# 14/02/2020 (Friday - 2h 30m: 2:45pm to 5:15pm)

* We discussed timings for future meetings and began evaluating the required functionality of the program as well as consideration for the best ways to implement said functionality.
* We created a shared work file so we can be viewing the log and code on our computers whenever we meet up and easily make changes to the code whilst writing our report.
* We planned what kinds of data structures would be used for different parts of the program.

# 15/02/2020 (Saturday - 1h 45m: 12:30pm to 2:15pm)

* We began by evaluating the exception classes as well as the basic classes sent with the project briefing.
* We considered storing data within a dictionary type structure, where the key is the 8-character hex identifier and the value contains a list of information pertaining to different configurations of beanbags.
  + For stock indexing to work with this data structure we would need to include a field for quantity such that any bean bags which are interchangeable with identical details are counted correctly as these could have identical hex identifiers or different identifiers if the differed in colour.
  + We needed to assess the ObjectArrayList to see how best to implement the desired data structure without the need for additional modules.

# 16/02/2020 (Sunday - 1h 10m: 1:30pm to 2:40pm)

* We drafted an example data structure listing all the information required for an individual bean bag.
  + This was done using the examples from the Class, Strings, Array Workshop.
* Person A suggested testing manipulation of data by starting off with a simple function to edit the quantity of an item using its Hex identifier.
* Person B attempted to implement this class and create a driver class to run all the methods contained within the package.
* The Week 5 Quiz included an example of reading from text files.
  + We considered later implementing this feature in our program so that the state of the stock manager is saved upon closing, but first needed to ensure all our functionality worked.

# 17/02/2020 (Monday - 50m: 9:35am to 10:25am)

* We considered the different data structures that could be implemented.
* We settled on an ObjectArrayList which operates with a dictionary structure using keys and values.
  + Instances of an object can have identical names as they are stored in different memory locations as such it would be possible to count the quantity of beanbags in stock by counting the number of objects created.
  + When an item with an identical hex ID is sold you can simply delete one of the instances of the matching objects.

# 18/02/2020 (Tuesday - 4h: 10am to 2pm)

* Person A completed a rough draft of the UML diagram and set up a GitHub repository which would allow us to keep track of code revisions and later add the package upon completion as one our projects on LinkedIn.
* Person B wanted to begin the project by implementing a simple class responsible for sending data to the ObjectArrayList as mentioned previously, where the object names would be the HexID of the beanbags.
  + Contained within each object would be all the information relating to a beanbag.
  + We started with a single bean bag and wanted to attempt to change the data held by the object. The simplest test case we devised would be to change.
* Person A suggested a more detailed revision of the availability of both partners, so we decided to add our availability to each other’s calendars.
* Person B reconfigured the directories to match those suggested in the ‘Jar File Creation Walkthrough’ which was recently released.
* Person A found a nice website called draw.io which allowed for seamless completion of our UML file which had previously been drawn out on paper.
  + This was based on the diagram given in the Class, String, Array workshop.

# 20/02/2020 (Thursday - 1h 40m: 12:40pm to 2:20pm)

* In our OOP lecture today, we learnt about inheritance and the keywords 'super' and 'extends' to refer to parent and child classes.
* Person A considered that we could have the Reservations class inherit the constructor method from Stock classes.
  + To do this this subclass needs to extend the superclass as such we could include additional parameters for 'reserved (Boolean)’, 'reference (String)', 'customer (String)'.
  + The access modifier for an overriding method can allow for more but not less access than the overridden method so this approach should be suitable for our application.
* We started to break down the briefing that we had been given to shortlist the key information provided and bullet point the key features that we needed to implement in our system. The package is required to:
* Keep track of several attributes:
  + An 8-digit hex id (String)
  + Manufacturer (String),
  + Bean bag name (String),
  + priceInPence (int),
  + Year of manufacture (short),
  + Month of manufacture (byte),
  + Number of a type of bean bag (int) both remaining and later sold,
  + Optional free text component containing any additional information (String).
* Ability to reserve items currently in stock
* Reserved items must only be sold to customers with a matching reservation number.
* Customers can cancel reservations and add items back to general item stock.
* We would need to keep track of an attribute relating to reservationNumber (int)
* Ability to change price of items
* Must ensure that and items reserved are sold at the lower price.
* Hence, both agreed it would be appropriate to split the problem up with separate classes for dealing with the reservation system, tracking sales totals and managing the store’s stock.
* Person A said that from the information provided it looks like it would be possible to create multiple different types objects containing varying attributes for example one type of objects for stock management and separate type of object for reservations, whilst still being able to store all these objects within the same ObjectArrayList class provided.
* Person B liked the sound of this idea but said it would be wise to start by declaring the required attributes as null variable within their appropriate classes.
  + This would allow for compartmentalisation of different kinds of functions within the overall package where similar functions relating to the same feature would be grouped together.

# 23/02/2020 (Sunday - 5h 30m: 12pm to 5:30pm)

* Person B said we would need two constructor methods one which included a reference to the optional description variable containing the free text and a second identically named method without this variable.
  + When running the package, the class selects the appropriate method based on the number of variables included when calling the method.
* Person A remembered that no variable for quantity need be included in the constructor as when multiple identically named beanbags are in stock, they would each be referenced as separate objects in memory within the java virtual machine.
* Person B labelled methods getter and setter declared private variables alongside the public methods to call said variables.
* Person A created a new class called Stock responsible for managing any object containing information relating to the details of beanbags in stock.
  + As discussed above we intend to segregate functionality within the program such that we are implementing several smaller classes rather than one large class containing all functionality.
* We discussed at length about the benefits of separating different parts of the package some of the benefits include upgradeability of the system, readability, error checking/bug fixes, reliability.
  + This may however come at the cost of a less efficient system in terms of the amount of storage required to store a greater number of objects in the ObjectArray.
  + This is as a result of splitting up reservations and beanbag details into two separate objects.
* Upon later review of our code, person B realised that within the reservations object we would need to store the price of the item at the time it was reserved.
  + This would allow us to later compare this price with the current price of any identical items in general stock and change the customer the lower of the two prices.
  + As such Person A changed the code to include a field for the original price.
  + Included an additional getter method in order to retrieve this new field from the object.
* The tables below detail the two different methods that could be implemented in our system. The latter is the format that we have decided to go with however as we continue to add functionality to our program, we will continually evaluate if this data structure is the most appropriate.
* For a stock object:

|  |  |
| --- | --- |
| **8-Bit Identifier (String)** | A1B2C3D4 |
| **Manufacturer (String)** | Argos |
| **Name (String)** | Recliner |
| **Price (double)** | 29.99 |
| **Year of Manufacture (int)** | 2019 |
| **Month of Manufacture (int)** | 11 |
| **Optional Description (String)** | Teal Blue |

* For reservations:

|  |  |
| --- | --- |
| **8-Bit Identifier (String)** | A1B2C3D4 |
| **Reserved (Boolean)** | True |
| **Reservation Reference (String)** | Order-1234 |
| **Customer Name (String)** | Jane Doe |

* The first possibility for data handling would require two separate objects for any reservation. This would lead to better readability within the program and all the other benefits listed above.
* The second method below incorporates all data into a single object so takes up less storage, but any changes made to the code would be more difficult to make due to reduced readability.
  + Hence, bug fixes and error checking would be far more difficult, hindering development of the package classes.
* For an object combining both stock and reservations:

|  |  |
| --- | --- |
| **8-Bit Identifier (String)** | A1B2C3D4 |
| **Manufacturer (String)** | Argos |
| **Name (String)** | Recliner |
| **Price (double)** | 29.99 |
| **Year of Manufacture (int)** | 2019 |
| **Month of Manufacture (int)** | 11 |
| **Optional Description (String)** | Teal Blue |
| **Reserved (Boolean)** | True |
| **Reservation Reference (String)** | Order-1234 |
| **Customer Name (String)** | Jane Doe |

* Following on from this, Person B suggested that for the reservations class we could follow the same approach as the stock class, such that any reservation objects are named using the reservation number.
  + The difference would be that for reservation, we would never have any identical objects.
* Person B found an error with Person A’s code; even when calling the reservation object by its reservation reference, we would still need access to the hex identifier to change the availability of reserved items.
  + We updated our Reserved class to include a field for the identifier within the object.
* Currently, the reserved class only includes a setter method for changing the Boolean value referring to whether the item is reserved.
  + If this is set to false, then the Reservation object is removed from the ObjectArrayList and available for purchase within the general store.
  + Later, we may choose to include setter methods for changing the beanbag that a customer has decided to reserve or the customer name relating to that reservation.
  + However, to keep the initial program simple currently, these fields are only accessible through getter method.
    - Therefore, in order to change any details relating to a reservation, the store would first need to cancel the original reservation before creating a new reservation entirely.
    - This currently isn’t a particularly major issue but is something we could revisit at a later stage depending on the functionality required.
    - In some instances this implementation may be better than the alternative, as it reduces the possibility of accidental deletion or editing of details relating to items that a customer has already decided to reserve.

# 24/02/2020 (Monday - 1h 40m: 12:40pm to 2:20pm)

* Started coding for our project.
* Person A took on the role of the driver whilst Person B was the observer in this instance.
* Began by creating a new Store class and copying over all the constructor methods listed in the BeanBagStore interface class.
  + Person B suggested that we work by first creating a template of all the getter and setter methods described in the briefing and highlighting any key areas where we could potentially reuse parts of the code by overriding certain methods.
  + After we had completed the outline of our project and what it was going to look like, we began filling in the simpler methods responsible for creating bean bag objects.
  + We also coded getters and setters for accessing and changing the quantity of a bean bag stored in the object.
* Person B had realised that the data types of each parameter that we needed to state were given in the BeanBagStore interface.
  + Hence, Person A changed some of the variable types such as month from int to byte and year from int to short.
  + We also took this opportunity to create a separate class containing our main method used to control the rest of the package.

# 26/02/2020 (Wednesday - 1h 35m: 12:40pm to 2:15pm)

* This session Person B was the driver and Person A the observer.
* Person A realised that later on in development, when we get to creating the getter method for year of manufacture, it would be possible to use a switch statement to convert the byte data type to a String type, containing the full name of the month in which the beanbag was manufactured.
  + This got us thinking about other useful Java features that we had learned in lectures that could be used in our program.
  + Person A determined that we would need some way to enforce the 8-digit hexadecimal naming scheme described in the brief such that only String values conforming to the standard were allowed.
  + Person B suggested it may be possible to use the regex search pattern for input validation. When giving each beanbag a unique identifier, we would revisit these discussions in later stages of development.

# 27/02/2020 (Thursday - 1h 20m: 12:40pm to 2pm)

* Further to the discussion above regarding the reuse of code, Person A noted that it may be possible to override the setter method for addBeanBags for reuse when selling bean bags.
* When we get to this part of the project, we will evaluate whether this is the case or if there is a better solution.
* Person B realised that we had missed declaration of two parameters (namely reservationNumber and filename) used to identify reserved sales and store the contents of BeanBagStore respectively.

# 29/02/2020 (Saturday - 2h: 1pm to 3pm)

* With the separate objects for stock and reservations, Person A remembered that you would need a way to store the details relating to reserved items.
* This could be done in one of two ways.
  + - The first would be to include an optional reserved field in the stock object which would require yet another constructor method with space for this field.
      * This particular method would only be invoked whenever any reservations are made.
      * This reserved variable would only need to be set when the Boolean value is true.
      * By including this addition method and relying on the same stock objects as before we could use the same setter methods from the Stock class to make changes to reservations instead of having to include duplicate methods within the Reservations class.
    - The alternative is to copy over all the details relating to the item for use in the reserved object.
      * This would reduce the need for an additional constructor method and would not affect storage required, as you remove the stock object from the ObjectArrayList and replace it with a reserved object with identical details.
* Person B noted that we had not even begun to incorporate any functionality into our program and we could possibly use four different implementations for our data structure.
  + As such we decided to keep things as simple as possible for the time being whilst still allowing ourselves to make any necessary changes to our design in the future, should we decide that our chosen structure was no longer suitable.
  + Our UML diagram was significantly useful to us in allowing us to plan ahead, making sure we would not later have to compromise on functionality, whilst ensuring we did not encounter the headache of needing to rewrite large portions of code at later stages of development.

# 07/03/2020 (Saturday - 4h: 6:30pm to 12:30am)

* Both agreed to move the log document from Microsoft Word on OneDrive to Google Docs on Drive to make collaboration easier.
* Person B reformatted the existing log for easier readability, making it easier to track previous progress. Changes included bullet-pointing of progress made, and simpler phrasing.

# 09/03/2020 (Monday - 5h: 11:30am to 4:30pm)

* Went over existing code to confirm that both people had a good understanding about what was going on.
* We created a driver file, StoreDriverApp, to test the front end of the application.
* Person A worked on implementing code for the mismatch exception in the Mismatch file, with person B acting as the observer to check that the code made sense syntactically and logically.
  + We swapped after the working code for Mismatch was completed and tested to work; person B commented on the code, whilst person A observed to ensure that both had a good understanding of the code.
* Person B started adding comments to CheckID, with person A observing to ensure that the comments made sense and were correct.
  + We researched into Java commenting conventions so that we knew we were using commonly accepted standards, which would make it easier for developers to continue working on the code in the future.
* Person A started adding code for the addBeanBags and setBeanBagPrice methods in Store, with person B acting as the observer.
  + We swapped after this was finished; person B added comments to Store, with person A acting the observer.
* Person B added comments to BeanBag, with person A observing to clarify any confusion about parts of the code.
* Person A completed the code for the sellBeanBags method in Store, with person B observing.
  + Once this was completed, we swapped roles to test it. Person B added code to the StoreDriverApp, whilst person A observed to ensure his testing methodology was correct.

# 10/03/2020 (Tuesday - 4h: 12:30pm to 4:30pm)

* Person A implemented the reserveBeanBags and unreserveBeanBags methods in Store, with person B being the observer.
  + We decided to create a new class, Reservation, to handle the reservations of items using a new array.
  + We swapped roles during the commenting phase; Person B did the commenting whilst person A observed.
* Person B moved all of the exception handling to the Checks class (previously named CheckID), whereas it was previously handled mostly in CheckID, but not entirely. Person A observed to ensure this was done correctly.
  + Mismatch was removed, as all of the code was moved to Checks.
* Person B started implementing the empty method in Store, with person A acting as the observer.
  + We were initially unsuccessful after starting off well, so we swapped roles and successfully implemented it. For this, person A was the driver, and person B was the observer.
* Whilst we were optimising some code, person A suggested changing the bean bag ID checks to be non-case-sensitive, as this would improve the user experience. Person B implemented this after researching how this could be done.
* Person A implemented the replace method from Store, with person B acting as the observer.
  + We swapped roles for the commenting of this. Person B acted as the driver, and person A acted as the observer.
* Person B realised that the Stock class had not been entirely commented on, so they worked on that, with person A observing to ensure comments made sense.
* Person B implemented sellBeanBags for a reservation (selling bean bags which had been previously reserved by the customer), with person A acting as the observer.
  + Person A was the driver for commenting, and person B was the observer
* Person B implemented the getNumberOfSoldBeanBags and getNumberOfSoldBeanBags(id) methods for Store, with person A observing.
  + We were able to reuse most of the code from beanBagsInStock(id), changing some of the parts to be relevant for this use case.
* Person A implemented getTotalPriceOfReservedBeanBags in Store, with person B observing.
  + We swapped roles for commenting this method; person B was the driver, whilst person A was the observer.
* Person B commented on the existing methods and some additional parts within methods in Store, as well as reordering some code for readability, with person A observing to ensure that a mutual understanding was retained.

# 11/03/2020 (Wednesday - 4h 30m: 1:00pm to 5:30pm)

* Person A updated the reserveBeanBags method and replace method in Store to be fully functional.
  + Person B swapped with person A for the commenting of the new code added to the reserveBeanBags method and replace method.
* Person A implemented the getNumberofDifferentBeanBagsInStock method in Store, with person B observing.
  + Person B swapped with person A for the commenting of this method to ensure a mutual understanding.
* Person B added the getBeanBagDetails method in Store, with person A observing.
  + We swapped roles for the commenting of this code.
* We improved the addBeanBags method by calling a new method, getExistingPrice, within it. Person B acted as the driver, whilst person A was the observer.
  + This ensures consistency in the stock list, as bean bags with the same ID will have the same price when they’re added.
  + It complements the method setBeanBagPrice, as it means bean bags added in the future will also have this price set.
* Person A added additional testing methods, toString, array, and printArray, in Store. Person B acted as the observer.
  + These were made to help us test the application.
  + We swapped roles for commenting; person B acted as the driver, whilst person A was the observer.

# 12/03/2020 (Thursday - 4h: 10:30pm to 2:30pm)

* Person A implemented the saveStoreContents and loadStoreContents methods in Store, with person B observing.
  + We swapped roles for commenting this method to ensure both parties understood the code fully.
* We removed duplicated code by moving it to a new method, countBeanBags. This counts the number of matching IDs for any objects in the given list(s).
  + This new method was called in beanBagsInStock and getNumberOfSoldBeanBags, where an ID is passed as an argument.
    - It replaced the blocks of code which previously made up these methods.
* We decided to remove the toString method, as it wasn’t being used in any parts of our program.
* Person B updated the setBeanBagPrice method to sell reserved bean bags to the customer for the lowest price since they reserved it (in the case that they reserved it, and it changed prices since then to be lower). Person A was the observer for this.
  + We swapped after the code was finished, so person A was the driver for commenting, whilst person B observed.
* We tested our app by creating a test driver file, Driver.
  + To ensure that our program was appropriately tested, we isolated the cases one by one (made easier with our alphabetical ordering of methods), and tested different combinations.

# 13/03/2020 (Friday - 4h 30m: 12am to 4:30am)

* Person B created doc comments for the classes and methods within those classes, with person A acting as the observer to ensure that mistakes weren’t being made.
  + We used @inheritDoc for the overridden methods, as the doc comments were already provided for those.
  + We created doc comments for new methods which we had created.
* Person A added assertions to help with final testing before submission, with person B observing to ensure that the assertions made sense.
* We created the jar file for submission.