Object Oriented Design and Analysis
CPE 372

Lecture 6

UML Class Diagrams

Dr. Sally E. Goldin
Department of Computer Engineering
King Mongkut's University of Technology Thonburi
Bangkok, Thailand

Exercise 5 Issues

Static versus Instance Methods

My solution includes only one static method (aside from main)

```
/**
 * static method to find and return the first shape
 * for which the passed point is inside the bounding box.
 * @param  X    X coordinate of selected point
 * @param  Y    Y coordinate of selected point
 * @return first shape found, or null if point is not in any shape.
 */
public static AbstractShape findSelectedShape(int X, int Y);
```

This method *must* be static because it needs to access the allFigures list, which is static and only visible to *AbstractClass* and its subclasses.

Avoid using static methods unless there is no alternative!

Often they will create dependencies that can lead to bugs

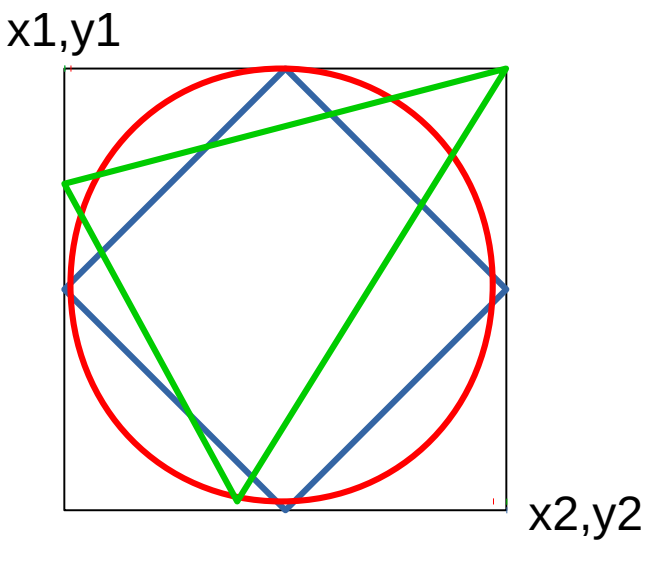
Many novice Java programmers use them because they are “easy”, but they can break encapsulation and reduce modularity.

Superclass versus Subclass Methods

Data and operations common to all or most subclasses should be implemented in the superclass

- Bounding box data items
- Method to determine if a point is in the bounding box

However, code to calculate and set the bounding box differs for each shape.



What class should implement the interface?

Possibilities:

DrawingCanvas

FigureViewer

ShapeFileTester

Choose the one that will best preserve encapsulation

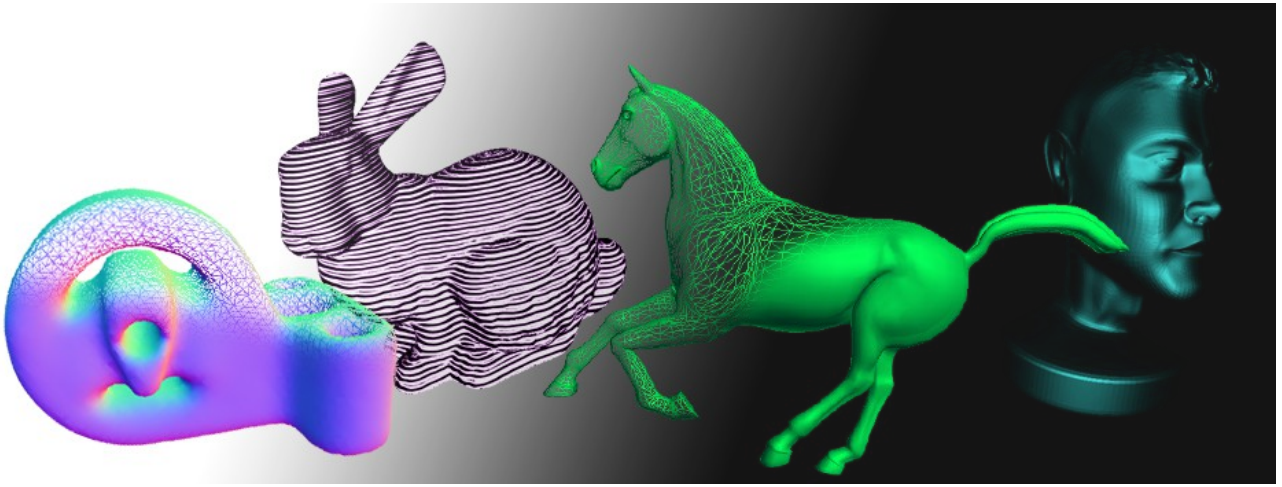
- If you put it in *ShapeFileTester*, how will you get the reference to that class for use in the addListener() call in *FigureViewer*?
- I put it into *FigureViewer* because *DrawingCanvas* could be used for drawing anything, not just *AbstractShape* objects.

Graphics Issues

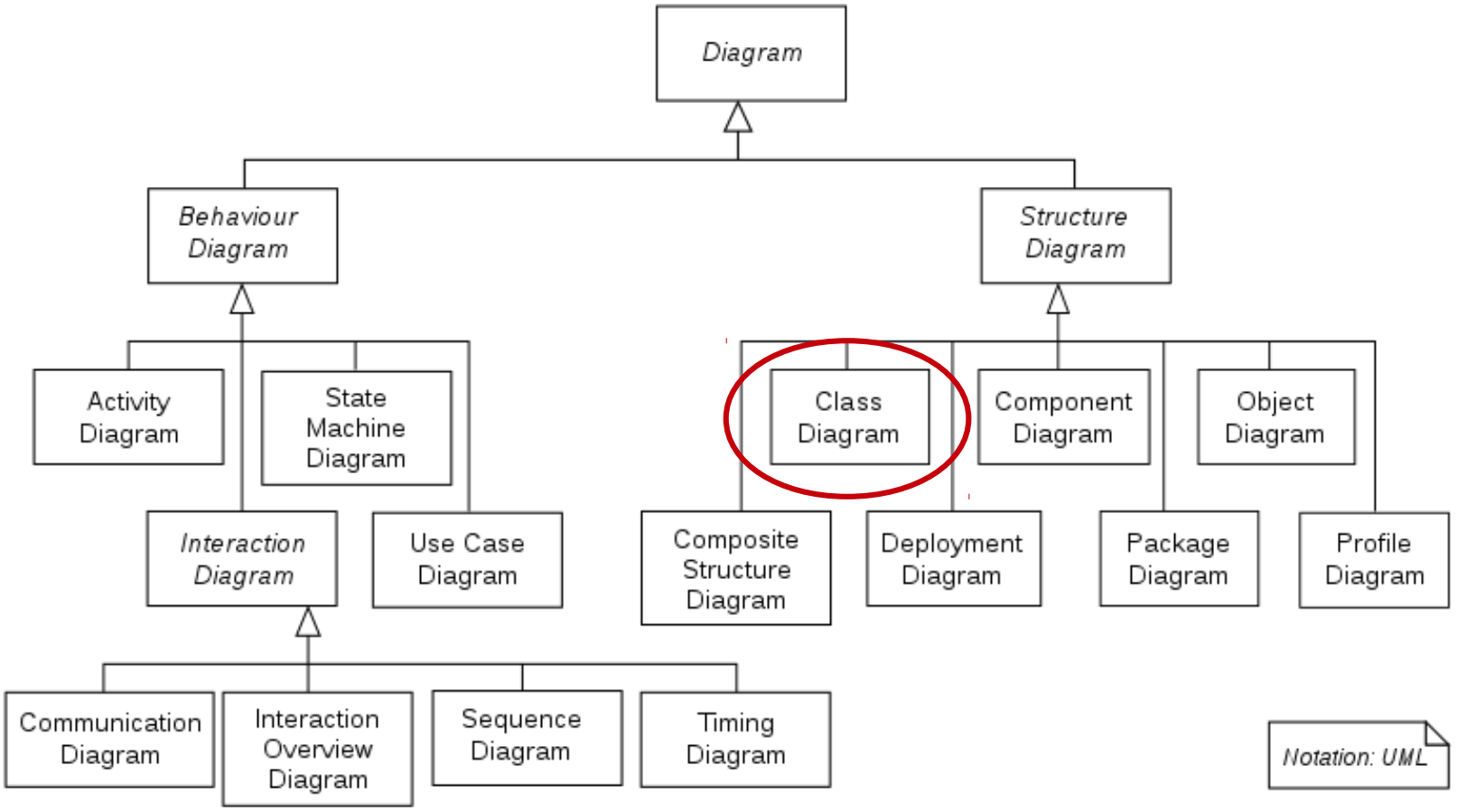
Graphics are always tricky!

Need to consider

- » Coordinate systems
- » Timing and concurrency (drawing is not immediate)
- » Event order is not predictable
- » UI is a containment hierarchy



UML Class Diagrams



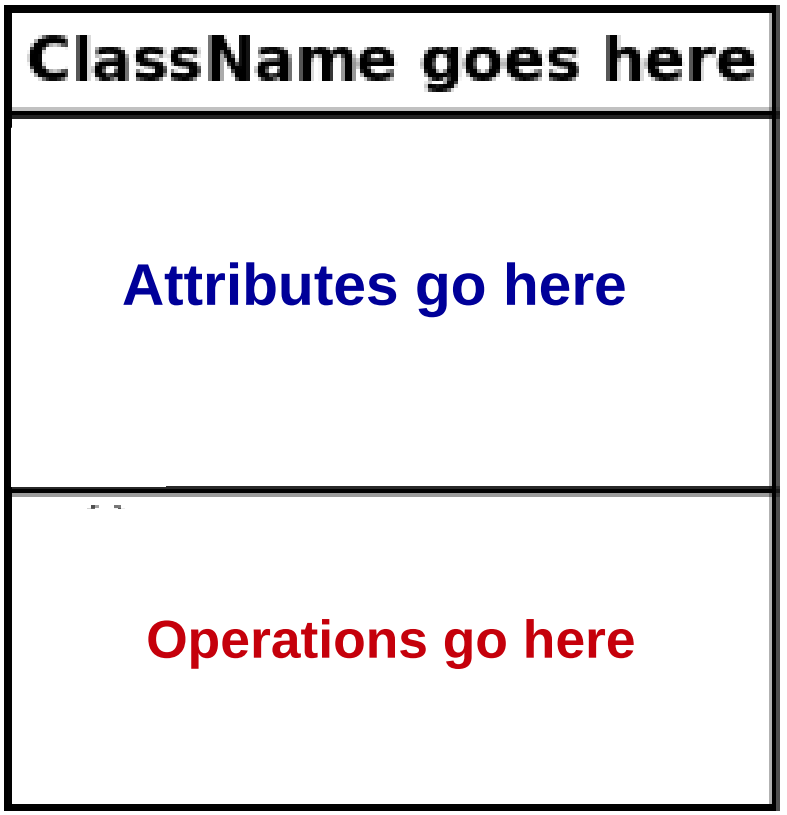
What is a class diagram?

"A **class diagram** describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected." Martin Fowler, *UML Distilled Third Edition (2004)*

Operations == methods

Properties (also called attributes) == members

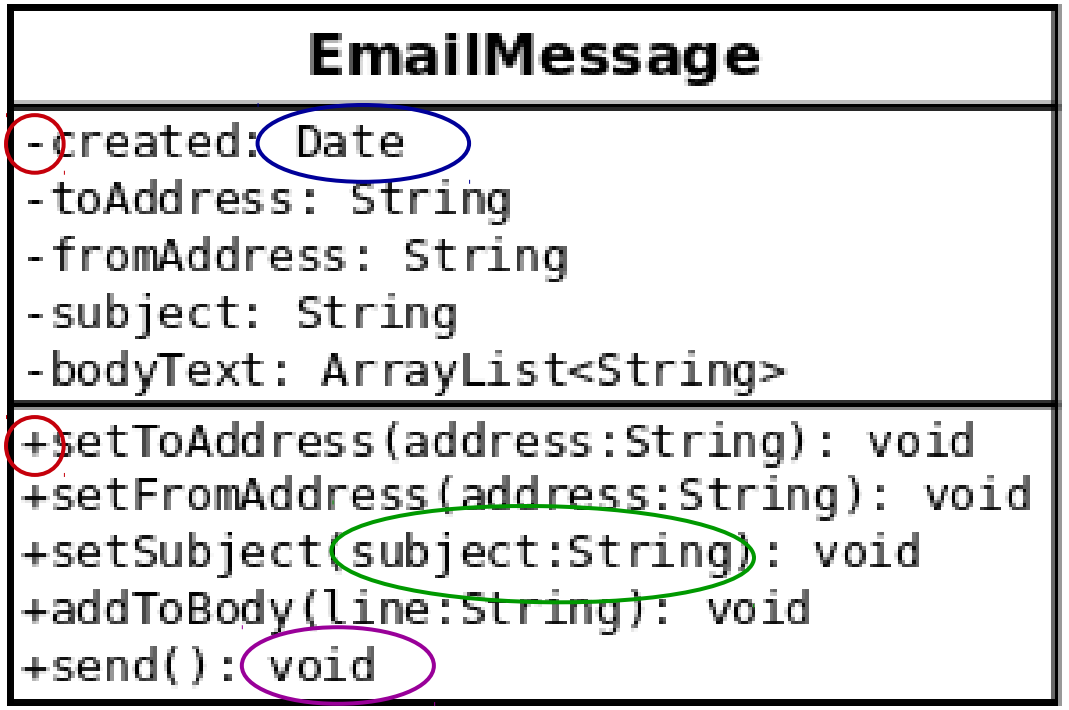
Symbol for a Class



EmailMessage Class

visibility

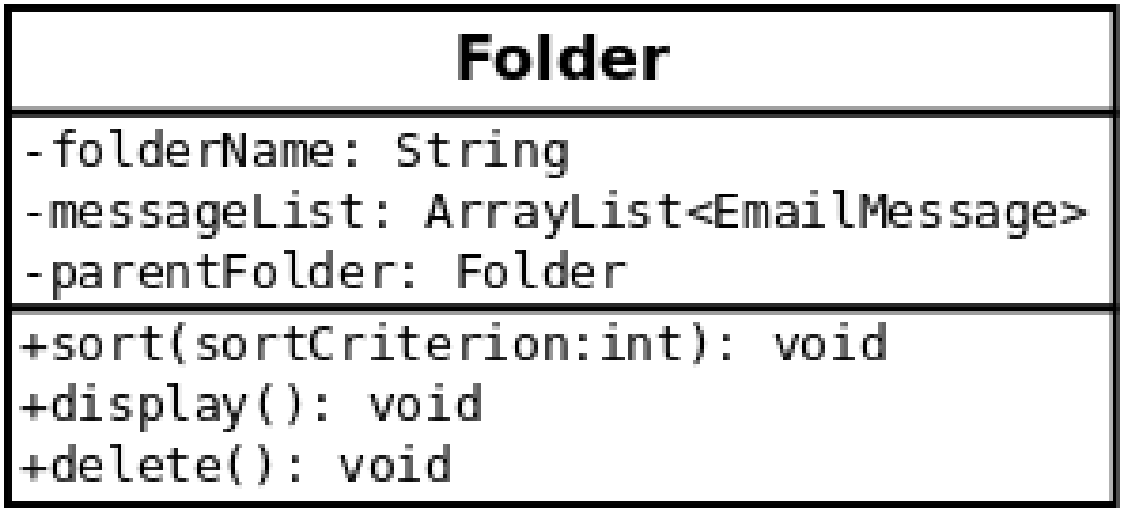
data type



method argument

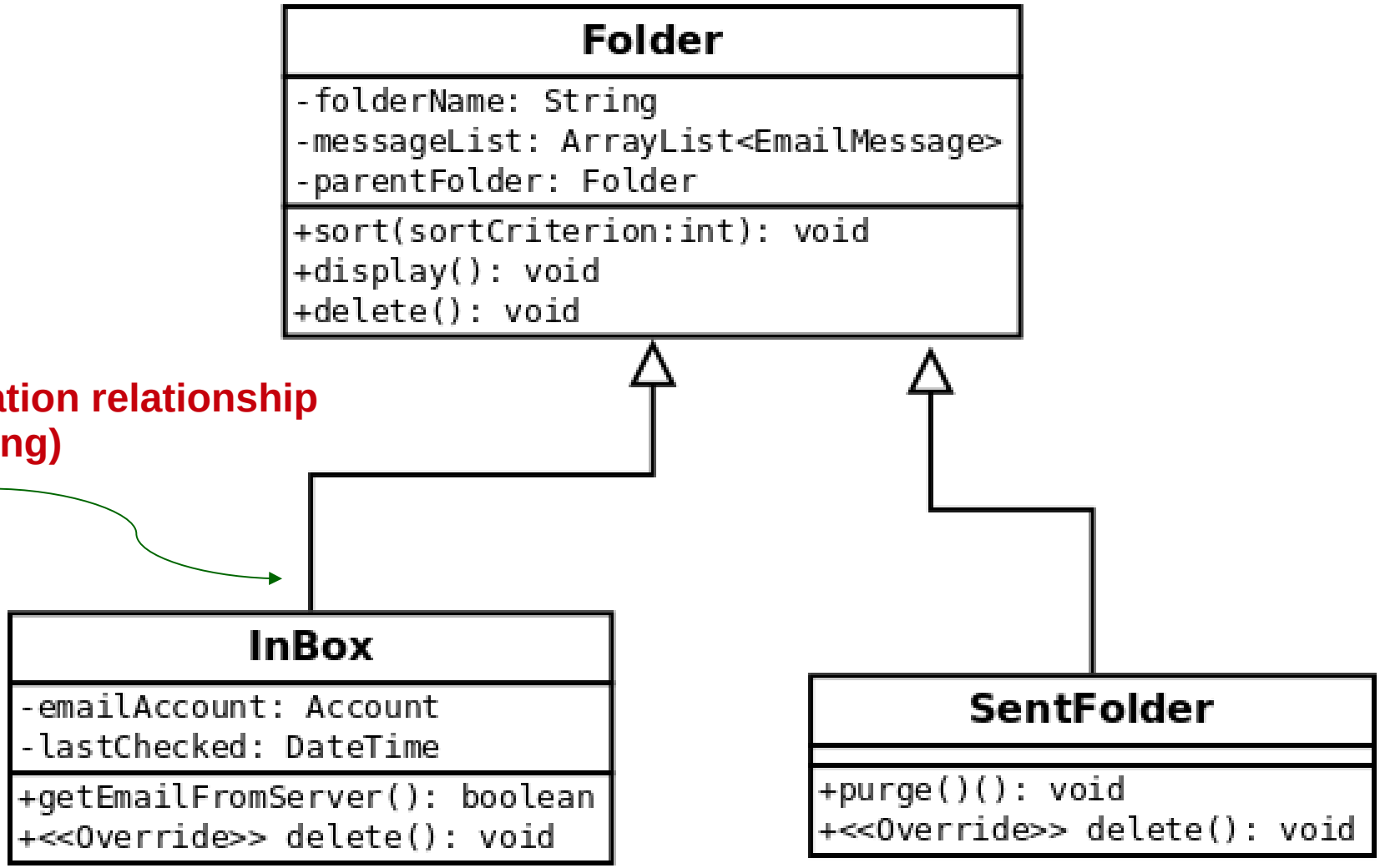
method return value data type

A *Folder* Class



- Represents a general folder where messages can be saved
- The parentFolder property allows for hierarchical (tree-structured) organization of folders
- Methods for displaying the folder contents, sorting the messages, deleting the folder, etc.

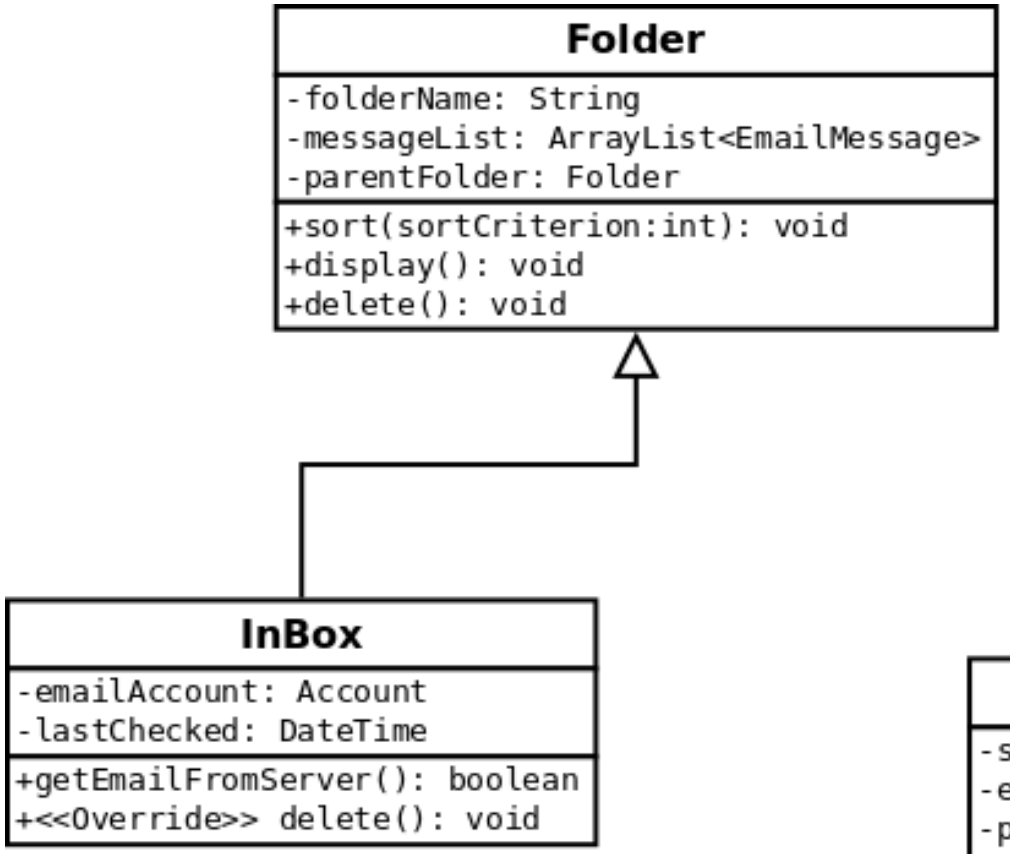
Special Folders



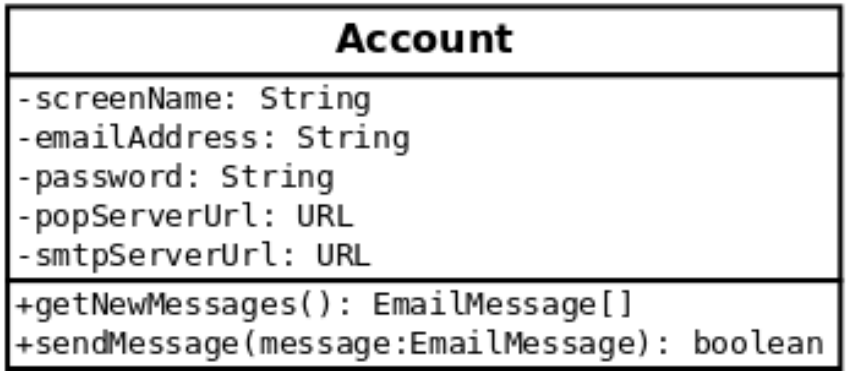
Generalization relationship (subclassing)

Override delete() method; should not be possible to delete these folders

One inbox for each email account

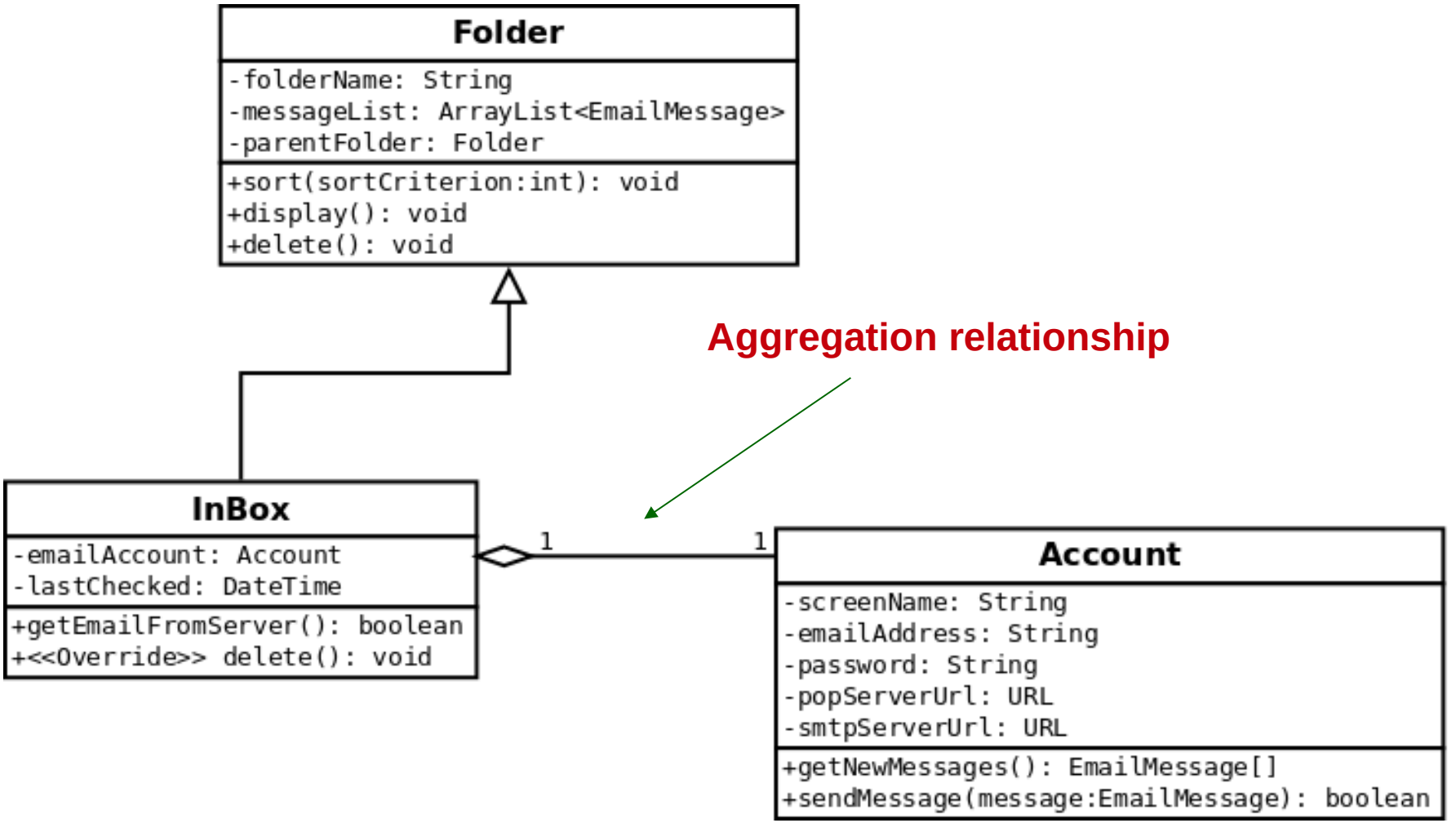


Account **class includes all the information necessary to send or receive email for a particular email address**

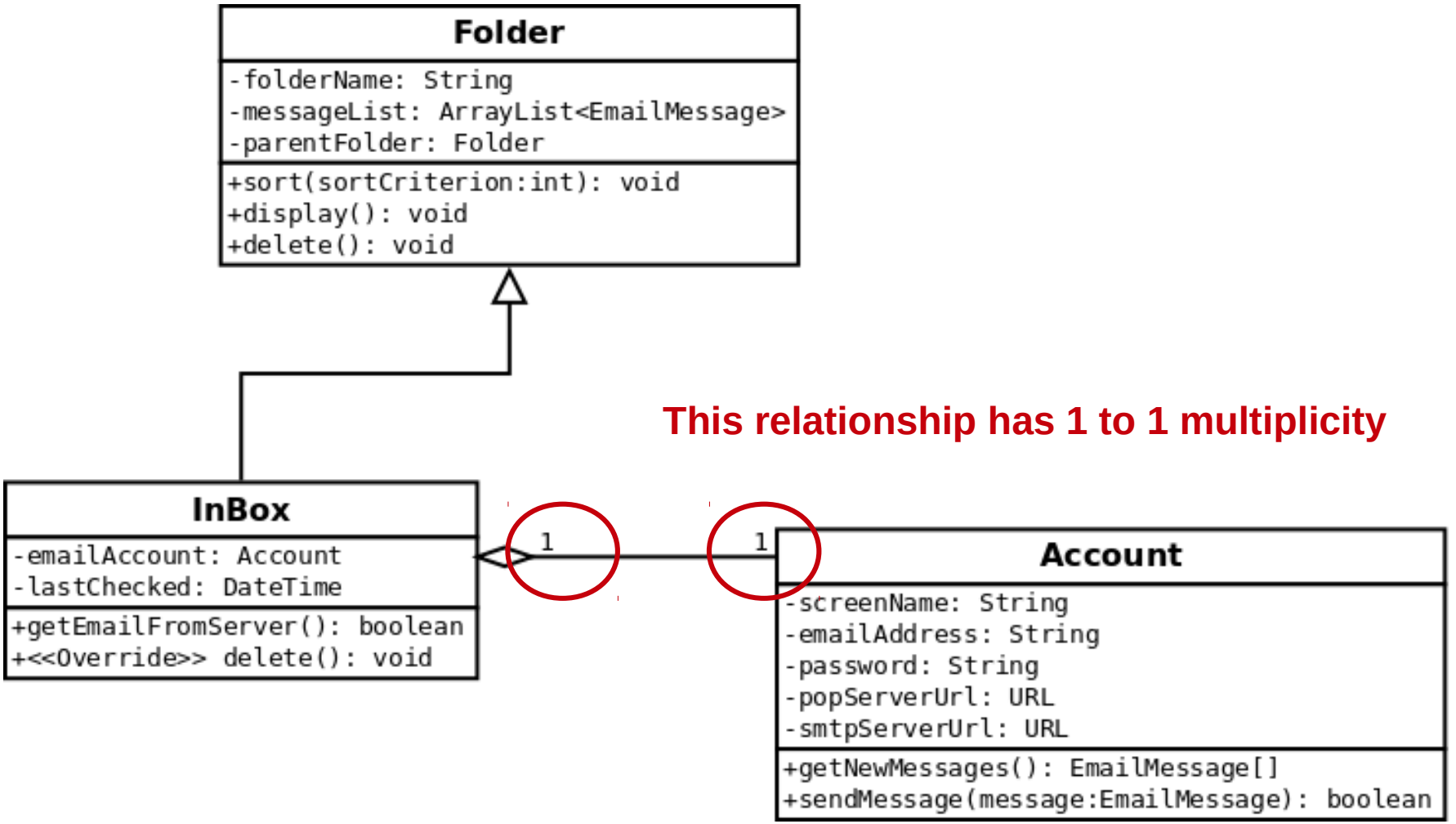


How can we show the relationship between *Account* and *InBox*?

Aggregation/Composition



Multiplicity

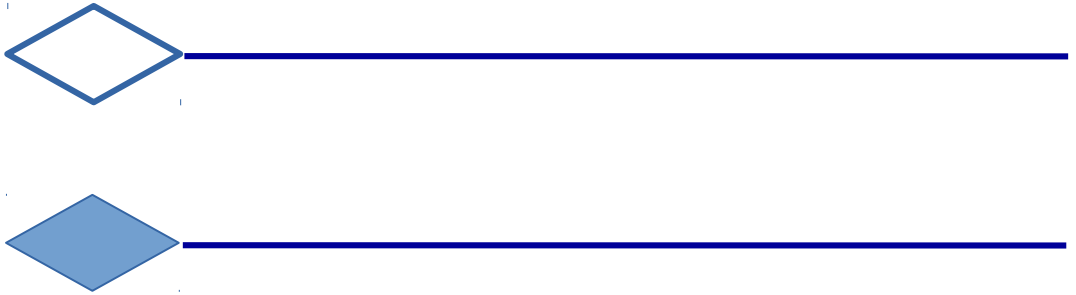


This relationship has 1 to 1 multiplicity

Each *InBox i*nstance must have exactly one associated *Account*
Each *Account* can be associated with only one *InBox*

Aggregation versus Composition

UML actually specifies two relationships



Aggregation

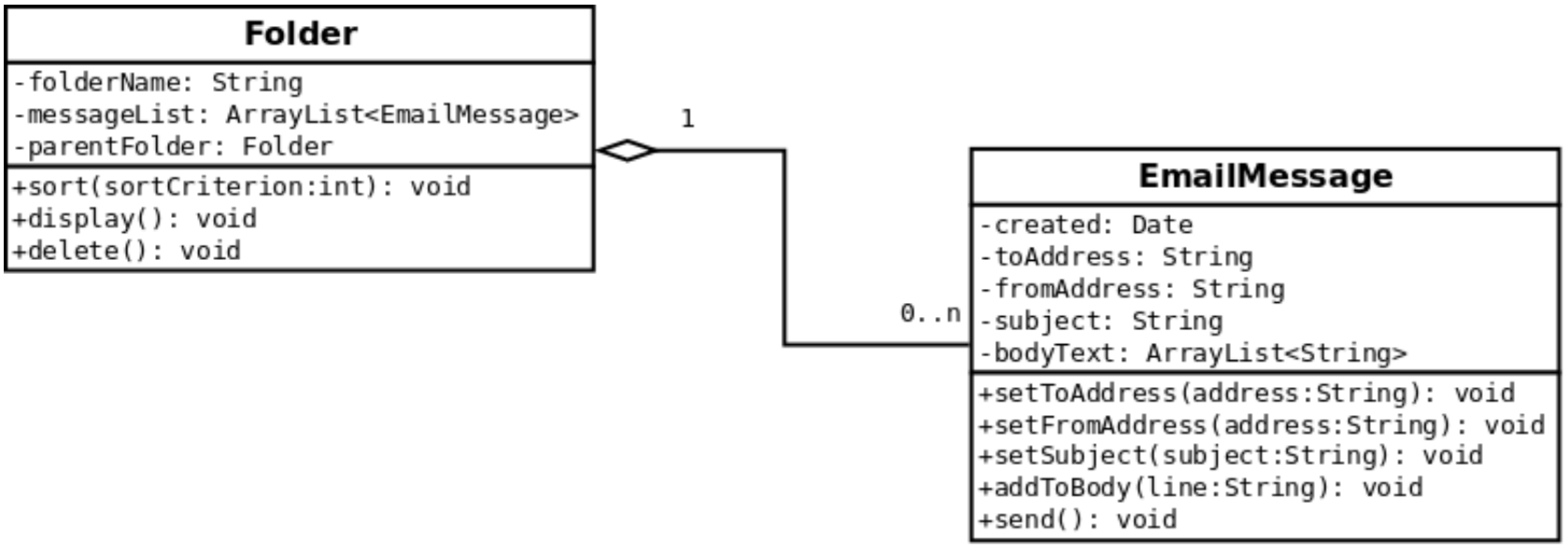
Composition

Both represent part-whole relationships.

Composition is “stronger”. It implies that the part is tightly and permanently bound to the whole. If the whole is deleted, the part should also be deleted.

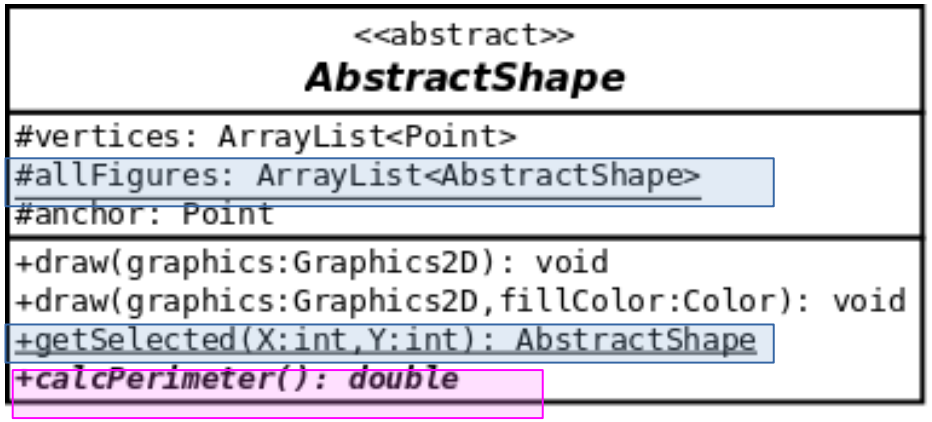
Sometimes it is hard to decide which relationship to use.
Don’t worry too much about this now!

Another Aggregation Example



- A ***Folder*** holds ***EmailMessage*** instances as its “parts”
- Multiplicity is **1 to 0..n.**
- This means one ***Folder*** can have contain no ***EmailMessage*** instances, up to an unlimited number
- This is clearly aggregation, not composition, since an ***EmailMessage*** instance can be moved from one ***Folder*** to another

Representing Abstract Classes



static members/methods

abstract method

This diagram shows conventions for static members/methods (underlined), protected visibility (hash), and abstract methods (italic bold). This is a note which you can add to explain anything on your diagrams.

A UML note

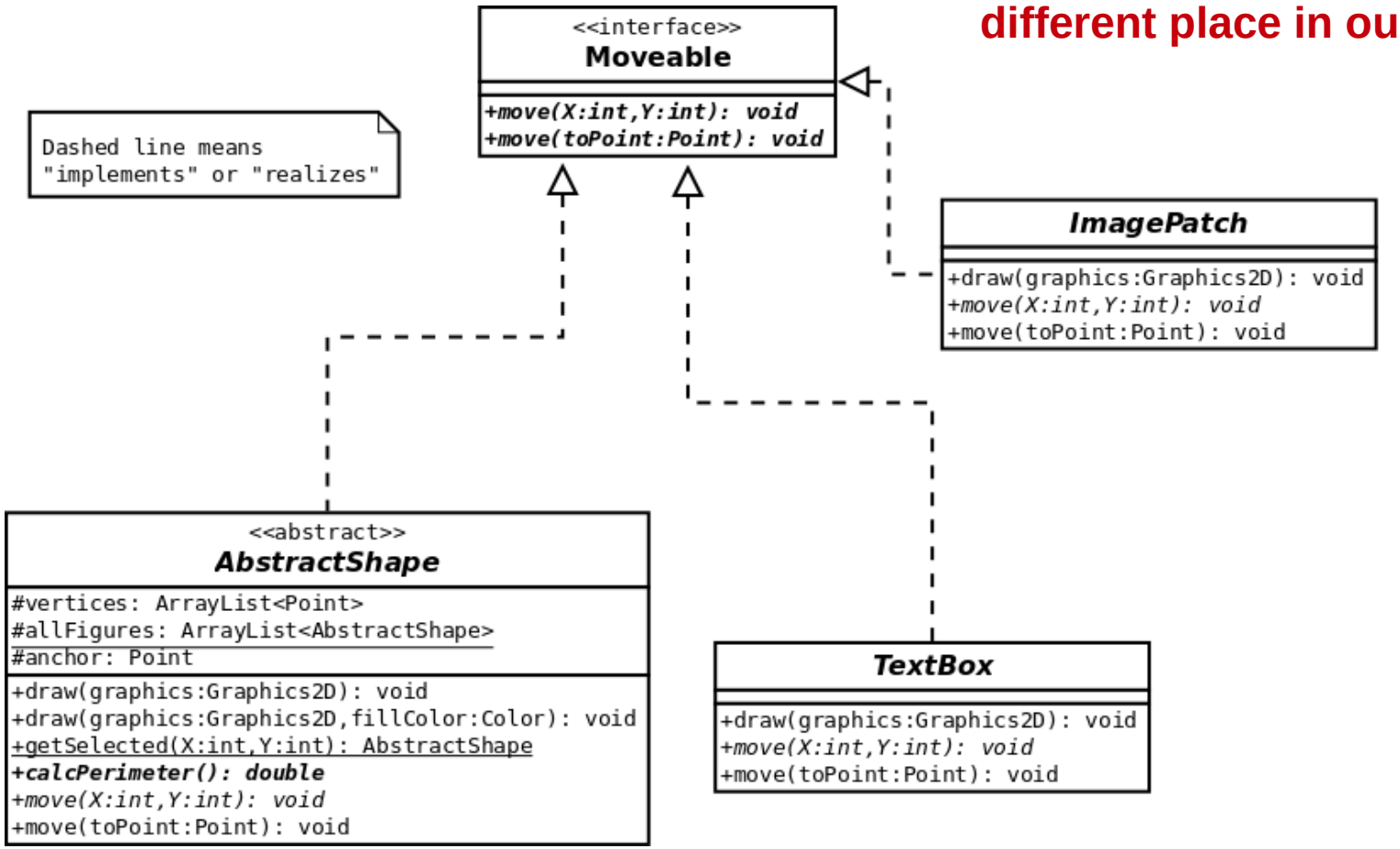
Abstract classes in a class diagram have their class names in italics.

You can also specify a “stereotype” as I have done above, to make this more clear.

Your UML tool should let you choose various options and handle the diagram conventions according to what you specify.

Representing Interfaces

In a graphics editor application, we might decide to create an interface called *Moveable*, for any sort of content that can be relocated to a different place in our drawing.



Class diagram notation is pretty simple

But where do the classes come from?

How can you figure out the classes you need?

Strategies for Identifying Classes

Use concepts from the application domain

- Extract nouns from the specification or requirements for your project
- Use your knowledge about the application domain to suggest additional objects

Let's consider our Scrabble example from last week

Scrabble Domain Concepts

Board

Square

Tile

Word

Player

Dictionary

Strategies for Identifying Classes (2)

Use CRC tables to discover relationships, new classes

CRC = Class--Responsibilities--Collaborators

Class	
Responsibility Responsibility Responsibility	Collaborating class Collaborating class Collaborating class

CRC Table for ***Board***

Board	
Keep track of words on the board and their locations	Word, Square
Check words are legally positioned	Word, Square
Create picture of board for players	BoardImage*
Save board if game paused	BoardFile*
Return information about a particular square (occupied? special? etc.)	Square

(* Newly discovered classes)

CRC Table for ***Word***

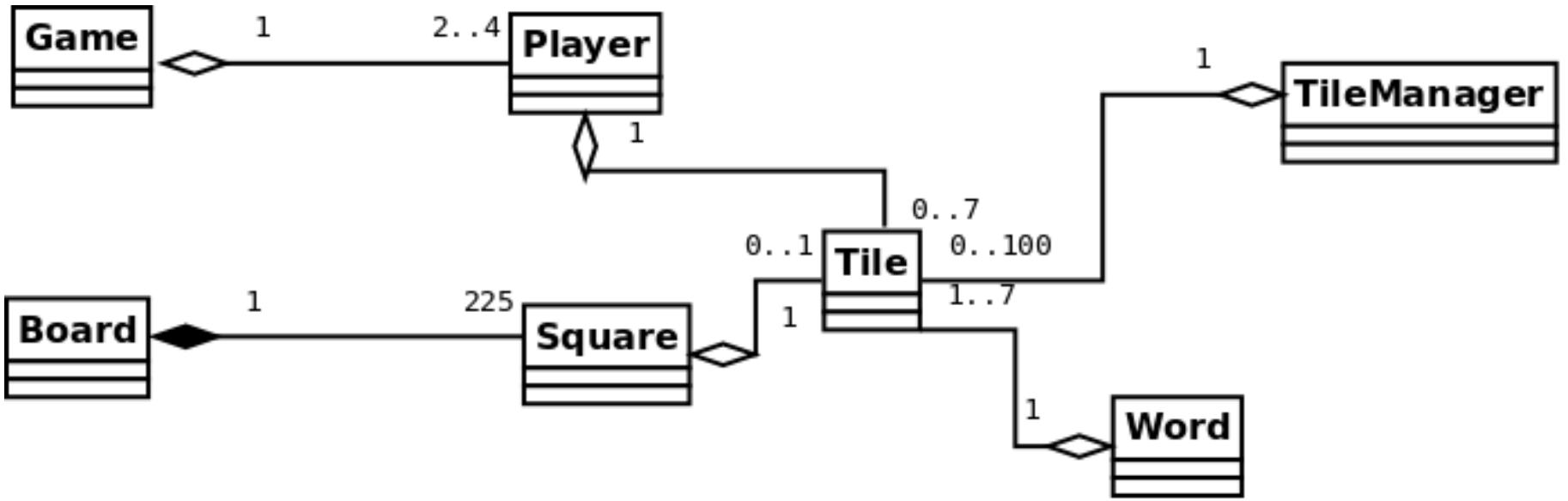
Word	
Calculate score	Board, Tile, Square
Return contents as string	String
Place on board	Board

CRC Table for ***Player***

Player	
Make a word	Board
Select tiles	TileManager*
Return tiles to pool	TileManager*
Challenge a word	Challenge*, Dictionary
Win or lose	Game*

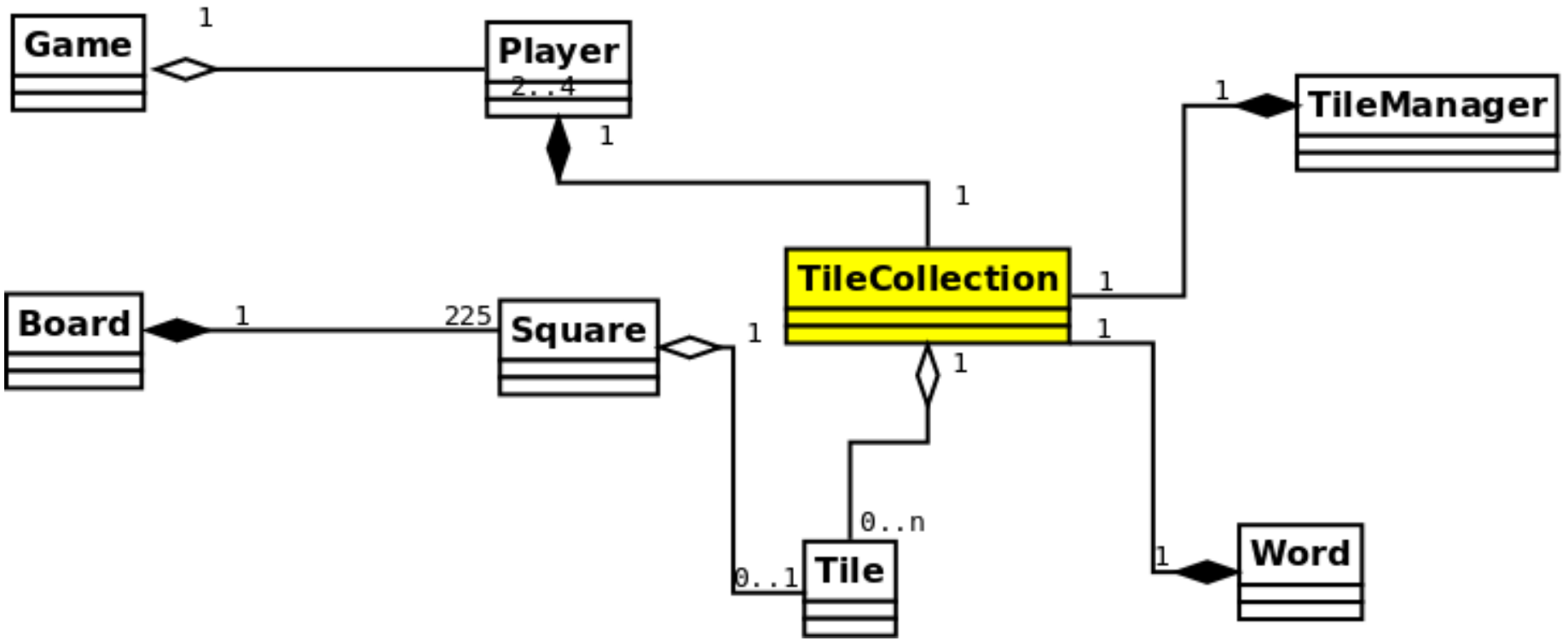
Strategies for Identifying Classes (3)

Consider part-whole (aggregation/composition) relationships



We have a number of one to many relationships between a class and a set of *Tile* instances.

Add **TileCollection** Class

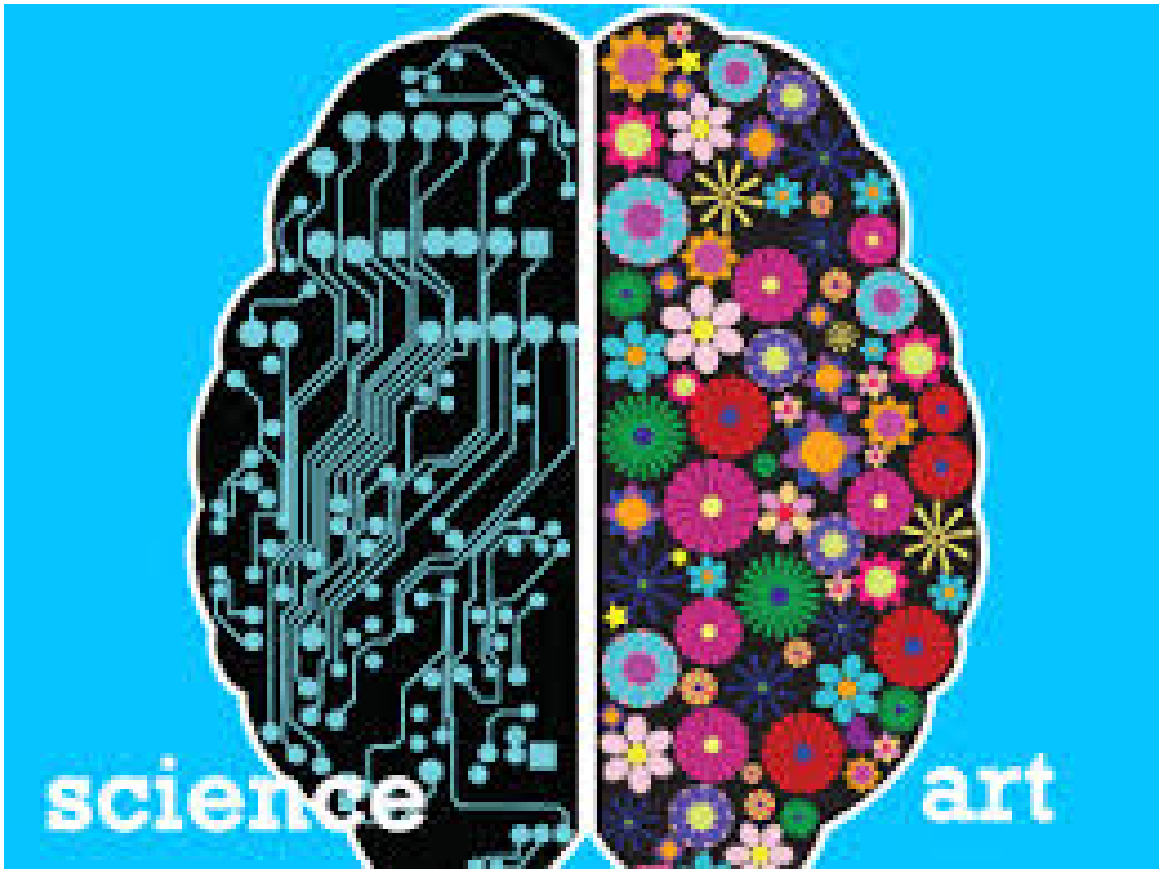


Create new *TileCollection* class to capture common data and behavior and simplify relationships .

- In early stages, focus on identifying major classes and their relationships. Don’t get into too much detail.
- Each class should be responsible for doing only thing only
- Different kinds of responsibility should be separated into different classes
- The object that has the necessary data for a task should be the object that performs that task
- Be willing to change your class hierarchy when you see a way to improve it!

Design is both an art and a science

The more programs you design, the more you will improve!



Assignments

1) Create a class diagram that accurately shows the classes, members, methods and relationships in my implementation of Exercise 5

- Show member data and public methods (private methods not necessary)
- Show generalization and aggregation relationships
- For inheritance or implementation of Java library classes or interfaces, just draw an empty box for the Java class.

Submit as printed, hardcopy diagram(s) with your name and ID

Hand-drawn diagrams will not be accepted

2) Begin working on class diagrams for your own project