ICOM 4035 – Data Structures

[Lecture 1 2](#_Toc50185053)

[ Introduction 2](#_Toc50185054)

[ Abstract Data Type 2](#_Toc50185055)

[ Ingredients 2](#_Toc50185056)

[ Data Structure 2](#_Toc50185057)

[ Ingredients 2](#_Toc50185058)

[ Vector Class Creation 2](#_Toc50185059)

[ Vector Class Operations 2](#_Toc50185060)

[ Process to write code 2](#_Toc50185061)

[Lecture 2 3](#_Toc50185062)

[ Multiple implementations for ADT 3](#_Toc50185063)

[ Using Interfaces to specify ADT 3](#_Toc50185064)

[ Using Abstract Factories to create instances 3](#_Toc50185065)

[ Benefits 3](#_Toc50185066)

[Lecture 3 4](#_Toc50185067)

[ Role of collection classes and Bags 4](#_Toc50185068)

[ Collection 4](#_Toc50185069)

[ Bag 4](#_Toc50185070)

[ Design and implementation of the Bag ADT 4](#_Toc50185071)

[ Operations of bag 4](#_Toc50185072)

[ Iterating over a Bag 5](#_Toc50185073)

[ Implementing a Dynamic Bag 5](#_Toc50185074)

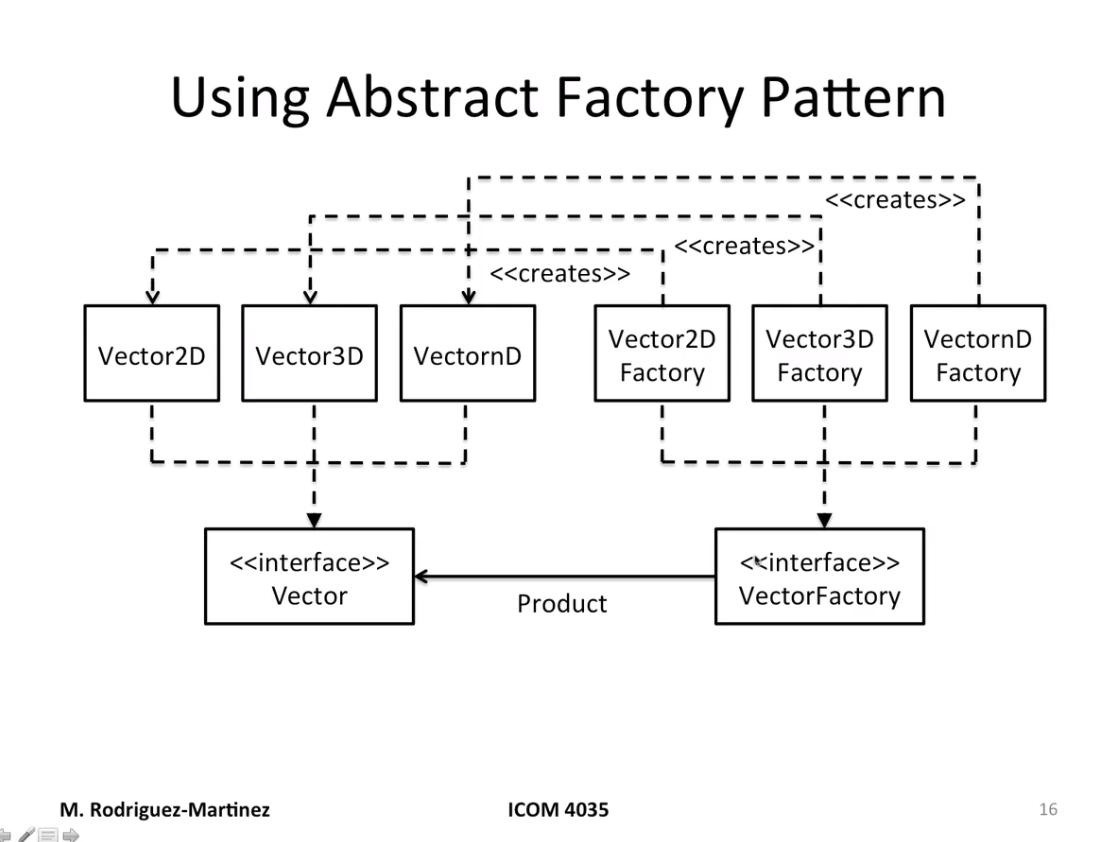
[ Notes 5](#_Toc50185075)

# Lecture 1

* Introduction
  + Abstract Data Type (ADT) = Mathematical Model that represents data
    - Ingredients
      * Values = data to be stored
        + Represents current state in model
      * Operations = actions that will be taken
        + Modifies the current state in model
        + Produces new information based on these values
  + Data Structure (DS) = Implementation of ADT using constructs of programming language
    - Ingredients
      * Variables and data types – they provide storage for ADT values
      * Methods – functions that will implement instructions associated with operations for each ADT
    - 1 ADT can be implemented with multiple types of Data Structures
  + Example: 2-D Vector ADT
    - Values
      * X coordinate (real number)
      * Y coordinate (real number)
    - Operations
      * Creation – build a new vector at (a,b)
      * Mutator – (re-assign vector coordinates)
        + However, it’s better practice to keep the values immutable to prevent unexpected behavior
      * Accessor – (inspects the values of coordinates)
      * Externalizer – (convert to a string of characters)
      * Mathematical Operations
        + Magnitude
        + Sum
        + Substract
        + Dot Product
* Vector Class Creation
  + Values of implementation should be private since accessing and changing them should be the accessor’s and mutator’s job
* Vector Class Operations
  + Process to write code
    - Define ADT values and operations
    - Pick Data Structure to implement ADT
      * Chose basic types or other data structures to store values
      * Write member methods to implement operations
    - Revise, test, revise, test

# Lecture 2

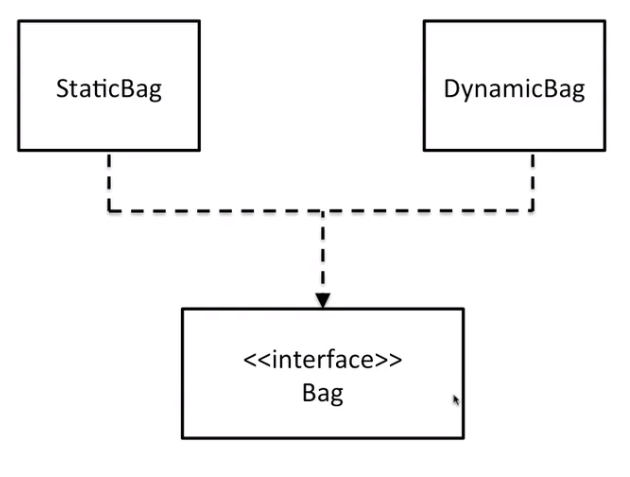
* Multiple implementations for ADT
  + Even though we implemented a 2D vector class, we cannot use 3D vectors or N dimensional vectors
    - This shows how an ADT can be implemented multiple ways
* Using Interfaces to specify ADT
  + An interface sort of acts like a prototype that includes all the operations and values a specific implementation will have. However, it doesn’t actually do anything
  + Classes will then implement the interfaces. Therefore, the classes will store the values and perform operations, adding its own peculiarities, restrictions, and methods.
    - 2D vector hast *getX* and *getY* methods, but the 3D vector additionally implements *getZ*
    - The *getCoordinate* method is hardcoded for 2 and 3 dimensions, but not for N dimensions
    - If trying to make operations by mixing types, an error will be thrown
* Using Abstract Factories to create instances
  + Can be used to control object creation
    - Instead of calling constructor in the code, the Abstract Factory class is in charge of making the new instance of the object
    - We don’t necessarily know which concrete class is used to implement the interface



* + Benefits
    - You can have as many vector implementations as I need
    - Main program might not need to be changed
      * Factory class may read out of a configuration file form command line or properties file
    - Code is based on the Vector interface (only one concept to deal with)
  + Is basically an interface with a lonely method *newInstance* that validates input values for the factory and then returns new object instance if everything turned out alright

# Lecture 3

* Role of collection classes and Bags
  + Collection = Abstract Data Type whose purpose is to store data items for us
    - They should keep object instances organized and ready to use
  + Benefits
    - Variable size
      * Size can be dynamically changed without need of recompiling
    - Implementation can be changed
      * Arrays are not necessarily required. Maybe Linked Lists, Hash Tables, etc. can be used
    - Information hiding
      * Controls access to items without exposing specific details that might reveal vulnerabilities
    - Reuse (Don’t Repeat Yourself | DRY)
      * No need to make the same allocation or same operations to find things around. The collection will make sure to access information in an efficient way with the same pattern
  + Bag
    - **Unordered** collection of things
    - Repetitions are allowed
    - Mathematical term: Multi-set
    - They can store whatever data item the application needs
    - Static Bag
      * Maximum size of bag is fixed at build-time
    - Dynamic Bag
      * Maximum size of bag can grow to accommodate elements
    - How do we implement it?
      * Array of type Object
      * ADT Bag is an interface
      * Static Bag or Dynamic bags will implement the Bag interface
* Design and implementation of the Bag ADT



* + Operations of bag (9)
    - Add new element (add)
      * For static bag, we need to check if there is enough space
      * Dynamic bags grow as needed
    - Remove **one** copy of an element (erase)
      * Return true if erased or false if not found
      * If found, remove that specific copy and swap it with the last element (so the currentSize index points to the end of the array again)
        + Making swap is not a problem since the multi-set is not ordered
        + The previously last element should become null to prevent memory leaks
    - Remove **all** copies of an element (eraseAll)
      * for(int result = 0; multiset.erase(value); result++); return(result);
    - Count copies of element(count)
      * Scan array and accumulate each time you see target value
        + In Java the *equals* method must be used to compare objects
    - Clear bag (clear)
      * Loop through array, setting everything to null and setting currentSize to 0
    - Test for element membership (isMember)
      * return(multiset.count(value) > 0);
    - Get Bag size (size)
      * return(currentSize);
    - Test if empty(isEmpty)
      * return(multiset.size == 0);
    - Iterate over all stored values
* Iterating over a Bag
  + In order to iterate over a bag, we can make the Bag an iterable
    - The Bag interface will extend the Java Iterable interaface
    - Bag can then return an iterator
* Implementing a Dynamic Bag
  + Using ArrayList / C++ Vector logic (duplicate size once you run out)
  + Possibly using Linked List
  + Alternatives
    - Copy all methods from static bag, but change the add method
      * NO!
      * Has a lot of space for bugs than can cross over from the static bag
    - Use inheritance to make static bag a parent class for dynamic bag and change add method
      * Better, but you need to make array of elements and currentSize protected instead of private
      * Always aim to have private fields in the class
    - Object composition
      * Static bag is private object instance of dynamic bag, which maps every method to the static’s bag except the add method, which makes a new bag and overwrites the original static bag, making it twice the original size
* Notes
  + References that are unused MUST be set to null
    - Otherwise unused references will stay there without the garbage collector knowing what to do with it
  + It’s always good practice to write operations in terms of previous ones
    - If there’s a bug somewhere, it’s localized in only one place
  + Factories
    - The factory interface will have 1 method called *newInstance* which takes in any important parameters like maxSize in the case of Bags
    - The classes that implement this interface will simply return a new instance of the specific object the factory was made for