A four-stage object-oriented design methodology was described. Brainstorming is the process of coming up with a possible set of object classes in the problem solution. Filtering is the process of reexamining the tentative classes, eliminating those that are not appropriate, combining some classes, and creating additional classes if necessary. Scenarios is the part of the process in which you examine the responsibilities of each proposed class and role play to see if all situations are covered. Responsibility algorithms is the phase where algorithms are derived to carry out the responsibilities. CRC cards are used as a visual means of recording classes and their responsibilities. In contrast, UML diagrams are used to document a class and show the data fields and their types. This methodology was applied to the Case Study.

Exercises

1. The Sorted List ADT is to be extended with a Boolean member function, `IsThere`, which takes as a parameter an item of type `ItemType` and determines whether there is an element with this key in the list.
   a. Write the specifications for this function.
   b. Write the prototype for this function.
   c. Write the array-based function definition using the binary search algorithm.
   d. Describe this function in terms of Big-O.
2. Redo Exercise 1(c) using a linked implementation.
3. Rather than enhancing the Sorted List ADTs by adding a member function `IsThere`, you decide to write a client function to do the same task.
   a. Write the specifications for this function.
   b. Write the function definition.
   c. Were you able to use the binary search algorithm? Explain your answer.
   d. Describe this function in terms of Big-O.
   e. Write a paragraph comparing the client function and the member function for the same task.
4. Write a client function that merges two instances of the Sorted List ADT using the following specification.

MergeLists(SortedType list1, SortedType list2, SortedType& result)

Function:	Merge two sorted lists into a third sorted list.
Preconditions:	list1 and list2 have been initialized and are sorted by key using function ComparedTo.
	list1 and list2 do not have any keys in common.
Postconditions:	result is a sorted list that contains all of the items from list1 and list2.

a. Write the prototype for `MergeLists`.
b. Write the function definition, using an array-based implementation.
c. Write the function definition, using a linked implementation.
d. Describe the algorithm in terms of Big-O

5. Rewrite Exercise 4, making `MergeLists` an array-based member function of the Sorted List ADT.
6. Rewrite Exercise 5, making `MergeLists` a linked member function of the Sorted List ADT.
7. The specifications for the Sorted List ADT state that the item to be deleted is in the list.
   a. Rewrite the specification for `DeleteItem` so that the list is unchanged if the item to be deleted is not in the list.
   b. Implement `DeleteItem` as specified in (a) using an array-based implementation.
   c. Implement `DeleteItem` as specified in (a) using a linked implementation.
   d. Rewrite the specification for `DeleteItem` so that all copies of the item to be deleted are removed if they exist.
   e. Implement `DeleteItem` as specified in (d) using an array-based implementation.
   f. Implement `DeleteItem` as specified in (d) using a linked implementation.
8. A Sorted List ADT is to be extended by the addition of function `SplitLists`, which has the following specifications:

SplitLists(SortedType list, ItemType item, SortedType& list1, SortedType& list2)

Function:	Divides list into two lists according to the key of item.
Preconditions:	list has been initialized and is not empty.
Postconditions:	list1 contains all the items of list whose keys are less than or equal to item's key; list2 contains all the items of list whose keys are greater than item's key.

a. Implement `SplitLists` as a member function of the array-based Sorted List ADT.
b. Implement `SplitLists` as a member function of the linked Sorted List ADT.
c. Compare the algorithms used in (a) and (b).

d. Implement `SplitLists` as a client function of the array-based Sorted List ADT.

e. Implement `SplitLists` as a client function of the linked Sorted List ADT.

9. A Sorted List ADT is to be extended by the addition of a member function `Head`, which has the following precondition and postcondition:

   Precondition: list has been initialized and is not empty.

   Postcondition: return value is the last item inserted in the list.

   a. Will this addition be easy to implement in the array-based `SortedType`? Explain.

   b. Will this addition be easy to implement in linked `SortedType`? Explain.

10. A List ADT is to be extended by the addition of function `Tail`, which has the following precondition and postcondition:

    Precondition: list has been initialized and is not empty.

    Postcondition: return value is a new list without the most recently inserted item.

    a. Will this addition be easy to implement in the array-based `SortedType`? Explain.

    b. Will this addition be easy to implement in linked `SortedType`? Explain.

11. a. Change the specifications for the Sorted List ADT so that `InsertItem` throws an exception if the list is full.

    b. Implement the revised specifications in (a) using an array-based implementation.

    c. Implement the revised specifications in (a) using a linked implementation.

12. Write a class based on class `UnsortedType` as a bounded linked implementation. Provide a parameterized constructor that takes the maximum number of items as a parameter. If function `InsertItem` is called when the list if full, throw an exception.

13. Write a class based on class `SortedType` as a bounded linked implementation. Provide a parameterized constructor that takes the maximum number of items as a parameter. If function `InsertItem` is called when the list if full, throw an exception.

14. Write a class based on class `UnsortedType` as an unbounded array-based implementation. If the dynamically allocated array is full, create an array double the size and move the elements into it.

15. Write a class based on class `SortedType` as an unbounded array-based implementation. If the dynamically allocated array is full, create an array double the size and move the elements into it.

16. Write a formal specification for the problem explored in the Case Study.

17. Design and implement a class that represents a name (`NameType`).

18. Rewrite the Case Study using class `NameType`.

19. Design the UML diagrams for the revised class `HouseType` and class `NameType`.