Evidence Gathering Document for SQA Level 8 Professional Developer Award.

This document is designed for you to present your screenshots and diagrams relevant to the PDA and to also give a short description of what you are showing to clarify understanding for the assessor.

Fill in each point with screenshot or diagram and description of what you are showing.

Each point requires details that cover each element of the Assessment Criteria, along with a brief description of the kind of things you should be showing.

Week 1

Unit	Ref	Evidence
I&T	I.T.6	Demonstrate the use of a hash in a program. Take screenshots of: *A hash in a program *A function that uses the hash *The result of the function running

```ruby
@pet_shop = {
  name: "Edinburgh Petstore",
  pets: [
    {
      name: "Jimmy",
      pet_type: "dog",
      breed: "Whippet",
      price: "1000"
    },
    {
      name: "Ricky",
      pet_type: "dog",
      breed: "Cocker Spaniel",
      price: "1500"
    },
  ],
  admin: {
    total_cash: 10000,
    pets_sold: 17
  }
}
```

The screenshot above shows a hash in a Ruby program. The hash is assigned to the local variable *@pet_shop*, which contains the name of a pet shop, stock, total cash and pets sold. The stock is represented by an array within the hash, which itself contains two further hashes representing pets in stock. The total cash and pets sold are contained within a separate *admin* hash. In this hash the keys are represented by Symbols, and the values are either Strings, or Integers in the case of the *admin* hash.

```ruby
def find_pet_by_name(pet_shop_hash, pet_name)
  for pet in pet_shop_hash[:pets]
    if pet_name == pet[:name]
      return pet
    end
  end
  return nil
end

p find_pet_by_name(@pet_shop, "Jimmy")
```

The screenshot above shows a function to find a pet within the *@pet_shop* hash. It has two parameters; the first (*pet_shop_hash*) takes the hash to search and the second (*pet_name*) takes the *name* key of the pet to search for. The function contains a *for* loop which loops through every item in the pets array within the pet shop, and runs an *if* statement checking if the *:name* key of the *pet* in the array matches the name passed in. If there is a match, it returns the hash of the *pet* matched. If there are no matches after the loop has finished, the function returns *nil*. The last line prints the result of running the function with the arguments *@pet_shop* and *"Jimmy"* into the terminal.

```
→  week_01 ruby pet_shop_PDA.rb
{:name=>"Jimmy", :pet_type=>"dog", :breed=>"Whippet", :price=>"1000"}
```

Above is the result of running that function on the *@pet_shop* hash. It prints the hash that matches the *:name* key *"Jimmy"* into the terminal.

Week 2

Unit	Ref	Evidence
I&T	I.T.5	Demonstrate the use of an array in a program. Take screenshots of: *An array in a program *A function that uses the array *The result of the function running

```ruby
people = [
  {name: 'Ed', age: 10, cash: 5},
  {name: 'Ali', age: 12, cash: 10},
  {name: 'Gino', age: 8, cash: 2}
]
```

The screenshot above shows an array in a Ruby program that contains three objects, each with keys *:name*, *:age* and *:cash* represented by symbols.

```ruby
def get_total_cash(array)

  total_cash = 0

  for person in array
    total_cash += person[:cash]
  end

  return total_cash

end
```

The function above takes an array as its parameters, and loops through the array. For each object in the array it adds the value of the *:cash* key to the *total_cash* variable. It then returns the *total_cash* variable.

```
20  p get_total_cash(people)
21

→  week_02 ruby loops_in_functions.rb
17
```

Above shows the result of running a program which calls the function with the *people* array as its argument, and prints the result. The result *17* is printed in the terminal.

Week 3

Unit	Ref	Evidence
I&T	I.T.3	Demonstrate searching data in a program. Take screenshots of: *Function that searches data *The result of the function running

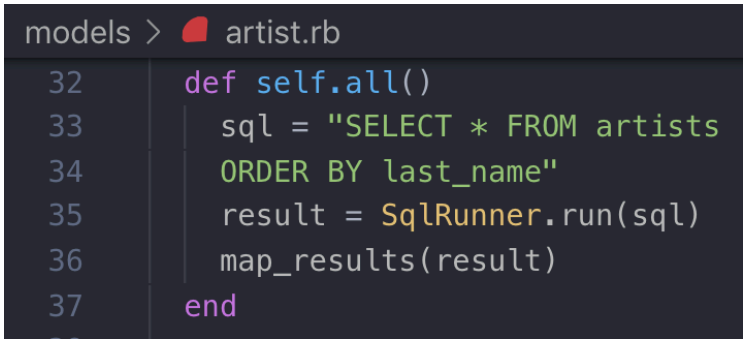
```ruby
def customers_booked()
  sql = "SELECT * FROM customers
  INNER JOIN tickets ON customers.id = tickets.customer_id
  INNER JOIN screenings ON screenings.id = tickets.screening_id
  WHERE film_id = $1"
  values = [@id]
  results = SqlRunner.run(sql, values)
  return map_results_to_customer(results)
end
```

The method above is called on an object of the *Film* class. It takes the *@id* instance variable of the *Film*, and searches the application's SQL database, returning all the lines from the *Customer* table where the *Customer* has a *Ticket* for a *Screening* of the film the method is called upon. It uses two inner joins of the SQL tables to achieve this.

```
[1] pry(main)> film1.customers_booked
=> [#<Customer:0x007fcdb91a0138 @funds=30, @id=70, @name="Mariette">,
 #<Customer:0x007fcdb919b200 @funds=20, @id=70, @name="Jean">]
```

Above shows the result of calling this method on the *film1* object; it returns two objects from the *Customer* class.

Unit	Ref	Evidence
I&T	I.T.4	Demonstrate sorting data in a program. Take screenshots of: *Function that sorts data *The result of the function running



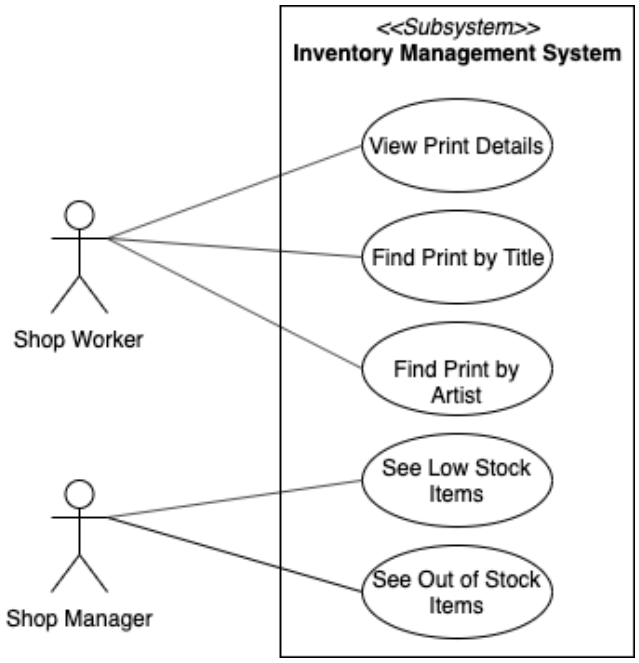
Above shows a class method of the *Artist* class, which sorts the data it returns by using the SQL ORDER BY statement to sort alphabetically by the *last_name* column.

```
[2] pry(main)> Artist.all
=> [#<Artist:0x007f9d96852d80
  @first_name="John",
  @id=22,
  @last_name="Constable">,
 #<Artist:0x007f9d96852cb8
  @first_name="Thomas",
  @id=26,
  @last_name="Gainsborough">,
 #<Artist:0x007f9d96852bf0
  @first_name="Edwin",
  @id=25,
  @last_name="Landseer">,
 #<Artist:0x007f9d96852b28
  @first_name="Henry",
  @id=24,
  @last_name="Raeburn">,
 #<Artist:0x007f9d96852a60
  @first_name="JMW",
  @id=23,
  @last_name="Turner">]
```

Above is the result of calling *Artist.all*. The returned objects of the *Artist* class are sorted alphabetically by the *@last_name* instance variable.

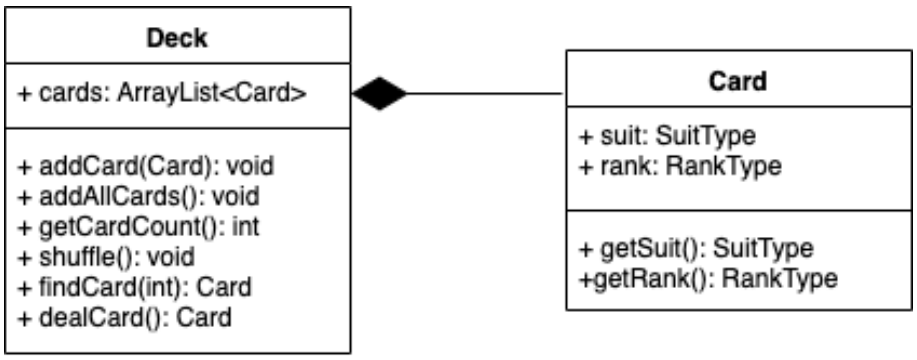
Week 4

Unit	Ref	Evidence
A&D	A.D.1	A Use Case Diagram



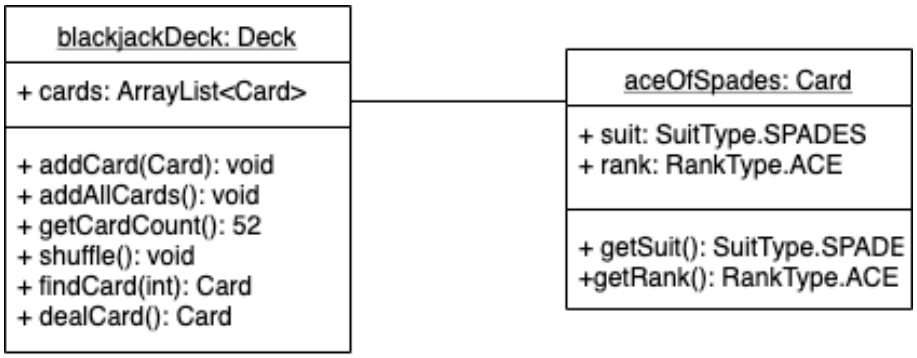
A Use Case Diagram for Art Print Inventory Management software, showing two actors, a Shop Worker and Shop Manager.

Unit	Ref	Evidence
A&D	A.D.2	A Class Diagram



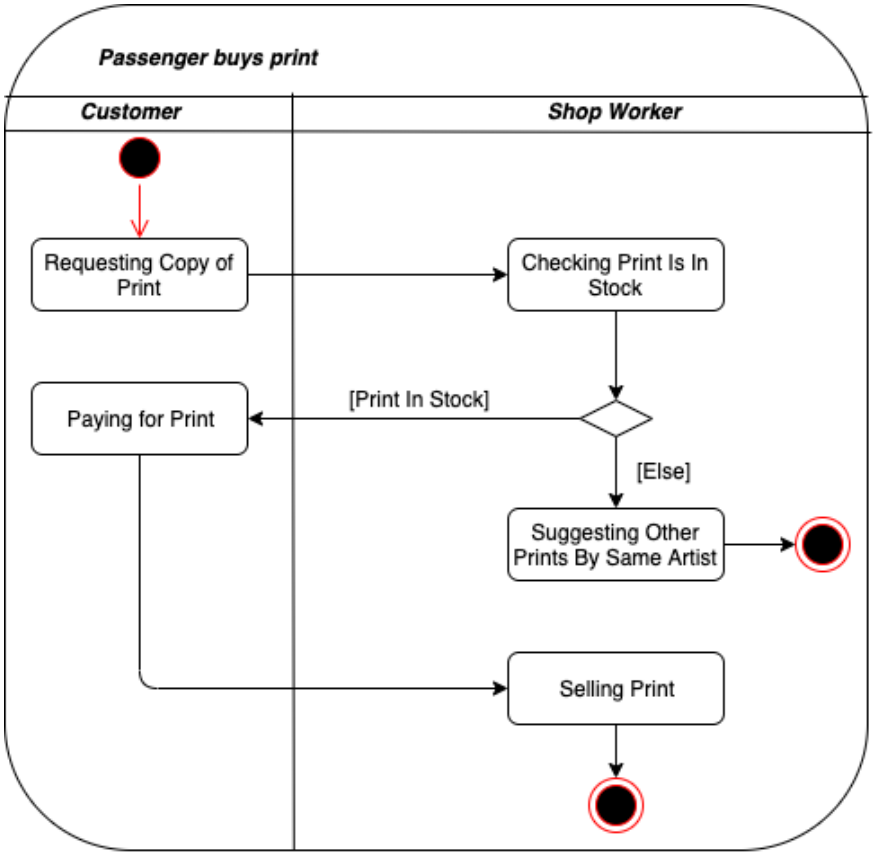
A Class Diagram showing the *Deck* and *Card* classes from a program to model the card game Blackjack. The *cards* ArrayList in the *Deck* class is composed of objects of the *Card* class.

Unit	Ref	Evidence
A&D	A.D.3	An Object Diagram



An Object Diagram showing examples of the *Deck* and *Card* classes from the previous diagram. The *blackjackDeck* instance of the *Deck* class has an ArrayList containing the 52 standard playing cards, instances of the *Card* class. The object *aceOfSpades* is an example of the *Card* class, it has *SuitType.SPADES* and *RankType.ACE*, both of which are controlled using enums.

Unit	Ref	Evidence
A&D	A.D.4	An Activity Diagram



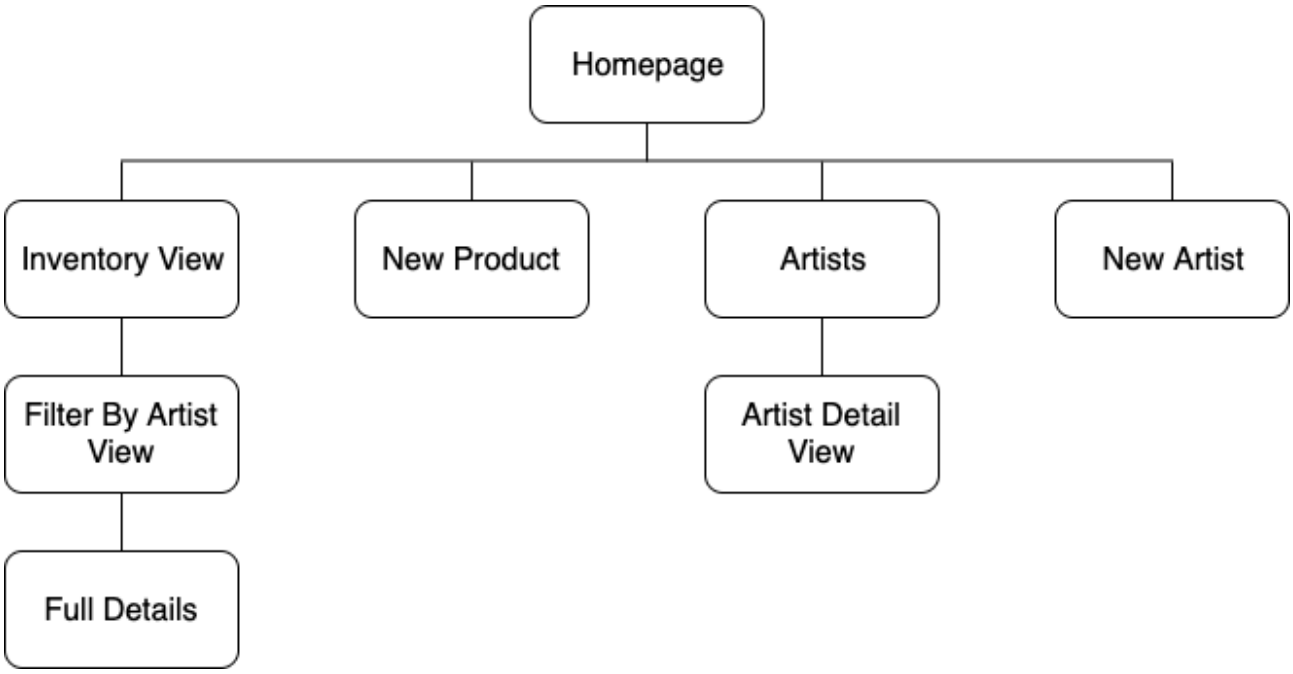
An Activity Diagram showing the process of a customer requesting to buy a print from a shop.

Unit	Ref	Evidence
A&D	A.D.6	Produce an Implementations Constraints plan detailing the following factors: *Hardware and software platforms *Performance requirements *Persistent storage and transactions *Usability *Budgets *Time

Constraint Category	Implementation Constraint	Solution
Hardware and Software Platforms	Browser compatibility: users with older browsers could experience broken functionality	Use only HTML and CSS features which are widely supported by all browsers
Performance Requirements	Reading and writing to the database could slow down performance of the app	Only perform the read and write operations at the point they are required
Persistent Storage and Transactions	Data is not stored persistently if running only the Ruby code	Connect a PostgreSQL database to the back end server for data persistence
Usability	The app runs in a browser, and the native Back and Forward buttons only provide limited navigation	Implement a persistent navigation bar so that users can navigate to the required page from anywhere in the app
Budgets	Proprietary software licenses can cause increases in the cost of a software project	Build the app with open source languages and frameworks such as Sinatra and PostgreSQL
Time	Project could run over time, with essential features missing, meaning it would not be usable	Set out a clear Minimum Viable Product and ensure this is completed before working on extra features.

Implementation Constraints Plan for Art Print Inventory Management software.

Unit	Ref	Evidence
P	P.5	User Site Map



A User Sitemap for the Art Print Inventory Management software, showing the different user views.

Unit	Ref	Evidence
P	P.6	2 Wireframe Diagrams

Wireframe for the Index view of the app, showing conditional highlighting for Low Stock (amber) and Out of Stock (red).

Wireframe for the Show view of the app, including buttons for Edit and Delete functionality.

Week 5

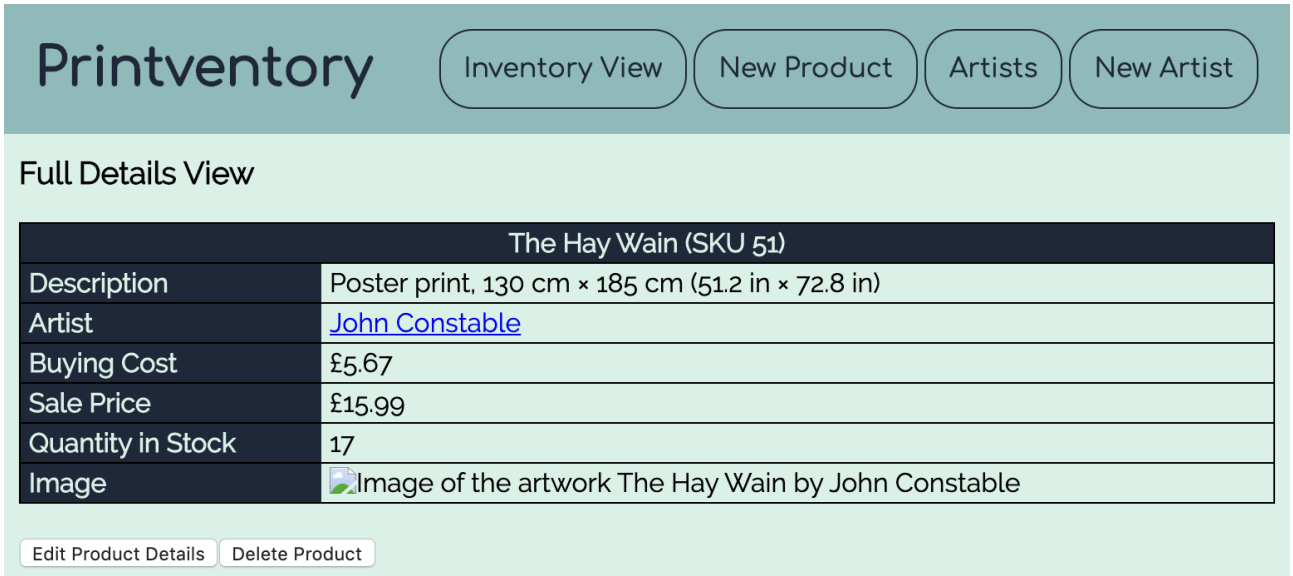
Unit	Ref	Evidence
P	P.10	Example of Pseudocode used for a method

```
Park.prototype.mostGuestsAttracted = function () {
  // Declare a variable with let, set it to the first exhibit in the array of exhibits
  // Loop through all the exhibits
  // For each exhibit, compare the value of guestsAttractedPerDay to that of the exhibit stored in the variable
  // If the exhibit's value is higher, assign it to the variable insetad of the previous exhibit
  // At the end of the loop, the variable will contain the exhibit with the highest guestsAttractedPerDay
  // The method should return the content of this variable
};
```

Pseudocode for a function in an app for a park of exhibits. The function is written in JavaScript and is to find the exhibit that attracts the most guests.

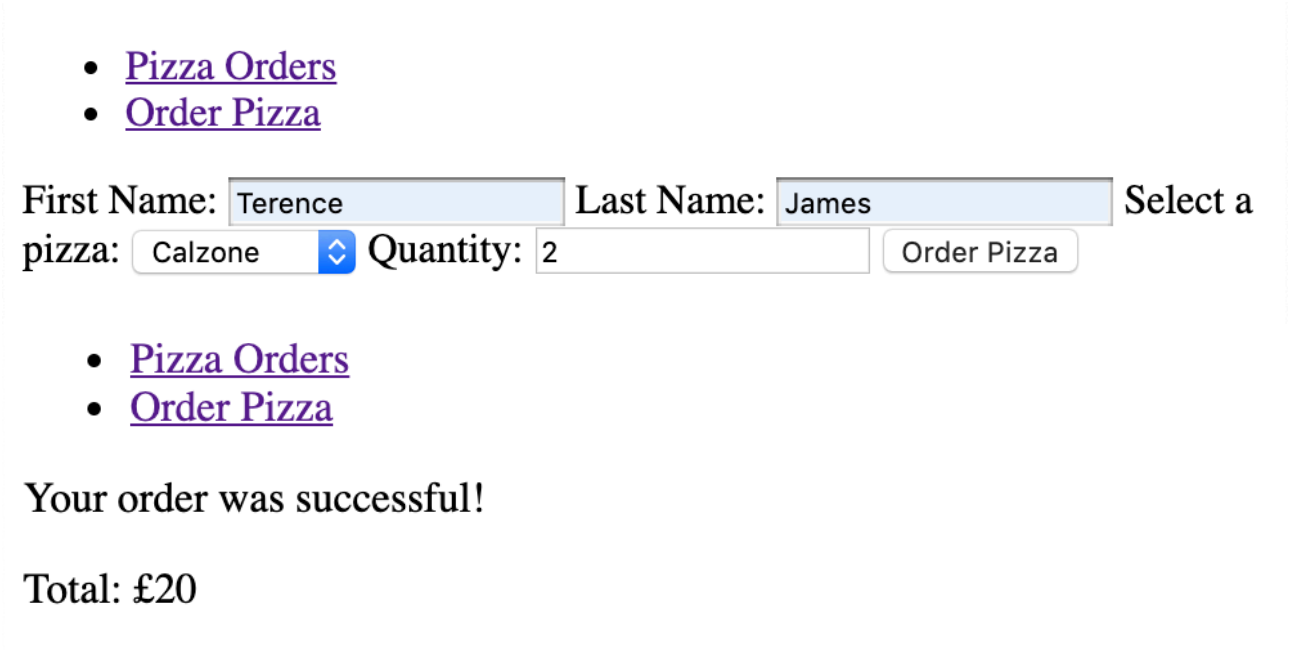
Unit	Ref	Evidence
P	P.13	Show user input being processed according to design requirements. Take a screenshot of: * The user inputting something into your program * The user input being saved or used in some way

Above shows the user entering new data into the form in the New view of the Printventory app.



Above shows the page that loads once the user clicks the Submit button. It is the Show view for the new *Print* record which has just been input.

Unit	Ref	Evidence
P	P.14	Show an interaction with data persistence. Take a screenshot of: * Data being inputted into your program * Confirmation of the data being saved

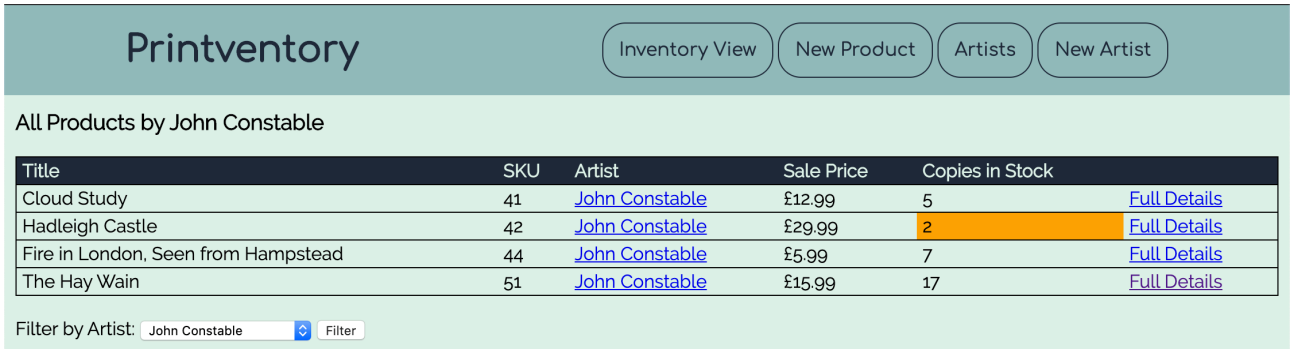


These two screenshots are from a pizza ordering app written in Ruby with Sinatra. The first shows the user entering a new order, the second is the confirmation screen that displays once the user has clicked the “Order Pizza” button, confirming that the data has been saved.

Unit	Ref	Evidence
P	P.15	Show the correct output of results and feedback to user. Take a screenshot of: * The user requesting information or an action to be performed * The user request being processed correctly and demonstrated in the program

Title	SKU	Artist	Sale Price	Copies in Stock	
A Highland Landscape	46	Edwin Landseer	£17.50	2	Full Details
An Irish Wolfhound	47	Edwin Landseer	£6.59	7	Full Details
Cloud Study	41	John Constable	£12.99	5	Full Details
Dugald Stewart Esq.	50	Henry Raeburn	£9.99	5	Full Details
Fire in London, Seen from Hampstead	44	John Constable	£5.99	7	Full Details
Hadleigh Castle	42	John Constable	£29.99	2	Full Details
Sheep Dog Sleeping	48	Edwin Landseer	£4.99	12	Full Details
Sir Walter Scott	49	Henry Raeburn	£6.50	0	Full Details
The Hay Wain	51	John Constable	£15.99	17	Full Details
Tummel Bridge, Perthshire	43	JMW Turner	£13.50	0	Full Details
Wooded Landscape with a Cottage and Shepherd	45	Thomas Gainsborough	£11.50	11	Full Details

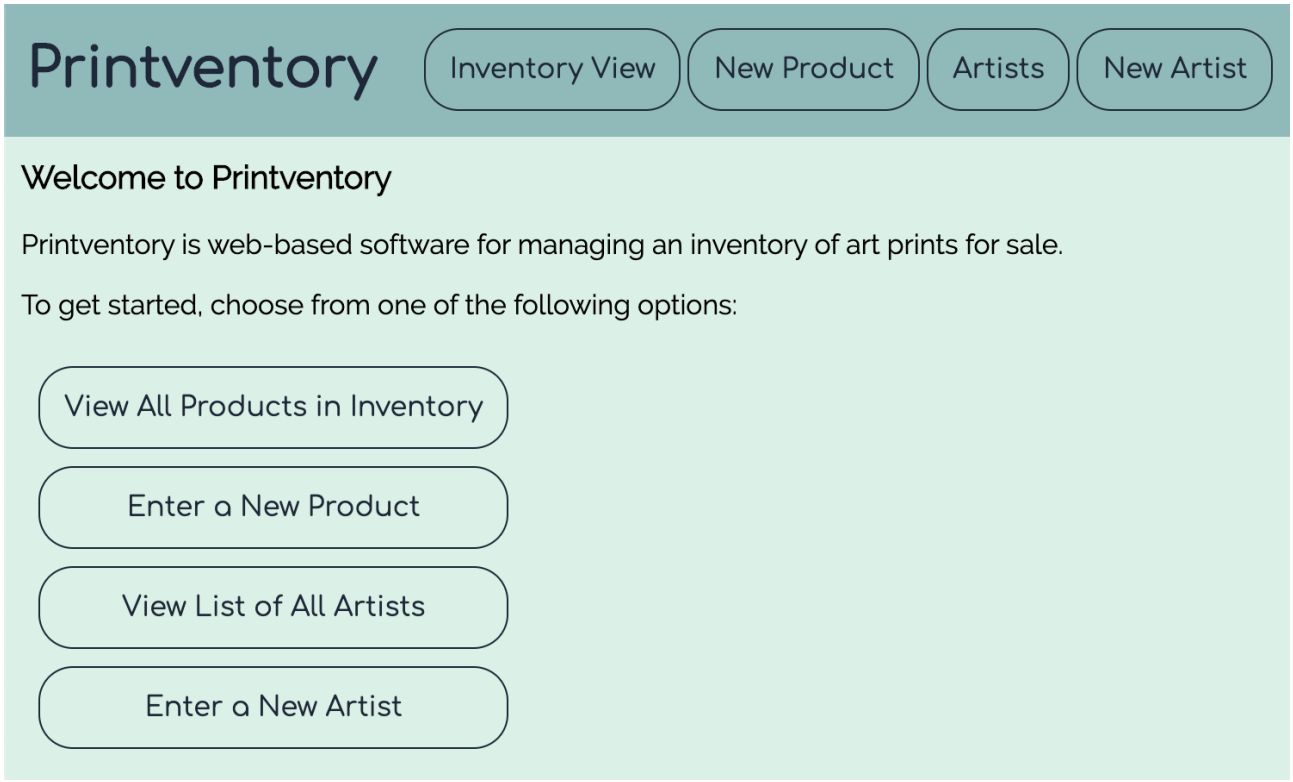
On the Index view, the user has chosen “John Constable” as an option in the “Filter by Artist” drop-down list, and then clicks the “Filter” button to request a view of prints only by that artist.



Title	SKU	Artist	Sale Price	Copies in Stock	
Cloud Study	41	John Constable	£12.99	5	Full Details
Hadleigh Castle	42	John Constable	£29.99	2	Full Details
Fire in London, Seen from Hampstead	44	John Constable	£5.99	7	Full Details
The Hay Wain	51	John Constable	£15.99	17	Full Details

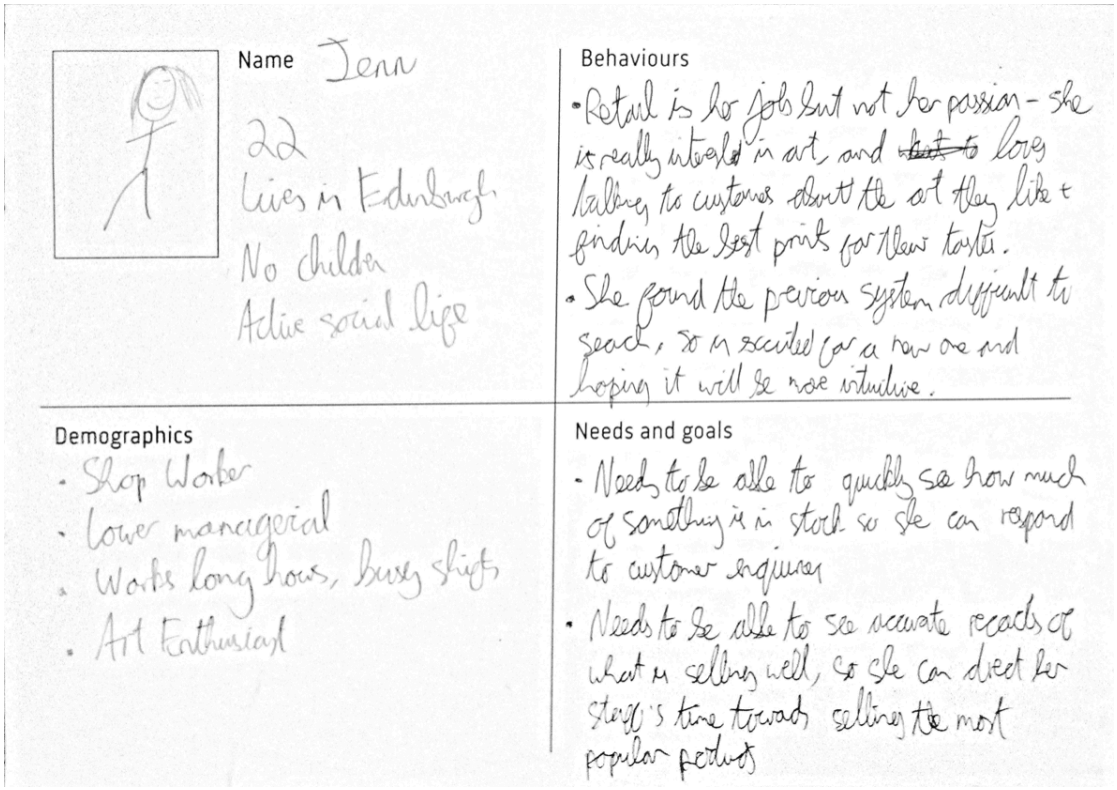
This request is processed by the program’s Sinatra web server, which returns the user a new view of just the products by John Constable, as shown above.

Unit	Ref	Evidence
P	P.11	Take a screenshot of one of your projects where you have worked alone and attach the Github link.

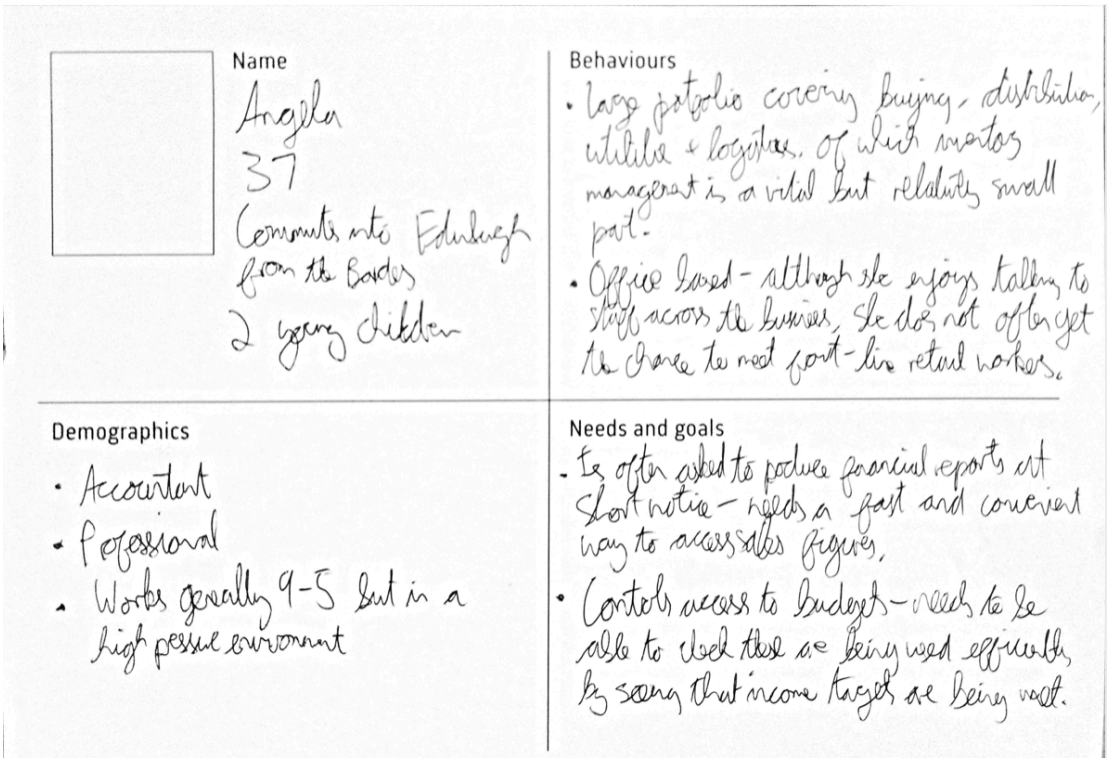


https://github.com/josephcooper3/Printventory

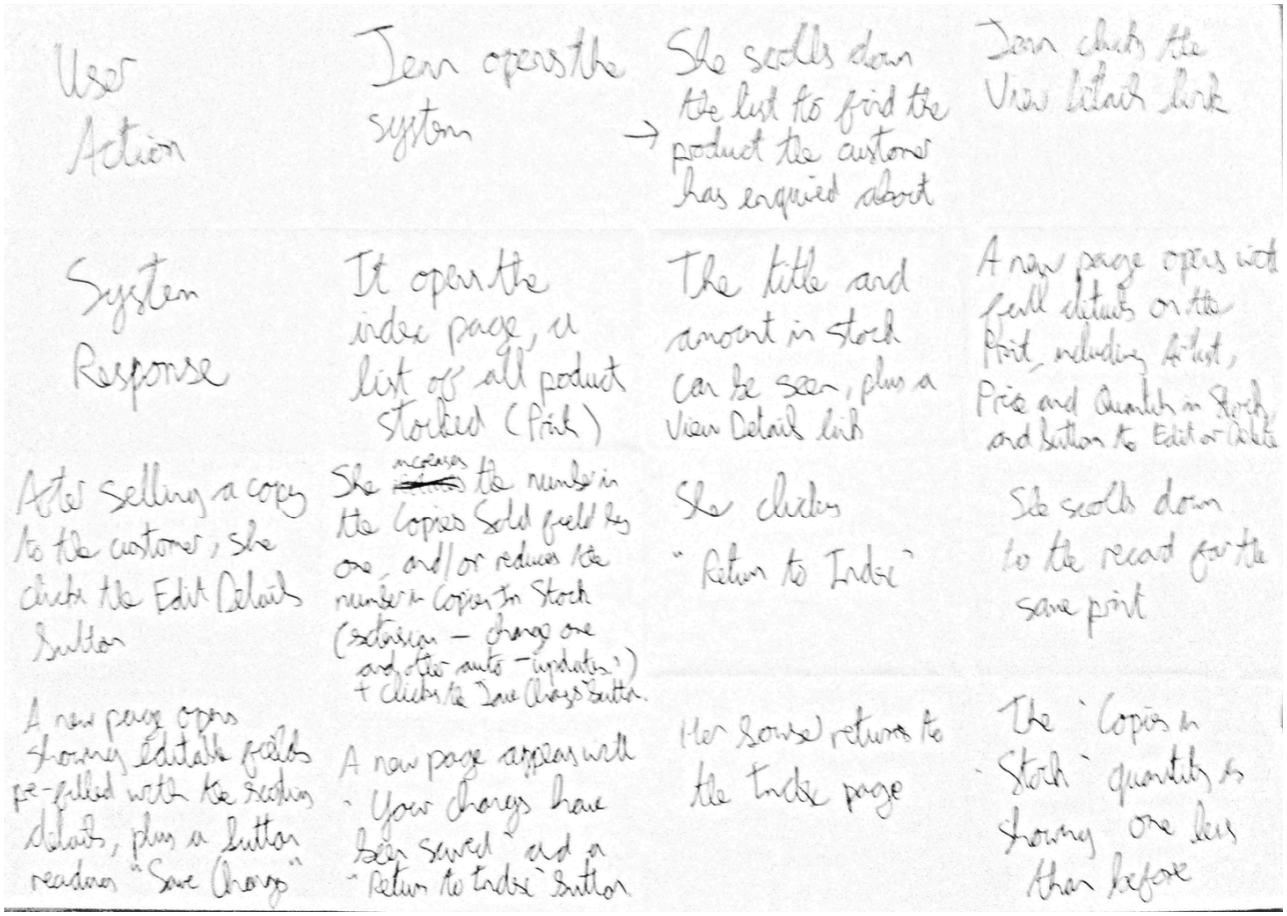
Unit	Ref	Evidence
P	P.12	Take screenshots or photos of your planning and the different stages of development to show changes.



Proto Person 1 - Jenn, shop worker, lower managerial, art enthusiast



Proto Persona 2 - Angela, professional accountant, budget holder



User Journey diagram - for Proto Persona 1

Printventory

Inventory View

SKU	Title	Artist	Sale Price	Copies in Stock	
54	Water Lilies	29	1299	5	Full Details
55	Guernica	30	2999	2	Full Details

First iteration of Index view

Printventory

Inventory View

SKU	Title	Artist	Sale Price	Copies in Stock	
59	Guernica	34	2999	2	Full Details
58	Water Lilies	33	1299	0	Full Details

Index view once Minimum Viable Product reached, showing amber highlight for low stock, red highlight for no stock, and Full Details links now active.

Week 7

Unit	Ref	Evidence
P	P.16	Show an API being used within your program. Take a screenshot of: * The code that uses or implements the API * The API being used by the program whilst running

```
methods: {
  fetchStatusDetails() {
    fetch(`https://api.tfl.gov.uk/Line/Mode/tube/Status?detail=true&app_id=${Key.app_id}
    &app_key=${Key.app_key}`)
    .then(result => result.json())
    .then(statusDetails => this.statusDetails = statusDetails)
    .catch(console.error);
  },
```

The method *fetchStatusDetails()* makes a request to the Transport for London API, which returns a list of all the London Underground lines and their current operational status.

Tube Line Status Checker

Filter Lines: Show All Lines

Refresh Status

Line	Service Status (click to see details)
Bakerloo Line	Good Service
Central Line	Good Service
Circle Line	Minor Delays Circle Line: Severe delays Clockwise and MINOR DELAYS Anti-clockwise due to a trespasser on the track earlier at Shepherd's Bush Market.
District Line	Part Closure
Hammersmith & City Line	Minor Delays
Jubilee Line	Good Service
Metropolitan Line	Part Closure
Northern Line	Good Service
Piccadilly Line	Part Closure
Victoria Line	Good Service
Waterloo & City Line	Service Closed Waterloo and City Line: Train service resumes 06.00 Monday

Tube Line Status Checker is a front-end app using the Vue.js framework. It is displaying the Service Status for each line based on the data returned from the *fetchStatusDetails()* function, which is run when the app is mounted. It also has a “Refresh Status” button which calls the *fetchStatusDetails()* function again.

Week 8

Unit	Ref	Evidence
P	P.2	Take a screenshot of the project brief from your group project.

Educational App

The BBC are looking to improve their online offering of educational content by developing some interactive browser applications that display information in a fun and interesting way. Your task is to make an a Minimum Viable Product or prototype to put forward to them - this may only be for a small set of information, and may only showcase some of the features to be included in the final app.

MVP

A user should be able to:

- view some educational content on a particular topic
- be able to interact with the page to move through different sections of content

Example Extensions

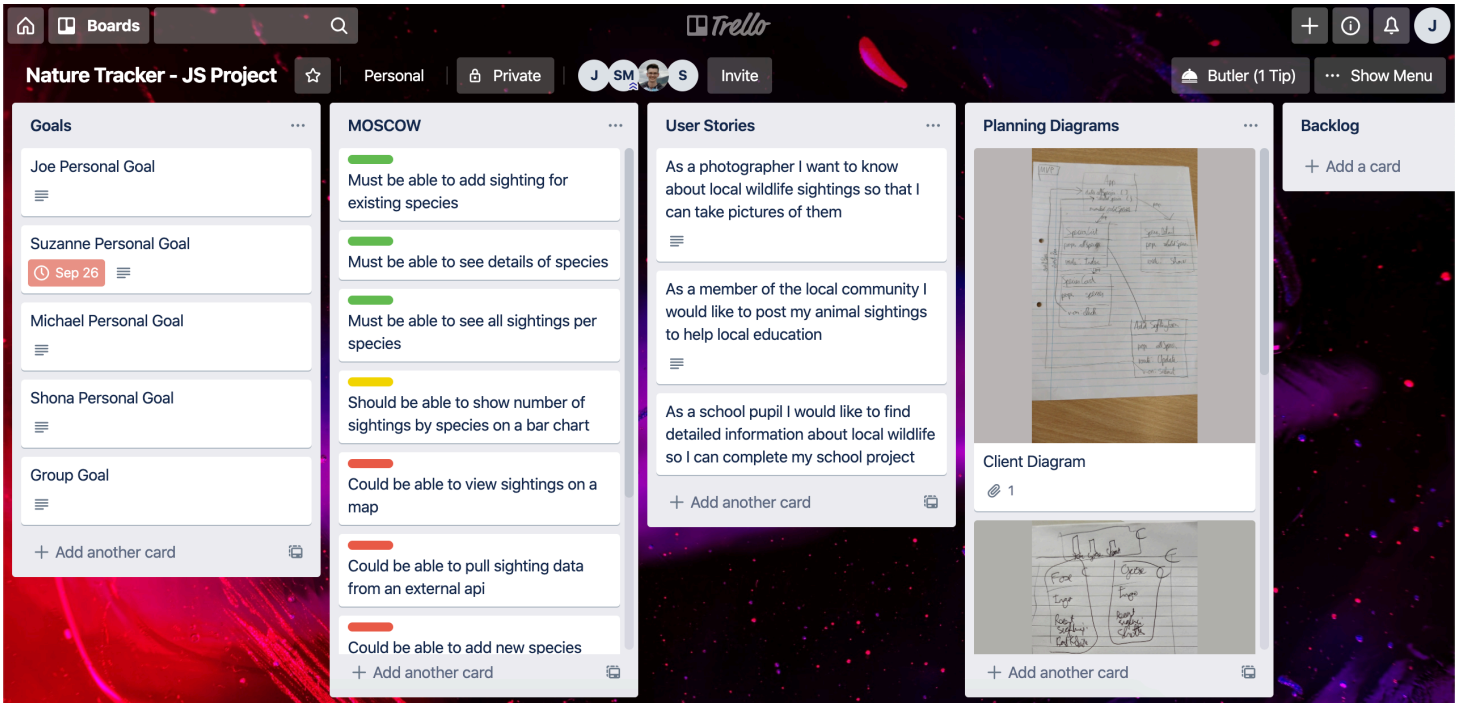
- Use an API to bring in content or a database to store information.
- Use charts or maps to display your information to the page.

API, Libraries, Resources

- https://www.highcharts.com/ HighCharts is an open-source library for rendering responsive charts.
- https://korigan.github.io/Vue2Leaflet/#/ Leaflet is an open-source library for rendering maps and map functionality.

The brief for our group project. This became “Nature Tracker”, an app to educate users such as schoolchildren and interested adults about local Scottish wildlife.

Unit	Ref	Evidence
P	P.3	Provide a screenshot of the planning you completed during your group project, e.g. Trello MOSCOW board.



Trello board for the Nature Tracker project. The left column features cards with the group's goal, and personal goals for each of the group members. The MoSCoW column has the Must-have features in green (required for MVP), Should-haves in Yellow and Could-haves in Red. The board also features User Stories for the porto-personas we had developed, and diagrams showing the data structures and Vue.js components.

Unit	Ref	Evidence
P	P.4	Write an acceptance criteria and test plan.

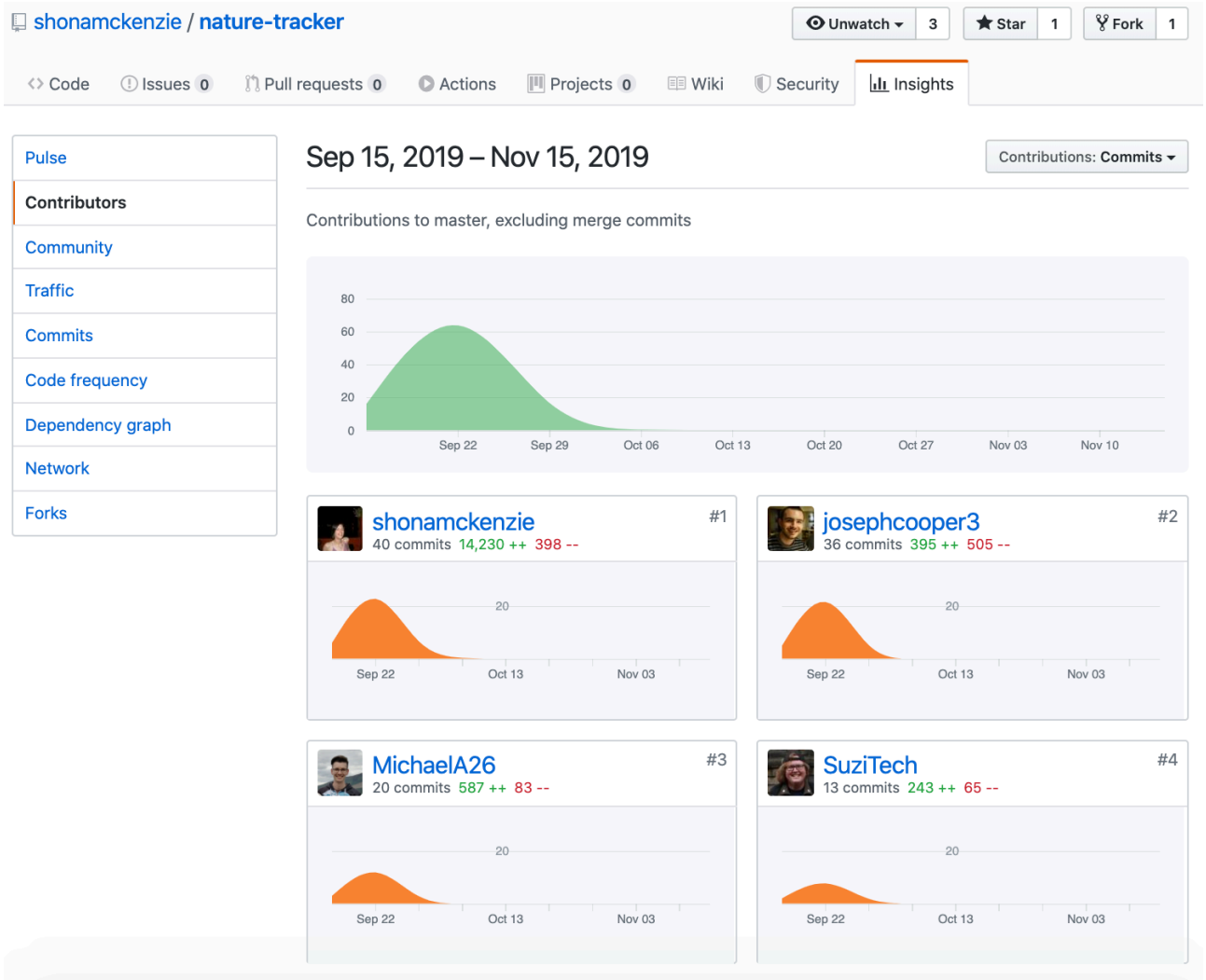
Acceptance Criteria	Expected Result	Pass/Fail
A user is able to see details of animal species	The app shows a card for each species on its homepage, and shows further details when the card is clicked	Pass
A user is able to see all the sightings of a given species	The app shows a table of sightings for the species when that species' card on the homepage is clicked.	Pass
A user is able to add new sightings for a given species	The app stores a new sighting object persistently when the user submits the Add New Sighting form	Pass

Acceptance Criteria	Expected Result	Pass/Fail
A user is able to see the number of sightings per species on a bar chart	The app shows a bar chart with a bar per species covered on the X-axis and the number of sightings on the Y-axis	Pass

Acceptance Criteria and Test Plan for the Nature Tracker app

Week 9

Unit	Ref	Evidence
P	P.1	Take a screenshot of the contributor's page on Github from your group project to show the team you worked with.



Contributors' page for the Nature Tracker project.

Week 11

Unit	Ref	Evidence
P	P.18	Demonstrate testing in your program. Take screenshots of: * Example of test code * The test code failing to pass * Example of the test code once errors have been corrected * The test code passing

```java
import java.util.ArrayList;

public class Library {

    private ArrayList<Book> books;
    private int capacity;

    public Library(int capacity){
        this.books = new ArrayList<Book>();
        this.capacity = capacity;
    }

    public int countBooks() {
        return this.books.size();
    }

    public void addBook(Book book) {
        this.books.add(book);
    }
}
```

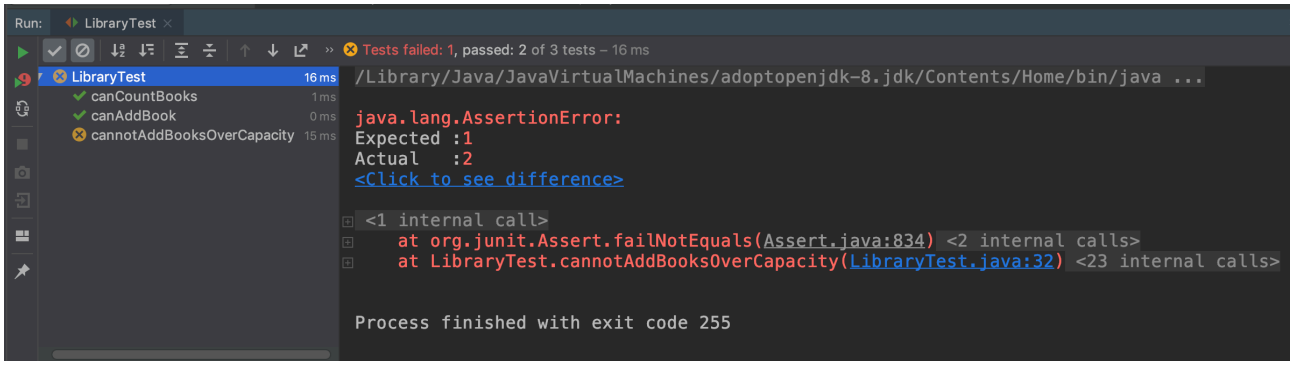
Above shows a Java class representing a *Library*. It has a variable for *Capacity* and and *addBook* method which takes an instance of the *Book* class.

```java
@Test
public void cannotAddBooksOverCapacity(){
    library.addBook(book1);
    library.addBook(book1);
    assertEquals( expected: 1, library.countBooks());
}
```

Above is a test (using the JUnit framework) to test that more instances of the *Book* class than its capacity cannot be added to the Library.

```java
@Before
public void before(){
    library = new Library( capacity: 1);
    book1 = new Book( title: "Pride and Prejudice", author: "Jane Austen", genre: "Romance");
}
```

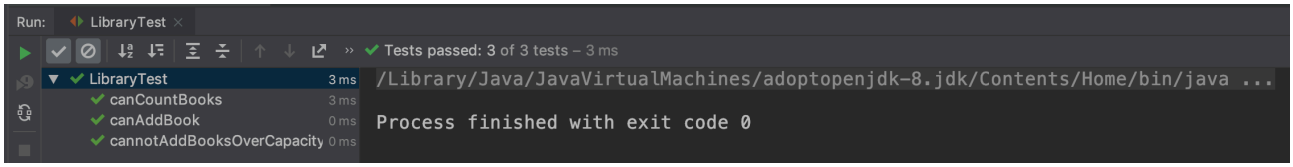
Above is the testing data used with the test; the instance of the *Library* class has a capacity of 1.



Above shows the test failing; library.countbooks() has returned 2, even though the capacity was only 1, meaning Book objects were still able to be added past the Library's capacity.

```
public void addBook(Book book) {
    if (capacity > this.countBooks()){
        this.books.add(book);
    }
}
```

Above shows the *addBook* method rewritten using an if statement, so that the Book is only added to the Library's *books* ArrayList if the capacity is less than the number of books (using the *countBooks()* method).



Above shows the same test passing once the *addBook()* method had been corrected.

Unit	Ref	Evidence
I&T	I.T.1	The use of Encapsulation in a program and what it is doing.

Book.java

```
1   public class Book {
2
3       private String title;
4       private String author;
5       private String genre;
6
7   @   public Book(String title, String author, String genre){
8           this.title = title;
9           this.author = author;
10          this.genre = genre;
11      }
12
13      public String getTitle() {
14          return this.title;
15      }
16
17      public String getAuthor() {
18          return this.author;
19      }
20
21      public String getGenre() {
22          return this.genre;
23      }
24  }
```

The above screenshot shows a Java class representing a Book, using encapsulation. Encapsulation bundles the data describing the book together with the methods operating on the data. This means that the instance variables (*title*, *author* and *genre*) can be stored as private, and are instead accessed by the class's Getter methods which are public.

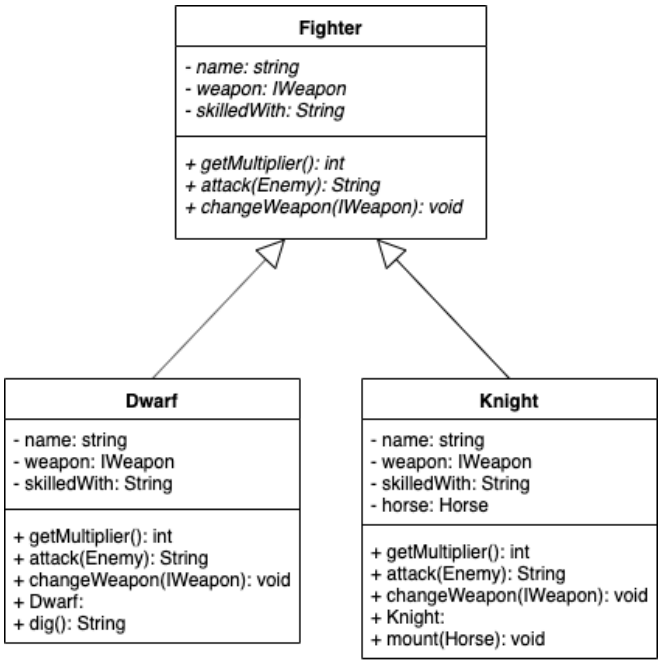
Week 12

Unit	Ref	Evidence
I&T	I.T.7	The use of Polymorphism in a program and what it is doing.

```java
package beings;

import abilities.IWeapon;

public class Fighter extends Player {

    private IWeapon weapon;
    private String skilledWith;

    Fighter(String name, IWeapon weapon, String skilledWith) {
        super(name);
        this.weapon = weapon;
        this.skilledWith = skilledWith;
    }
```

The Java code above shows a class *Fighter* from a programme modelling an adventure game. The *weapon* instance variable of this class is polymorphic, as it can take any object that implements the *IWeapon* interface. This means that different classes that behave differently can all be passed into the constructor of Fighter, as long as the implement *IWeapon*.

Unit	Ref	Evidence
A&D	A.D.5	An Inheritance Diagram



An Inheritance Diagram from the adventure game model. The *Dwarf* and *Knight* classes both inherit from *Fighter*. *Dwarf* has an additional *dig()* method while *Knight* has an additional *horse* variable and a *mount*() method which takes an instance of *Horse* as a parameter.

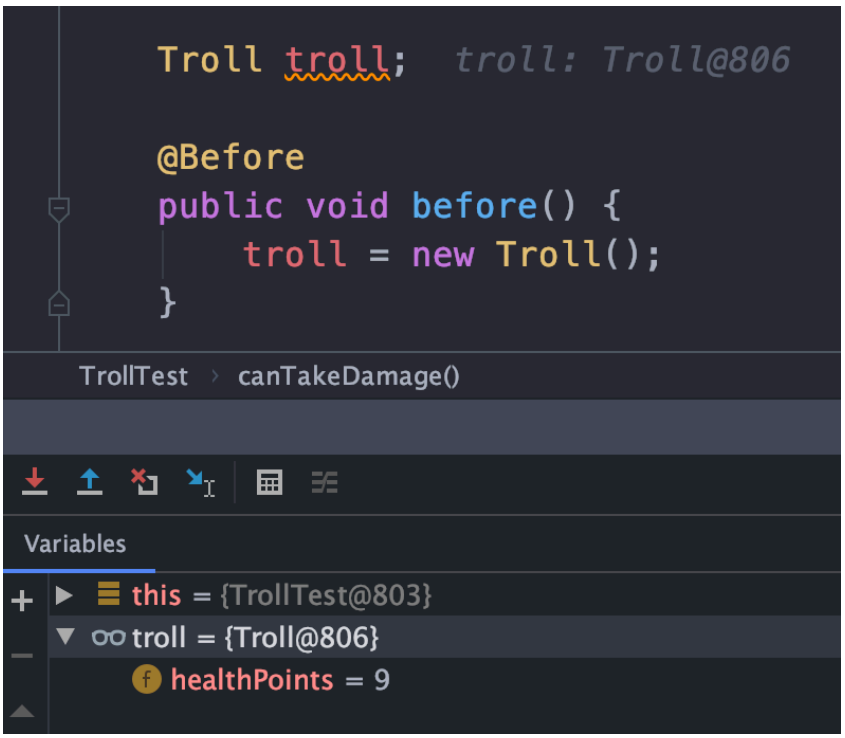
Unit	Ref	Evidence
I&T	I.T.2	Take a screenshot of the use of Inheritance in a program. Take screenshots of: *A Class *A Class that inherits from the previous class *An Object in the inherited class *A Method that uses the information inherited from another class.

```java
package beings;

public abstract class Enemy {

    private int healthPoints;

    public Enemy(int healthPoints) {
        this.healthPoints = healthPoints;
    }

    public int getHealthPoints() {
        return this.healthPoints;
    }

    public void takeDamage(int amount) {
        if (this.healthPoints - amount >= 0) {
            this.healthPoints -= amount;
        } else {
            this.healthPoints = 0;
        }
    }
}
```

An Abstract Class of *Enemy*, from the adventure game programme.

```java
package beings;

import abilities.IDefend;

public class Troll extends Enemy implements IDefend {

    public Troll() {
        super( healthPoints: 10);
    }

    public int defend() {
        return 1;
    }
}
```

A Class of *Troll* which inherits the variables and methods of the *Enemy* class. It differs in that its constructor method initialises the *healthPoints* variable at *10*. It additionally implements the *IDefend* interface and so has an additional *defend()* method.



The above screenshot is from the TrollTest spec file, and shows a new instance of the *Troll* class being instantiated, as well as showing the instance including the Troll's healthPoints variable, in IntelliJ's debugger.

```
public String attack(Enemy target, int multiplier) {
    int damageToDeal = this.damage * multiplier;
    if (target instanceof IDefend) {
        damageToDeal -= ((IDefend) target).defend();
    }
    target.takeDamage(damageToDeal);
    return "Swish Thwack!";
}
```

The above method is from the *Axe* class. It takes an instance of the *Enemy* class as its parameter, and calls the *takeDamage()* method of the Enemy. Additionally, if the *Enemy* implements *IDefend* (such as the *Troll* above) then it calls the *Enemy*'s *defend()* method.

Week 14

Unit	Ref	Evidence
P	P.9	Select two algorithms you have written (NOT the group project). Take a screenshot of each and write a short statement on why you have chosen to use those algorithms.

```
Park.prototype.mostGuestsAttracted = function () {
  let foundDinosaur = this.dinosaurCollection[0];
  for (const dinosaur of this.dinosaurCollection) {
    if (dinosaur.guestsAttractedPerDay >
    foundDinosaur.guestsAttractedPerDay) {
      foundDinosaur = dinosaur;
    }
  }
  return foundDinosaur
};
```

The algorithm above is from a programme modelling a fictional dinosaur park. The *Park* class has an array of *Dinosaur* objects. The algorithm sets a variable *foundDinosaur* to the first object in this array, then loops through all the objects comparing the *guestsAttractedPerDay* instance variable. If it is higher than that of *foundDinosaur*, *foundDinosaur* is reassigned to the *dinosaur* from the loop. Therefore when the *for* loop has finished running, *foundDinosaur* is the d*inosaur* with the highest *guestsAttractedPerDay.* The algorithm then returns this *foundDinosaur*.

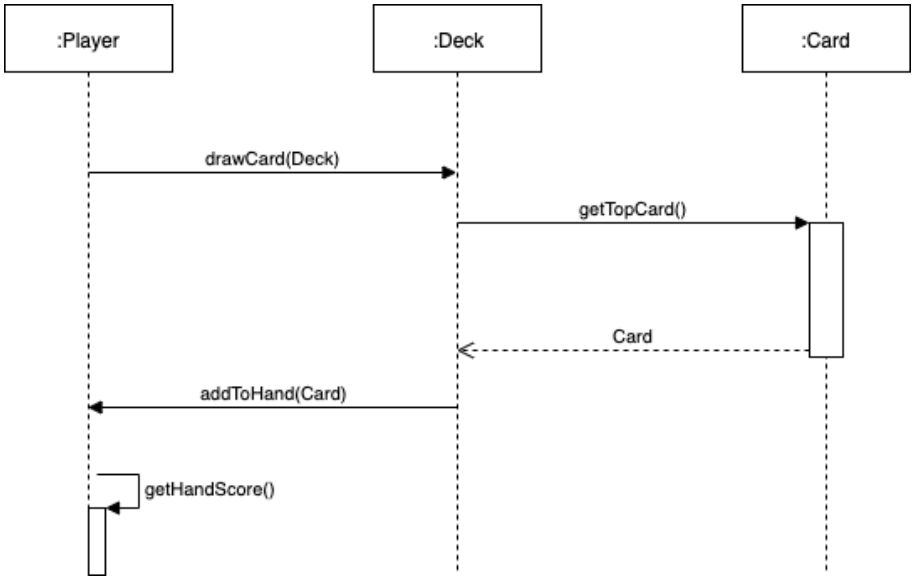
I chose this algorithm as it shows how variables can be compared using an *if* statement within a loop, and also demonstrates two different ways of declaring variables in JavaScript (*let* and *const*) and the scoping of these variables within the block. *foundDinosaur* must be declared with let as it may be reassigned as the loop runs, and must be declared outside the loop so it can be returned at the end of the loop.

```
Park.prototype.allDinosaursBySpecies = function (species) {
  const foundDinosaurs = []
  for (const dinosaur of this.dinosaurCollection) {
    if (dinosaur.species === species) {
      foundDinosaurs.push(dinosaur);
    }
  }
  return foundDinosaurs;
};
```

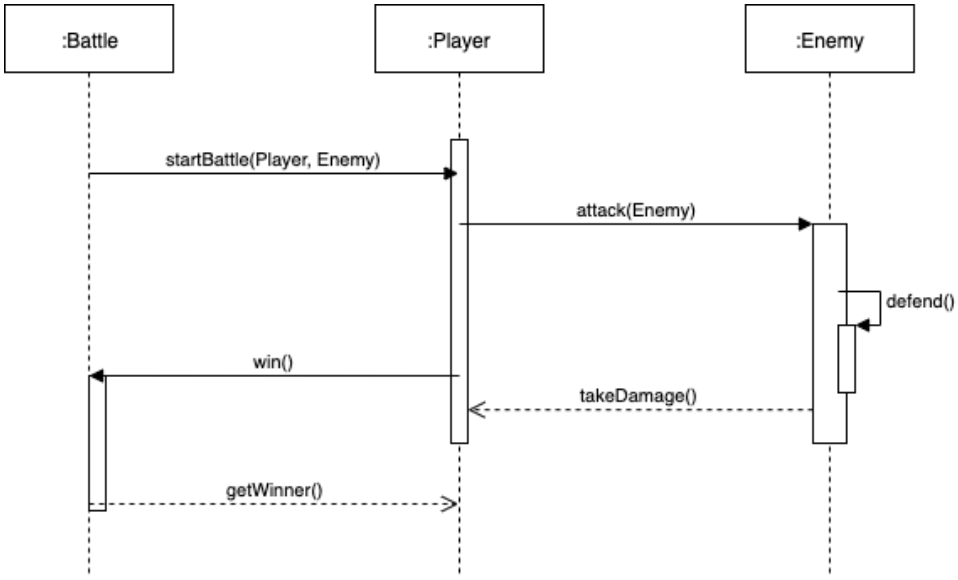
The second algorithm above is from the same programme. It takes an argument of *species*, and then loops through the array of dinosaurs to return all dinosaurs whose *species* instance variable matches the argument passed in.

I chose this as it is an example of how an algorithm can return varying numbers of results depending on the arguments passed in. This is achieved by setting the variable to be returned as an array. The contrast with the first algorithm also demonstrates the difference between variables declared with *let* and *const* - in the second algorithm, *const* still allows objects to be added to the array with *push()*, as the variable itself is not being reassigned.

Unit	Ref	Evidence
P	P.7	Produce two system interaction diagrams (sequence and/or collaboration diagrams).

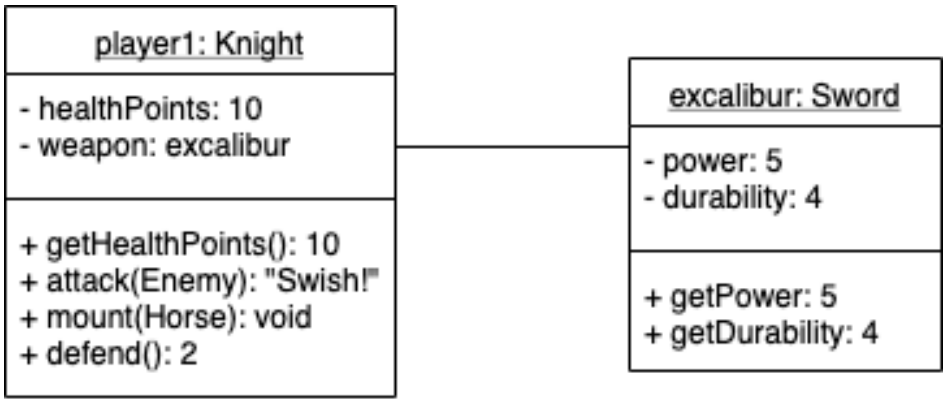


This first sequence diagram, from a card game programme, shows the process of a player drawing a card from a deck. The *Player* calls its *drawCard* method taking the *Deck* as an argument. The *Deck* then calls its *getTopCard()* method which returns the *Card* object at index 0 in its ArrayList of *Card* objects. The *Player* then calls its *addToHand* method taking the returned *Card* as a parameter, which adds it into its *Hand* ArrayList. Finally the player calls *getHandScore()* which calculates the score of the *Card* objects in its *Hand*, including the newly drawn card.

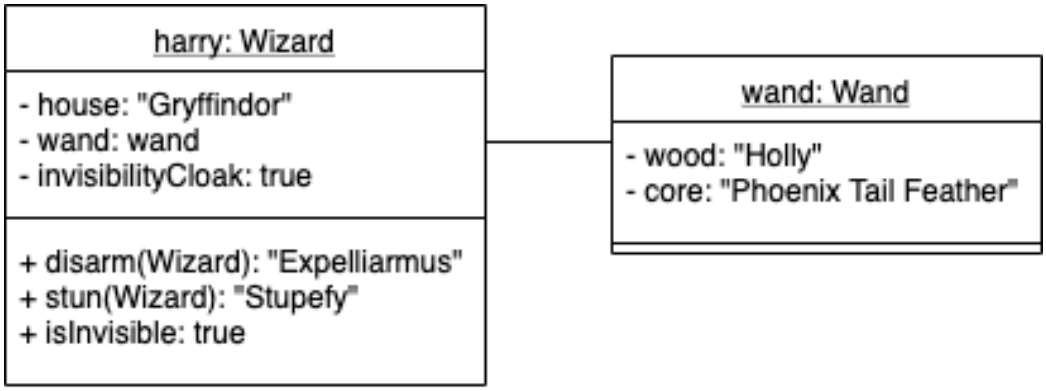


This second sequence diagram is from the programme modelling an adventure game, and shows the sequence of a battle. The *Battle* object calls its *startBattle* method, taking the *Player* and *Enemy* as arguments. The player then calls its *Attack* method taking the *Enemy* as its argument, which triggers the enemy's *defend()* method and then its t*akeDamage()* method, which depends on how much of the damage is prevented by the Defend method, and returns the damage taken to the Player. The Player is then able to call its *win()* method, and is then returned by the Battle's *getWinner()* method.

Unit	Ref	Evidence
P	P.8	Produce two object diagrams.



An Object Diagram from the adventure game programme. The object *player1* is an instance of the *Knight* class, and its *Weapon* variable is composed of an instance of the Sword class.



A second object diagram from the same programme. The object *harry* is an instance of the *Wizard* class, and has a *wand* variable composed of an instance of the *Wand* object.

Unit	Ref	Evidence
P	P.17	Produce a bug tracking report

Bug/Error	Solution	Date
App crashes when attempting to add a new sighting.	Request a new LocationIQ API key. Previous key had expired.	11/11/2019
Date display format in sightings list is in YYYY-MM-DD format whereas users would expect the UK standard of DD-MM-YYYY.	Write a method to rearrange the date between form submission and saving to the database (as the date picker in the form can only submit YYYY-MM-DD).	27/09/2019
Entry of new sighting from bottom of page form does not work, but still works when using the form at the top of the page.	The form at the bottom needs to be given the same props as the one at the top so it works in the same way.	27/09/2019
Component speciesCard.vue should be named SpeciesCard.vue instead.	Use git mv to rename file and then commit the change.	26/09/2019
The Add Sighting form requires latitude and longitude to be entered before submitting, but users are unlikely to know this.	Remove these fields from the input form, and instead use the LocationIQ API to automatically fill these in as records are saved.	24/09/2019

A bug tracking report from the Nature Tracker app.