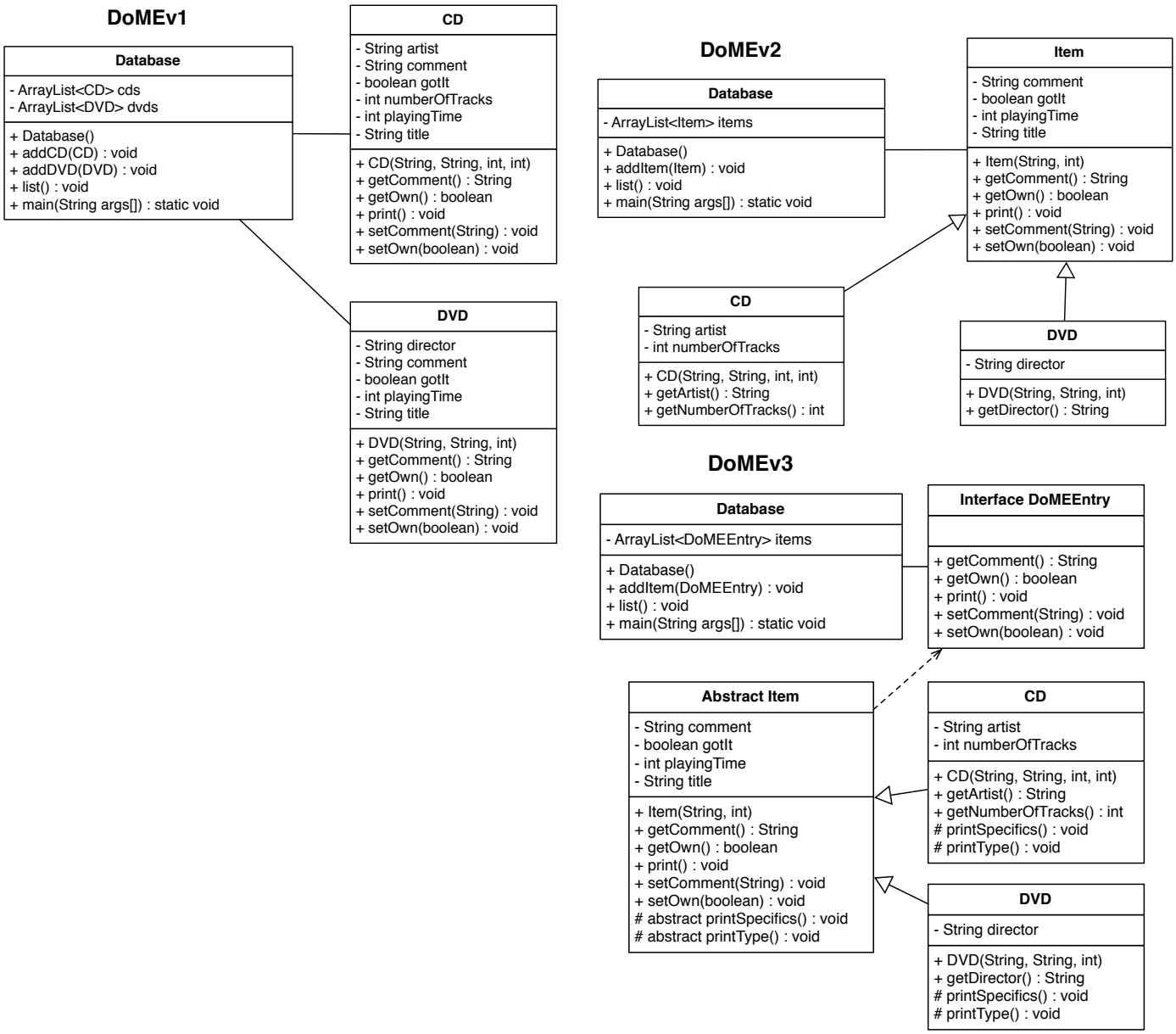
Dr. Emily Hill
CMPT 184

DOME HW

In this HW, you will begin to understand the difficulty of defining and implementing abstract classes & interfaces. Starting from the DoMEv1 project posted on BB, update the application to use abstract classes & interfaces to remove code duplication and make adding new types of media easier. We will be going over how to convert DoMEv1 to versions 2 & 3 in class.



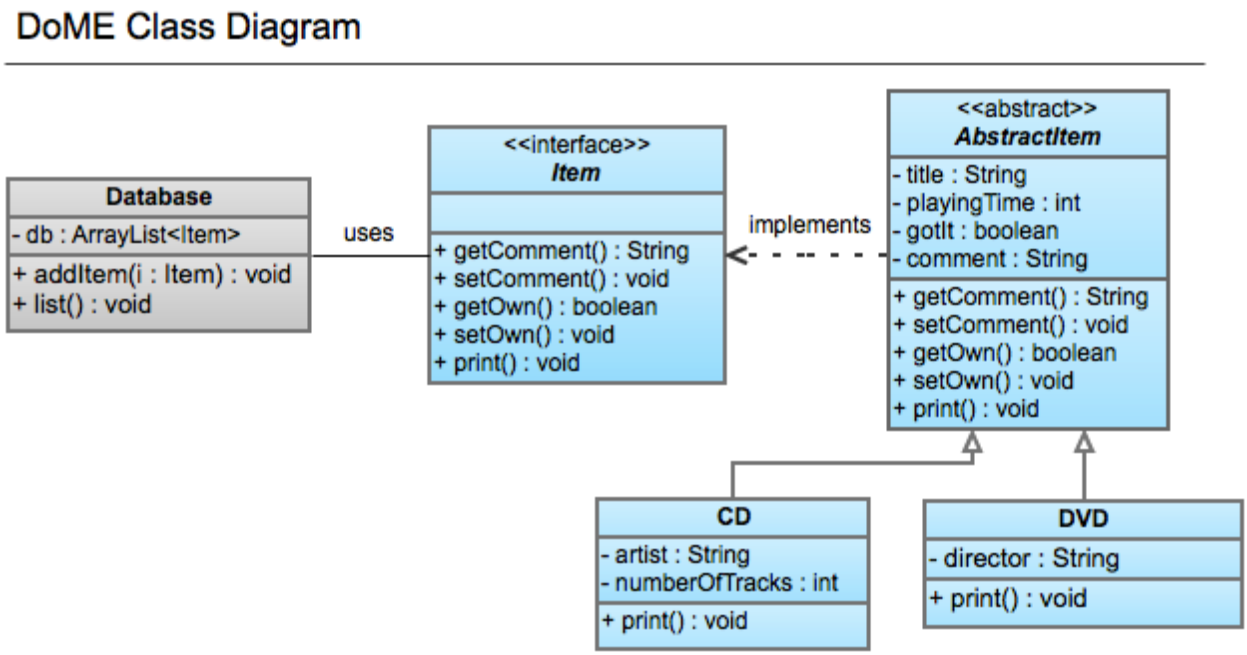
What if we wanted to add support for Video & Board Games to our DoME application? What types of information should we include in our DB for each of these? Is there any information that they share? Is it possible to add support for these types without changing the Database code (other than main)? How would you go about adding support for these new media types?

Draw a new class diagram (DoMEv4) for adding Video & Board Games & update your implementation to support these new types. Be sure to test them in main.

Part 1: Draw your final design for DoMEv4

Draw your class design diagram using a UML-styled notation like we've been doing in class. For more information on UML see http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/, although this is not necessary to complete the project.

Class design diagrams are easiest to follow in graphical format. There are many free drawing programs you could use (see Blackboard for links), as well as UML drawing tools. For example, I used gliffy (http://www.gliffy.com/uml-software/) to create a class diagram for DoMEv3:



(You can either take a screen shot or sign up for a free account and save it as a png.)

Each class is represented by a box with 3 sections separated by 2 lines: the class name, its fields, and its methods. Interfaces obviously have no fields, but still need an empty field section. Concrete classes are differentiated from abstract classes and interfaces using <<abstract>> and <<interface>> notations above the class name. Note that private is indicated with minus (-), public with plus (+), and protected with pound (#). After each field, parameter, and method name is a colon (:) followed by the type or return type. Inheritance is indicated using solid arrows, interface implementation indicated with dashed arrows. A solid line shows that the Database class uses the Item interface.

You may also submit your class design in textual form, as long as the extends and implements relationships are formalized and the difference between fields and methods are made clear:

class **Database**
fields
- db : ArrayList<Item>

methods
+ addItem(i : Item) : void
+ list() : void

class **CD** extends ***AbstractItem***
fields
- artist : String
- numberOfTracks : int

methods
+ print() : void

class **DVD** extends ***AbstractItem***
fields
- director : String

methods
+ print() : void

interface ***Item***
fields

methods
+ getComment() : String
+ setComment() : void
+ getOwn() : boolean
+ setOwn() : void
+ print() : void

abstract class ***AbstractItem*** implements ***Item***
fields
- title : String
- playingTime : int
- gotIt : boolean
- comment : String

methods
+ getComment() : String
+ setComment() : void
+ getOwn() : boolean
+ setOwn() : void
+ print() : void

Part 2: Implement the Design

I highly recommend you check-in your design with me before moving on to the implementation. You cannot get full credit for your implementation if your design is flawed.

Make sure to test your implementation in main in the Database class by adding at least one item of each type to the DB and making sure the printed results are correct.