Question #2

Part B

In the case of at()

We first of all try to access the value with the passed index. Then, if there is an out_of_range error thrown, then we catch it. In the case that we do catch one, we will return the default value that was registered when we created the FlexArray along the RobustArray.

In the case of assign()

Like for at(), we first of all try to assign with the passed index. If the function returns an out_of_range error we recover by doing two possible things. First of all, if the index is negative, for example -1 or -10, we change the index for 0. Then, if the index value is beyond the size of our current array, we change it for the value size - 1, i.e.: the last possible element of the array.

In the case of resize()

First of all, we add to create a class that inherits from std::bad_alloc, which will be used as our exception derived from std::bad_alloc which stores the value of newSize that was used in the most recent attempt to resizeRobustArray.

So in the function itself, like all the other ones, we first of all try to resize with the passed index.

In the case that it does not work, we try for the first time to resize with one less than the passed newSize. If it does not work, we try with 2 less. Then, if it still does not work, we return a new exception called “derived_exception”, which is a class that inherits from std::bad_alloc.

Question #3

Part A

For this question, our problem is related to a car collection. The situation is simple: we want to be able to iterate through our collection of cars. The implemented collection can simply add some cars for the moment, that is it. The program already implements a basic collection for demonstration. The collection
keeps track of the brand, model, color, year, mileage, price and availability of each car. We would like to be able to iterate through the collection in terms of: brand, mileage, price, year and the whole collection. We will therefore use the iterator pattern to solve our problem. Here is the UML diagram of the current situation.
**Part B**

The following is the UML with the added iterator pattern. Please note that the new elements are in bold/shadow, and the old ones have less borders and no shadow.
So we will be passing to the iterator what we specifically want to iterate over. The collection will be responsible for creating the iterator. Some of the options will be for example, iterate over the whole collection, iterate over only a brand and many more.