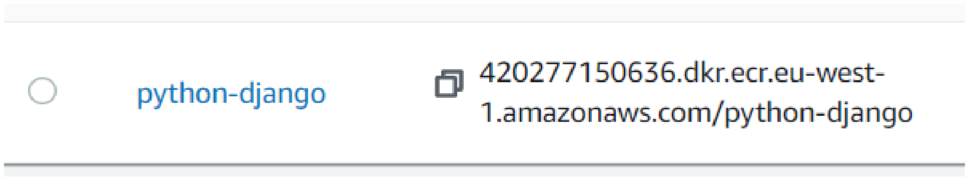
When trying to upload our docker image to the AWS ECR service we encountered an issue with pushing our image to the repository.
During the process we got the following error when trying to push the image:



```
766bf1931220: Preparing
3f826f214db4: Retrying in 1 second
28a11ad47b7e: Waiting
6f6e0c0535fd: Waiting
11b1d48a7618: Waiting
4e7bd47e4668: Waiting
EOF
```

Figure 65 Error uploading to AWS

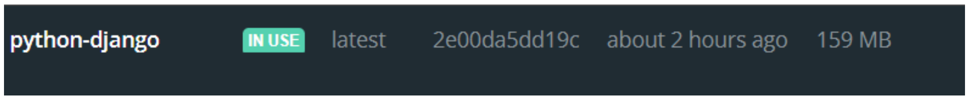
We identified that the source of the problem was due to a mismatch in naming between our docker image and the one that was created on AWS.



python-django 420277150636.dkr.ecr.eu-west-1.amazonaws.com/python-django

Figure 66 AWS image

The repository name above needed to match the container name within docker.



python-django IN USE latest 2e00da5dd19c about 2 hours ago 159 MB

Figure 67 A docker image

Code Issues and Team Dynamics

For this semesters project, we did not run into problems relating to the code. Aswell, we had great team dynamics.

Evaluation

Support for relevant NFRs

Security

The design and architecture patterns that we selected add to the NFR of security in our system. The pluggable adapter allows 3rd party services to use our system without needing to know the code base only the pluggable adapter. This adds a layer of security between the 3rd party services and our system. Furthermore, the interceptor adds another layer of security. The application can query the concrete framework via the context object instead of talking directly to the concrete framework. This adds a major layer of security to our system as well as a separation of concern.

Extensibility

Using design patterns, it made our system much more open to extensibility. The use of the visitor design pattern made it easy to add extra functionality with minimal changes in the concrete visitor conditional logic making the system more extensible. The use of the factory method also added to the extensibility of the system making it easier to add new types of users. The builder design pattern also made it easy to create new seasonal deals for customers making the project more extensible.

Language and Framework Selection

We chose python for our language of choice. We chose this language because as a group we wanted to become better python programmers as we have never tackled a project of this scale with this language. In addition, we felt learning python would be a beneficial skill to add to our arsenal as it has become one of the most used languages in the industry in recent years.

Another benefit of using the python language is its syntax simplicity, so it would be an easy language for all group members to learn. Due to the scale of this project, reducing the amount of code and complexity of trying to achieve certain concepts was necessary from a time perspective. From our final project, you can see that the lines of code are significantly lower than what this project would be in java.

A challenging but essential choice we also had to make was framework selection. Although we struggled initially with coming to a final decision on this aspect, we settled on the Django framework. Django is a large framework, and as none of the teams were familiar with MVC's frameworks, the learning curve was tricky. However, once we managed to overcome the basics of developing an application in Django, we began to see the framework's many advantages. Django is a loosely coupled framework, this allows the component in Django to act as separate entities rather than being dependent on each other. This was particularly useful for separating the business logic from the rest of the application, providing extensibility. However, due to how Django abstracts its code for components, the developer is constrained to coding in a certain way. It took some time to get the hang of this within the framework also.

Module Critique

Overall, we enjoyed this module more than CS4125. We felt that since this module focused more on coding than theory it was easier to understand the concepts discussed in lectures. Since the project is due before week twelve, it gives us more time to study for exams.

Version Control

Git and GitHub

We used GitHub for version control. We each used the Git CLI to commit, push, pull and perform otheroperations on the Git repository. GitHub allows for continuous integration which satisfied the Agile methodology.

GitHub

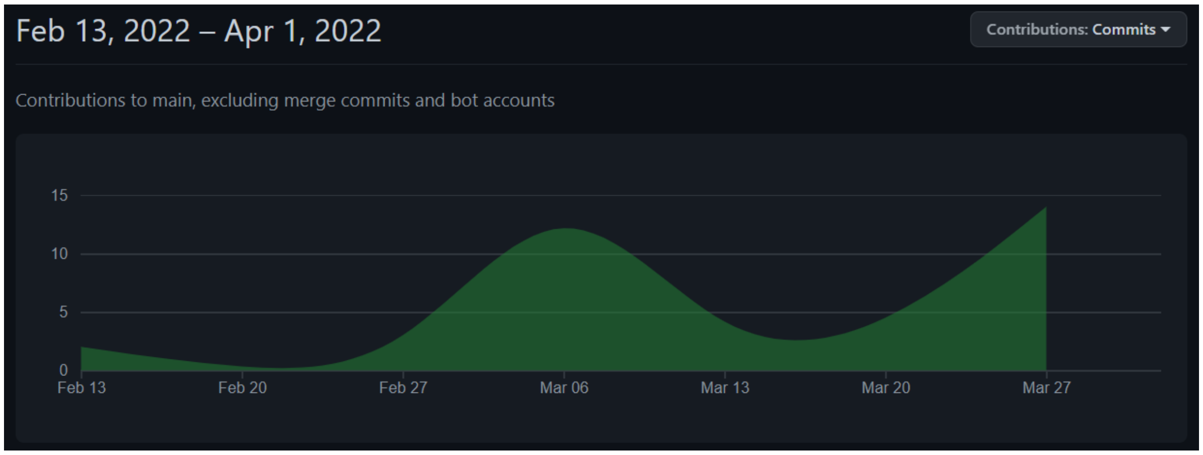


Figure 68 GitHub commits over development period

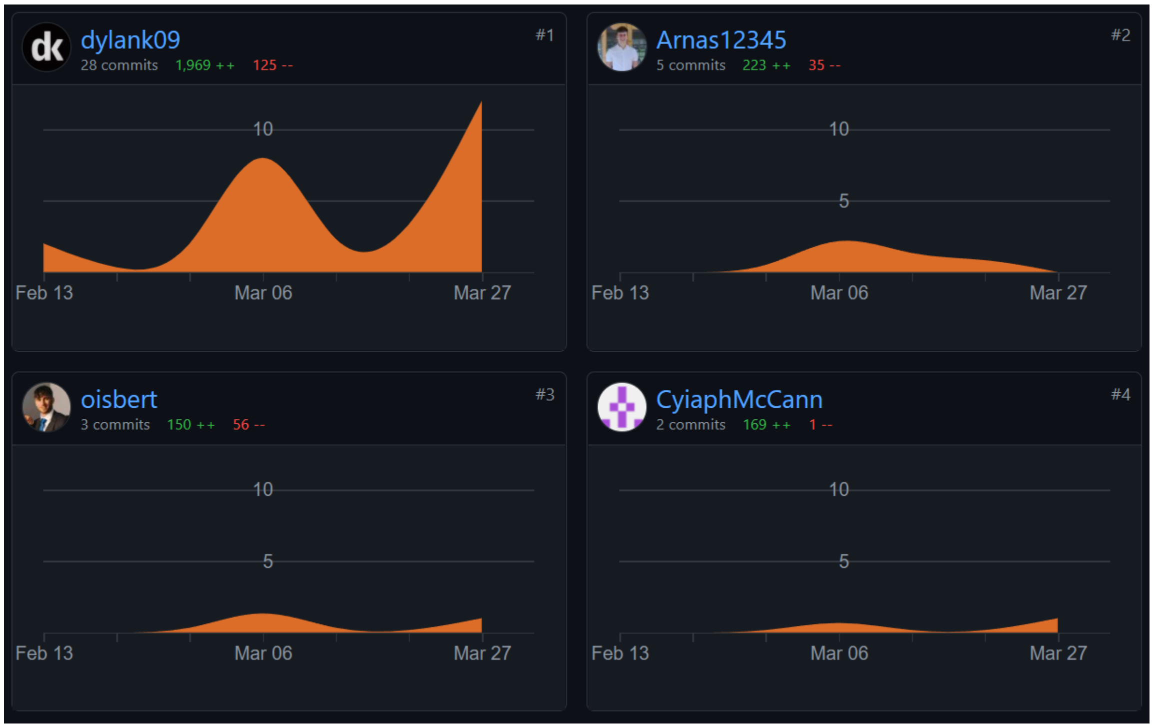


Figure 69 Individual commits

References

Dr.-Ing. Michael Eichberg, 2022. The Interceptor Architectural Pattern [online] Available at: <https://stg-tud.github.io/ctbd/2016/CTBD_10_components.pdf [Accessed 17 November 2021]. [Accessed 28 March 2022].

GeeksforGeeks. 2021. *Composite Design Pattern - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/composite-design-pattern/> [Accessed 28 March 2022].

Refactoring Guru, 2022. Adapter [online] Available at: <https://refactoring.guru/design-patterns/command> [Accessed 28 March 2022].

Refactoring Guru, 2022. Builder[online] Available at: <https://refactoring.guru/design-patterns/builder> [Accessed 28 March 2022].

Refactoring Guru, 2022. Command [online] Available at: <https://refactoring.guru/design-patterns/command> [Accessed 28 March 2022].

Refactoring Guru, 2022. Factory[online] Available at: <https://refactoring.guru/design-patterns/factory> [Accessed 28 March 2022].

Refactoring Guru, 2022. Prototype [online] Available at: <https://refactoring.guru/design-patterns/prototype> [Accessed 28 March 2022].

Refactoring Guru, 2022. Visitor[online] Available at: <https://refactoring.guru/design-patterns/visitor> [Accessed 28 March 2022].

Contribution

Source Code

Package	Design Pattern	Lines of Code	Authors
Views.Command	Command	83	Arnas, Dylan
Views.Visitor	Visitor	38	Arnas, Cyiaph
Views.Visitor	Pluggable Adapter	104	Oisin, Arnas
Views.Builder	Builder	128	Cyiaph
Views.Booking	Interceptor	259	Arnas, Dylan, Oisin
Models.Store	Prototype	60	Dylan
Models.Valet	Composite	107	Arnas, Cyiaph
Forms.Registration	Factory	23	Arnas
ContextObject	Interceptor	12	Dylan, Arnas, Oisin
Dispatcher	Interceptor	40	Dylan, Arnas, Oisin
Framework	Interceptor	28	Dylan, Arnas, Oisin
Interceptor	Interceptor	39	Dylan, Arnas, Oisin
Docker		22	Oisin
Bridge Anti-Pattern (On GitHub Branch)	Bridge	54	Dylan
GitActions		27	Dylan
	Total Lines:	989	
	Total Lines Arnas:	733	
	Total Lines Dylan:	602	
	Total Lines Oisin:	504	
	Total Lines Cyiaph:	273	

Report

Name	Contribution
Dylan Kearney	Requirements, Patterns, Code, WOW Factor, Testing, Issues, Evaluation
Oisin McNamara	Requirements, Patterns, Code, WOW Factor, Issues
Arnas Juravicius	Requirements, Patterns, Architecture, Structural, Code, Evaluation
Cyiaph McCann	Requirements, Patterns, Code

PENALTIES					
	Description	**TM1**	**TM2**	**TM3**	**TM4**
1	Late Submission				
2	Failure to contribute to coding effort				
3	Failure to contribute to writing of report				
4	Failure to report problems with team dynamics				
5	Failure to contribute to demo week 13				
	Sub-total (B)				

FINAL MARKS AWARDED				
	Student1	**Student2**	**Student3**	**Student4**
(A-B)				