More on the UML Class Model

Comp 47480: Object Oriented Design

(Slides based on *Priestly* Chapter 8)

More on the Class Model

- We looked at the UML Class Model when building the domain model for the Table Booking System.

- In this aside we look at a few other important aspects of the class model that didn't appear there. **In 2018, we only cover the bolded topics.**
    - **Aggregation and Composition**
    - **Association Classes**
        - **Object Identity**
    - Qualified Associations

- Based on *Priestly*, chapter 8.

Aggregation and Composition

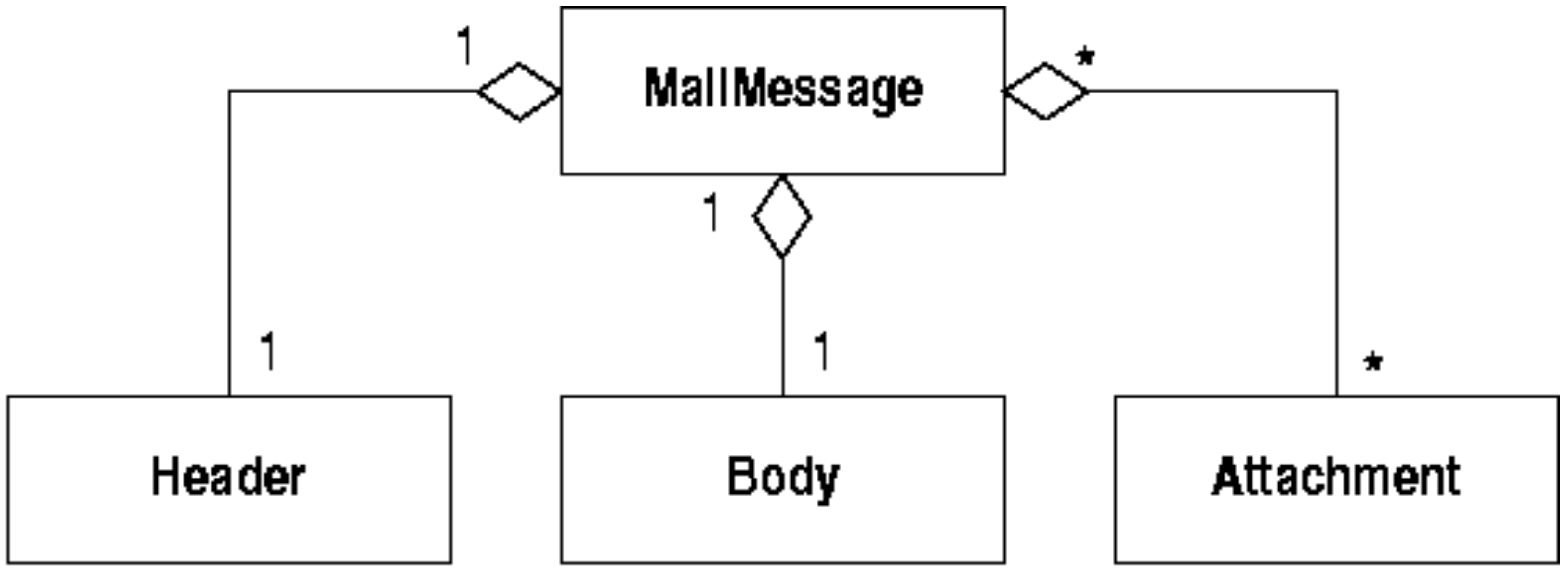
Aggregation: Whole-Part Associations

- Informal, ‘whole-part’ relationships are modelled using **aggregation**
  - a specialised form of general association between classes

- Aggregation is used where there is a notion of ‘weak containment’ between the classes
  - Aggregation has two precise properties, as we’ll see shortly

Aggregation: an example

- The notation is an open diamond on the ‘whole’ end.
- An aggregation association can have standard annotations at ends, e.g.,

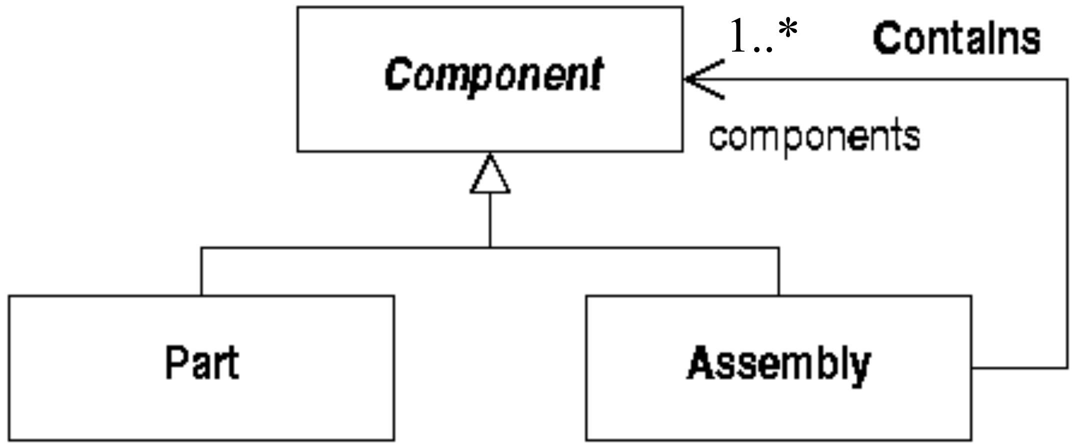


Aggregation: formal properties

- Aggregation has certain formal properties that distinguish it from normal Association
  - We explore these in the next few slides

Aggregation Example: a Typical Composite Structure

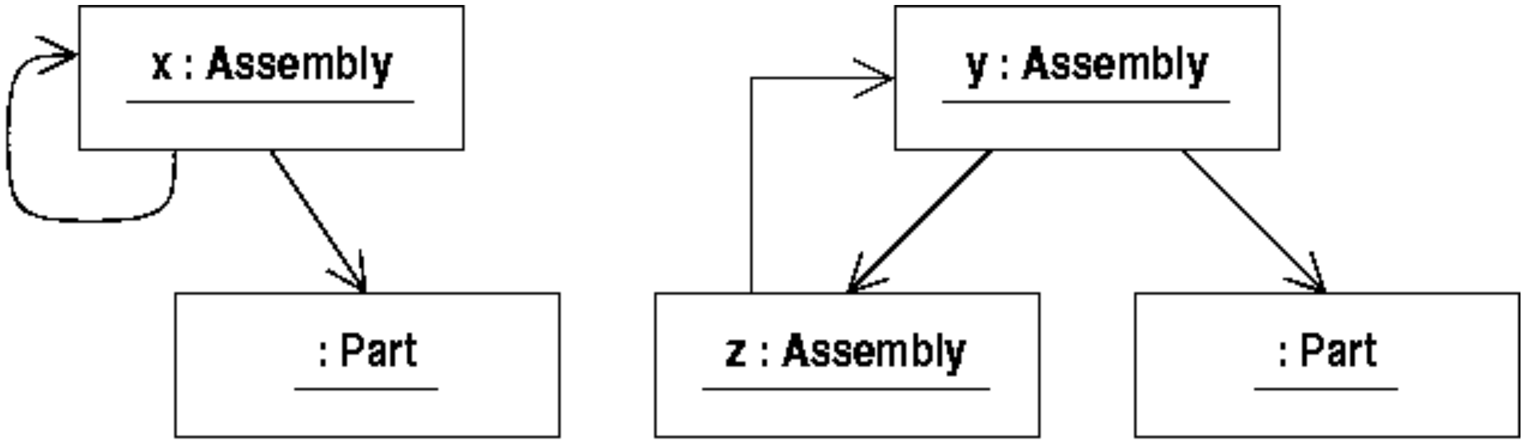
- Assemblies contain Components:
  - an instances of 'Contains' links an Assembly to an instance of either Part or Assembly
  - the Component class is abstract



What sort of **object** structures result from this class diagram?

Oops! This Permits Cyclic Object Structures

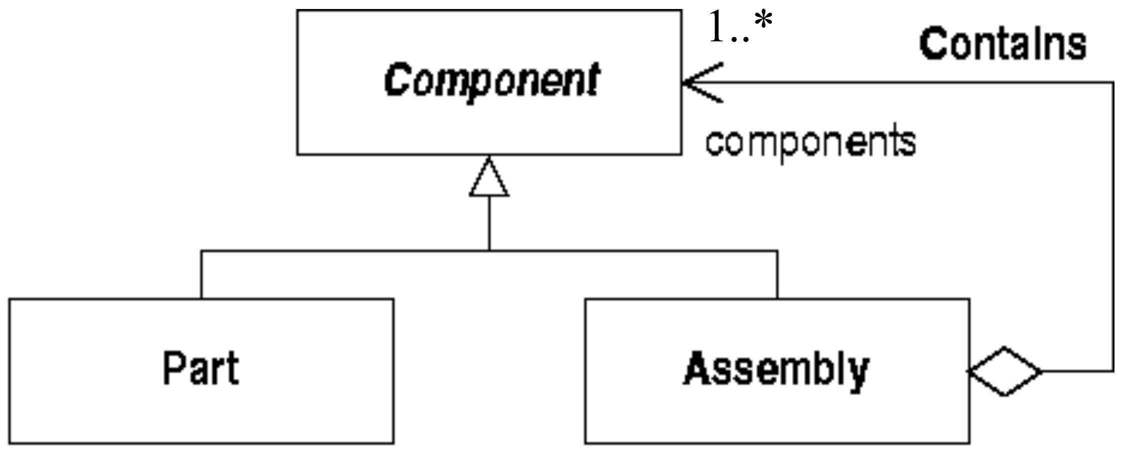
- A (theoretical) problem with the above class model is that it permits undesirable cycles in the object structure, e.g.,



- Given that aggregation means ‘containment’, cycles make no sense whatsoever

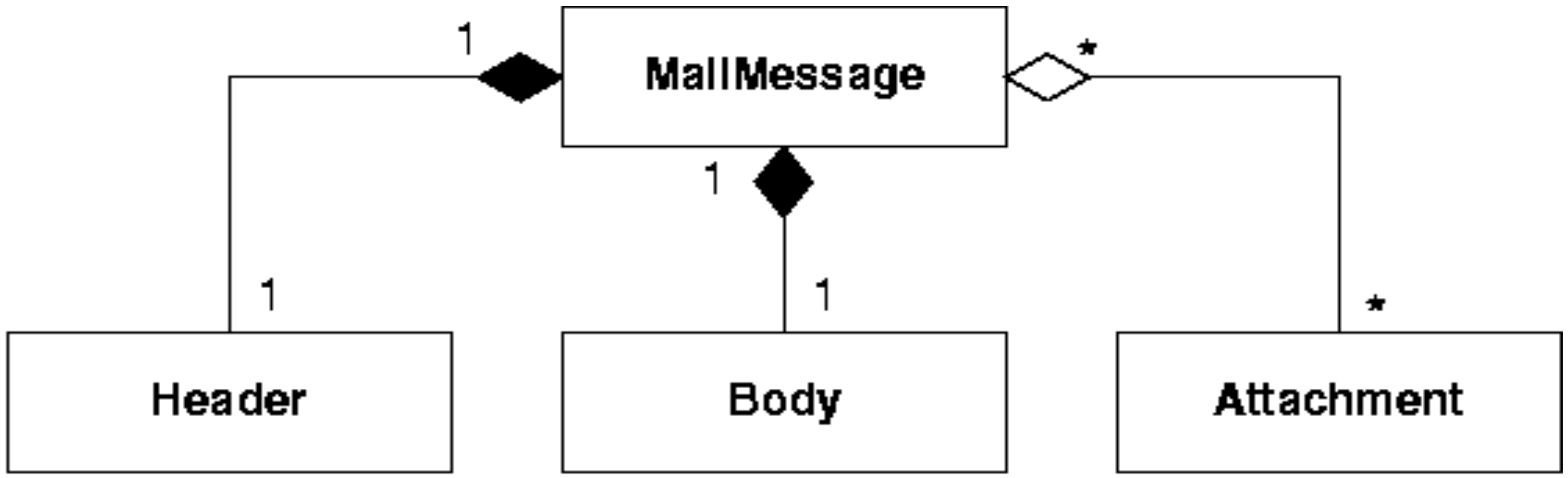
Aggregation solves this

- Aggregation rules out those undesirable cyclic object structures because it is:
  - **antisymmetric**: an object can't link to itself
  - **transitive**: if A links to B and B links to C, then A links to C



Composition: Stronger form of Aggregation

- Composition is a stronger form of aggregation
  - parts can only belong to one composite at a time
  - parts are destroyed when a composite is



Uses, Aggregation and Composition

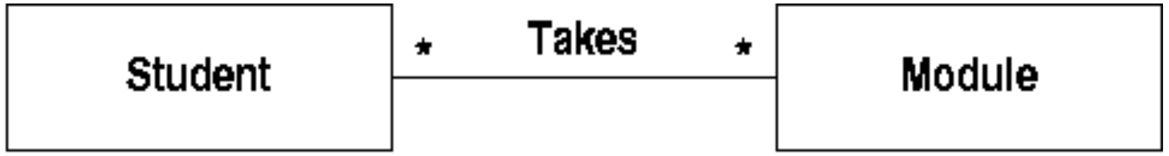
- How would you model the association between these classes?
  - **Club —> ClubMember**
  - **Pond —> Duck**
  - **Person —> BankBranch**
  - **University —> Student**
  - **Car —> Clutch**

- Don’t get too philosophical!
  - In practice, the decision will depend on the details of what the software is doing

Association Classes

Links can have Properties too!

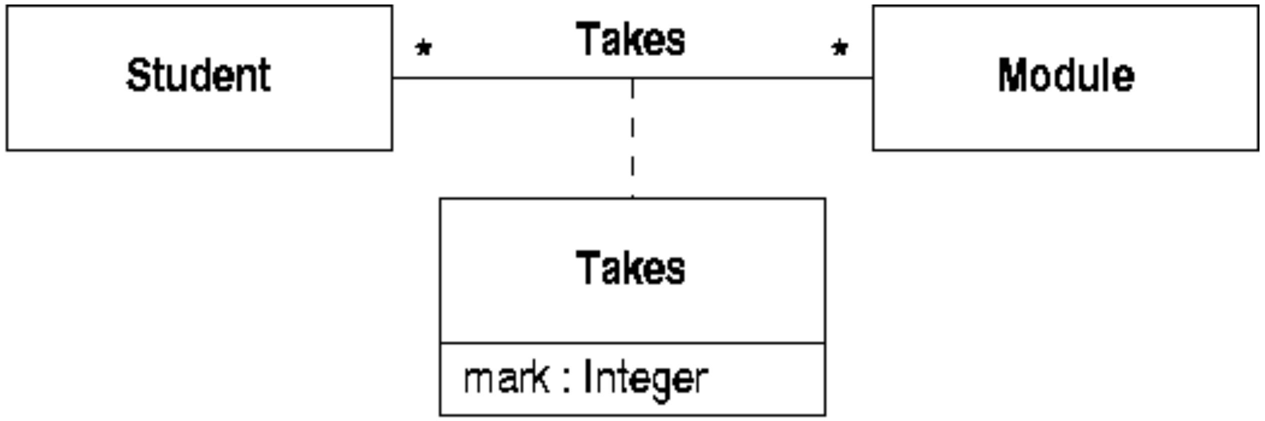
- Sometimes data belongs to a link



- A student takes a module and gets a mark for it
- the mark makes sense only if we know the student and the module
- Can mark be an attribute of either class?

Association + Properties = an Association Class

- Association classes share the properties of associations and classes
  - they can define links between objects
  - they allow attribute values to be stored



- An association class doesn't have the property of **identity** though
  - causes a problem if a student takes the same module twice
  - We now explore **identity** further

Object Identity

Object Identity

- Mathematical objects like numbers don't have identity, e.g.,
    - Here are 3 apples and 3 oranges:

- It's the same 3 in both cases

Real Objects have Identity!

Aside: Object Identity

- Real things have identity, e.g.,
  - Here are two identical watches:

  - They are nevertheless distinct objects
  - Identical software objects are also distinct objects

What's the Output?

```
public class IntIdentity {
   public static void main(String[] args) {

      int a = 2+1;
      int b = 1+2;

      System.out.println(a == b);
      System.out.println(a.equals(b));

   }
}
```

Compile time error on a.equals(b) because int is a primitive type, not an object. (a==b) evaluates to true.

What's the Output?

```
public class IntegerIdentity {
  public static void main(String[] args) {

    Integer a = new Integer(3);
    Integer b = new Integer(3);

    System.out.println(a == b);
    System.out.println(a.equals(b));

  }
}
```

Here we have Integer objects. (a == b) evaluates to false as a and b are distinct objects. a.equals(b) performs a by-value comparison and so returns true.

What’s the Output?

```
public class StringIdentity {
  public static void main(String[] args) {

    String a = new String("hello");
    String b = new String("hello");

    System.out.println(a == b);
    System.out.println(a.equals(b));

  }
}
```

Here we have String objects, similar to previous slide. (a == b) evaluates to false as a and b are distinct objects. a.equals(b) performs a by-value comparison and so returns true.

What’s the Output?

Aside: Object Identity

```
public class StringIdentity {
   public static void main(String[] args) {

      String a = "hello";
      String b = "hello";

      System.out.println(a == b);
      System.out.println(a.equals(b));

   }
}
```

a.equals(b) is true because the strings have the same value.

(a==b) also evaluates to true (surprisingly perhaps) because strings in Java are immutable and interned and so behave like primitive types.

What’s the Output?

Aside: Object Identity

```
import java.util.HashSet;

public class Identity {
   public static void main(String[] args) {

      Integer x = new Integer(12);
      Integer y = new Integer(12);

      HashSet<Integer> numbers = new HashSet<Integer>();
      numbers.add(x);
      numbers.add(y);

      System.out.println(numbers.size());
   }
}
```

A set cannot hold multiple copies of the same element. x and y are distinct objects and are added to numbers set. What’s the result?

The HashSet documentation tells us that the `equals` method is used to prevent duplicates, so adding y to the set has no effect. Output is 1.

Object Identity and Equality in Java

The purpose of the preceding examples is to demonstrate that:

Object Identity is not just an abstract concept. It has a direct impact on what **equality** means and this is crucial in understanding even very simple programs.

Equality can be based on object value or object identity, you simply have to know which is used in which context.

The details vary from language to language, and care must be taken where immutable objects are involved (e.g. strings in Java).

Misunderstanding equality is a major source of programming error!

Summary

- In this section we examined aspects of the UML Class Model in more detail.

- Topics included (only bolded topics in 2018):
  - **Aggregation**
  - **Composition**
  - **Association Classes**
    - **Object Identity**
  - Qualified Associations