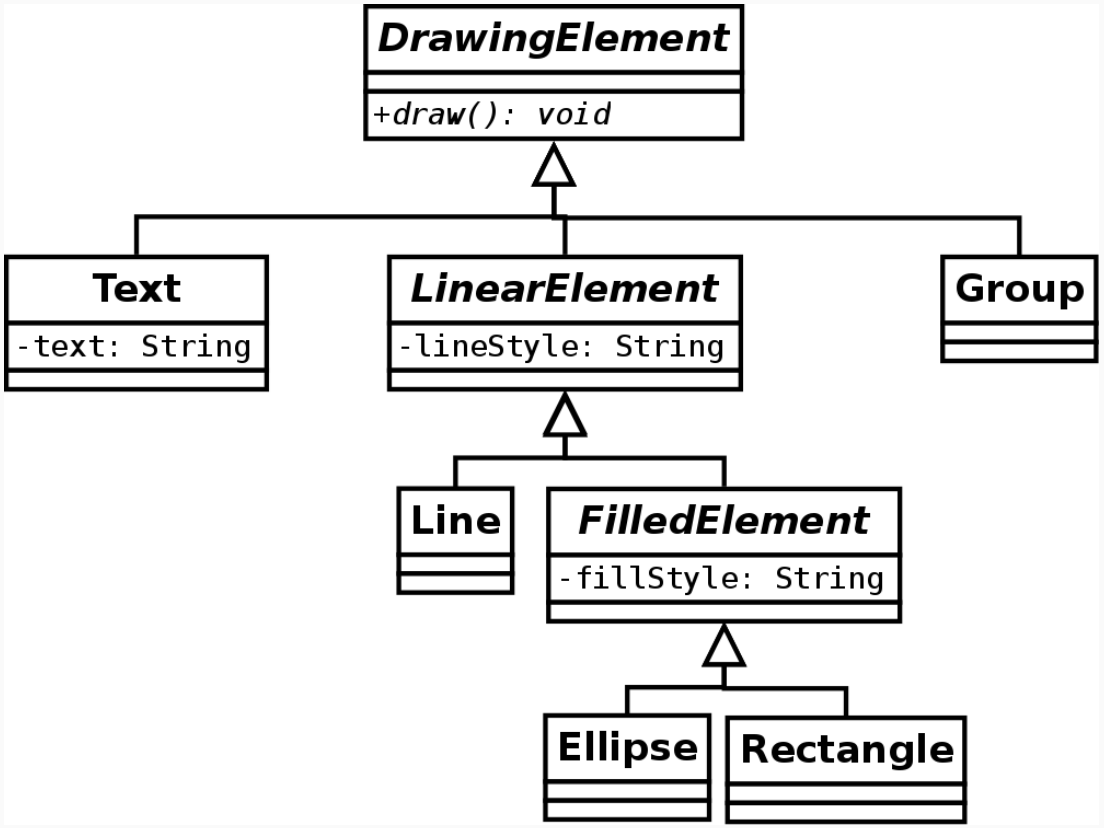
CS-230 Software Engineering

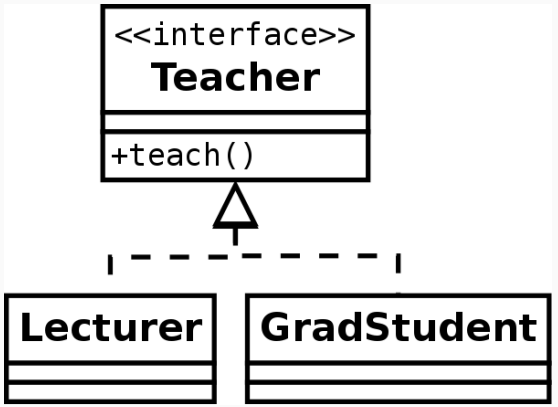
L07: Class Collaborations

Dr. Liam O'Reilly

Semester 1 – 2020

Previously in CS 230...





Responsibilities and Hierarchies!

Previously in CS 230...

- We talked about responsibilities
- Responsibilities are...
  - Knowledge maintained by object attributes.
  - Actions a class can perform behaviours/operations.
  - Should represent purpose of the class in system.

Previously in CS 230... (3)

- Inheritance is an is-a relationship.
- Suppose you have the following classes:
  - Line
  - Circle
  - Square
  - Curve
- What would be good superclasses/subclasses?
- Are these superclasses abstract? If so, what would be good abstract methods for them?

Previously in CS 230... (4)

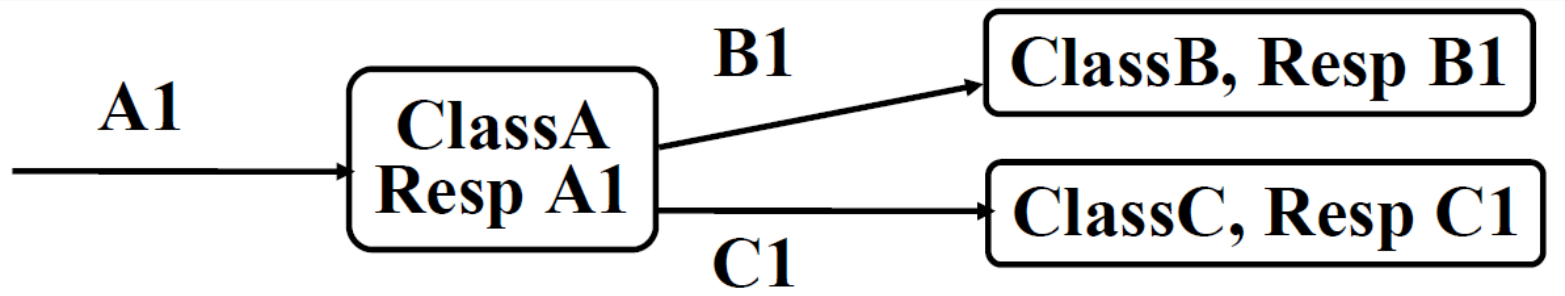
- We have talked about responsibilities and hierarchies
- Today we will talk about another relationships between classes:
- Association, Composition and Aggregation.

Collaborations

Collaborations

- Collaborations represent client object request to a server object.
  - Client may now be able to fulfil its responsibility by utilising server responsibility.
  - May need additional help from more than one server (i.e., other) classes (additional collaborations).
- Server provides services representing its own responsibilities.
  - A collaboration is normally one-way: from client to server.
- Does not mean that information only flows in one direction!
  - Each collaboration addresses a server responsibility.
  - Each collaboration helps to fulfil a client responsibility.
  - However, a client responsibility may need many collaborations.

Collaboration (2)



- A collaborates with B and C to fulfil responsibility A1.
- Represents flow of control and information in system.
  - Defines communication pattern.
  - Helps to identify subsystems – highly collaborating classes.
- Note: the above is not UML.

Collaboration - Example

Let's say we have a class named Notifier that provides the ability (i.e., responsibility) to NotifyUser.

It wants to be able to do this in 2 ways, via Twitter and via email.

Now, this class cannot talk to Twitter nor email. But some other classes can. TwitterSender and EmailSender might be other classes that expose the responsibilities of MakeTweet and SendEmail, respectively.

In this scenario, Notifier is the Client and TwitterSender (and/or EmailSender) are the server(s).

Notifier is now able to fulfil its own responsibility (i.e., NotifyUser) and offer it to the world by utilising the services (i.e., responsibilities of TwitterSender and EmailSender).

Finding Collaborations

- Look for dependencies among responsibilities.
- For each responsibility of each class:
  1. Is class capable of fulfilling the responsibility itself? What does it need?
  2. From what classes can it acquire the data/functionality?
  3. Shared responsibilities can define collaborations.
- For each class:
  1. What data does this class store and services does it provide?
  2. What other classes need these services and information?
  3. If no interactions, can class be eliminated? Only do so after rigorous walk-throughs and careful evaluation.

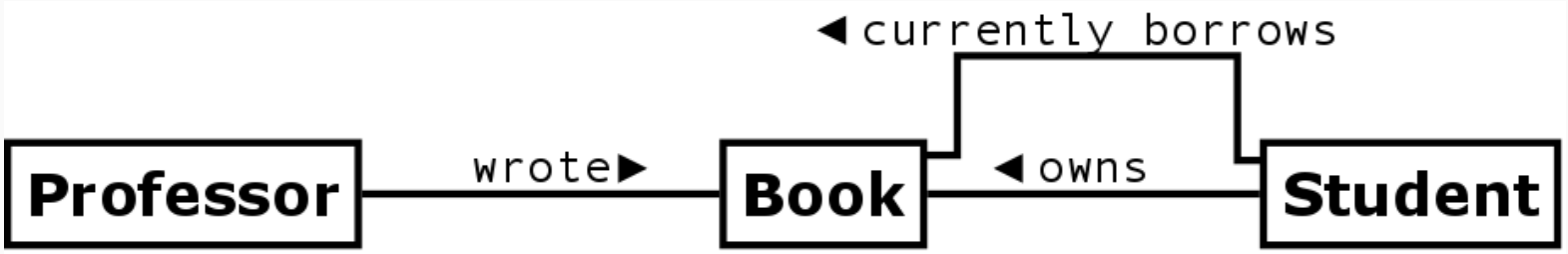
Finding Collaborations

- Has-knowledge-of relationship:
  - If class A has knowledge of B, usually implies a collaboration. Typically A collaborates with B (asks B for a service), though the server may need to have information from client also.
- Is-part-of relationship:
  - Defines a whole-part relationship between 2 classes. One class is a composite object, the other is a component.
  - Composite objects usually have responsibility for maintaining information about its components.

UML: Class Diagrams: Basic Relationships

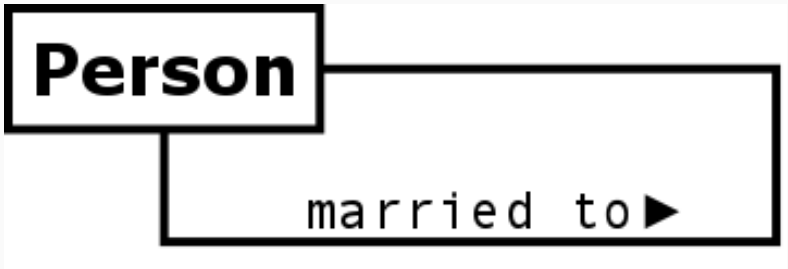
- These are basic associations.
- This model reads "There exists three classes: Professor, Book and Student. Each Professor wrote Books. Each Student owns Books.
- The small black arrow tell you which way round to read the name of the association. I.e. A Book does not own a Student.

UML: Class Diagrams: Multiple Basic Relationships



- You can have multiple associations between classes.
- In addition to students owning books, we also keep track of books borrowed by students.
  - Each arrow is read independently. A Book need not be owned and borrowed, but could be.
  - Here the intention is that different books are owned and borrowed.

UML: Class Diagrams: Self Associations



- A class can be related to itself.
- Here each person is married to other people?
  - Actually, it allows an individual to be married to themself - a tad strange.

Important: Relationships Are NOT Actions

Rule of thumb:

- These relationships should be described by nouns (naming words)
- Do not use verbs (doing words).

Why?

- These relationships represent the current state. A snapshot in time of the system.
- Verbs represent actions, i.e., behaviours that change the current state.

UML: Class Diagrams: Multiplicities

We can specify the number of objects involved in the relationships using multiplicities:

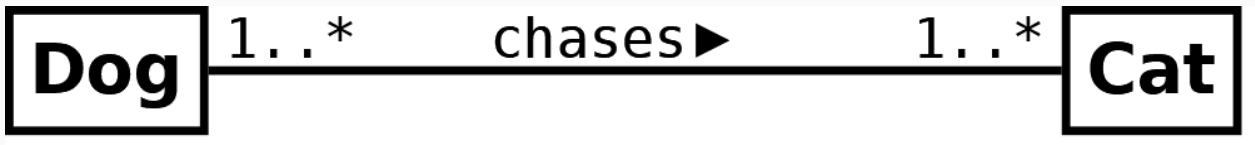
Now the model reads:

- Each Professor writes between 0 and many Books.
- Each Book is written by exactly 1 Professor.
- Each Book is owned by exactly 1 Student.
- Each Student owns 1 or more Books.

Leaving off the multiplicity implies the multiplicity is 1.

Note: It is normally very tricky to enforce a lower bound of anything but 0.

UML: Class Diagrams: Multiplicities (2)



This model reads: one or more Dogs chase 1 or more Cats.

This captures many cases:

- A single Dog chases a single Cat.
- A single Dog chases any number (or heard) of Cats.
- A pack of Dogs chases a heard of Cats.
- A pack of Dogs chases a single Cat.
- It disallows a single Dog that is not chasing a Cat.
- It disallows a Cat that is not being chased.

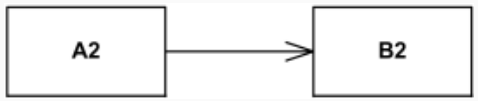
UML: Class Diagrams: Navigability

Associations can also show which classes know about the relationship, i.e., the direction we can navigate.

Each end of an association can be marked:

- navigable (using a stick arrowhead).
- Not navigable (using a small cross).
- Left unspecified (by not using an arrow nor small cross).

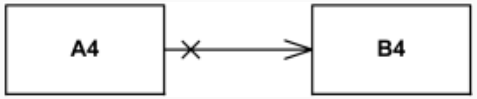
Both ends of association have unspecified navigability. From instances of A1 we might, or might not, be able to get to instances of B1; and vice-versa.



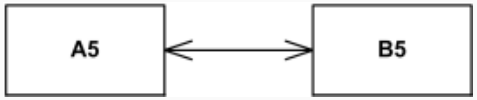
A2 has unspecified navigability. B2 is navigable from A2. From instances of A2 we can get to instances of B2. From instances of B2 we might, or might not, be able to get to instances of A2.

A3 is not navigable form B3. B3 has unspecified navigability. From instances of B3 we cannot get to instances of A3. From instances of A3 we might, or might not, be able to get to instances of B3.

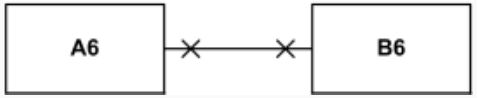
UML: Class Diagrams: Navigability (2)



A4 is not navigable form B4, while B4 is navigable form A4.



A5 is navigable from B5 and B5 is navigable form A5.



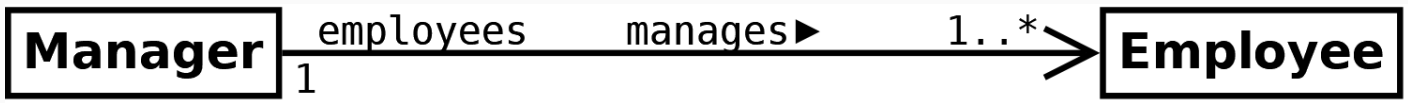
A6 is not navigable from B6 and B6 is not navigable from A6.

UML: Class Diagrams: Navigability Example

- Each manager manages at least 1 employee. Each employee has a single manager.
- From a manager we can get to (and access) the employees.
- However, from an employee we might or might not be able to access their manager.

Remember: the filled in triangular black arrow does not mean anything semantically – it just tells you how to read the sentence.

UML: Class Diagrams: Roles



- We can add a role to each end of the association.
- Here, the people that the manager manages are known as the manager's employees.

This is important if you want to specify the name of the attributes that will be used when translating to code.

```
public class Manager {
  private ArrayList<Employee> employees;
}
```

A Lot to Think About

- Getting the right associations, multiplicities and navigability is not easy!
- You really have to think and go through various Use Case examples (we will cover these soon).
- Think about how your operations of classes will work. How will they be able to carry out their jobs if they can't access the classes they need to (i.e., they need to be able to collaborate).
- Spend time thinking about this when designing. Do not just draw arrows without thinking!

Other Collaborations/Relationships Between Classes

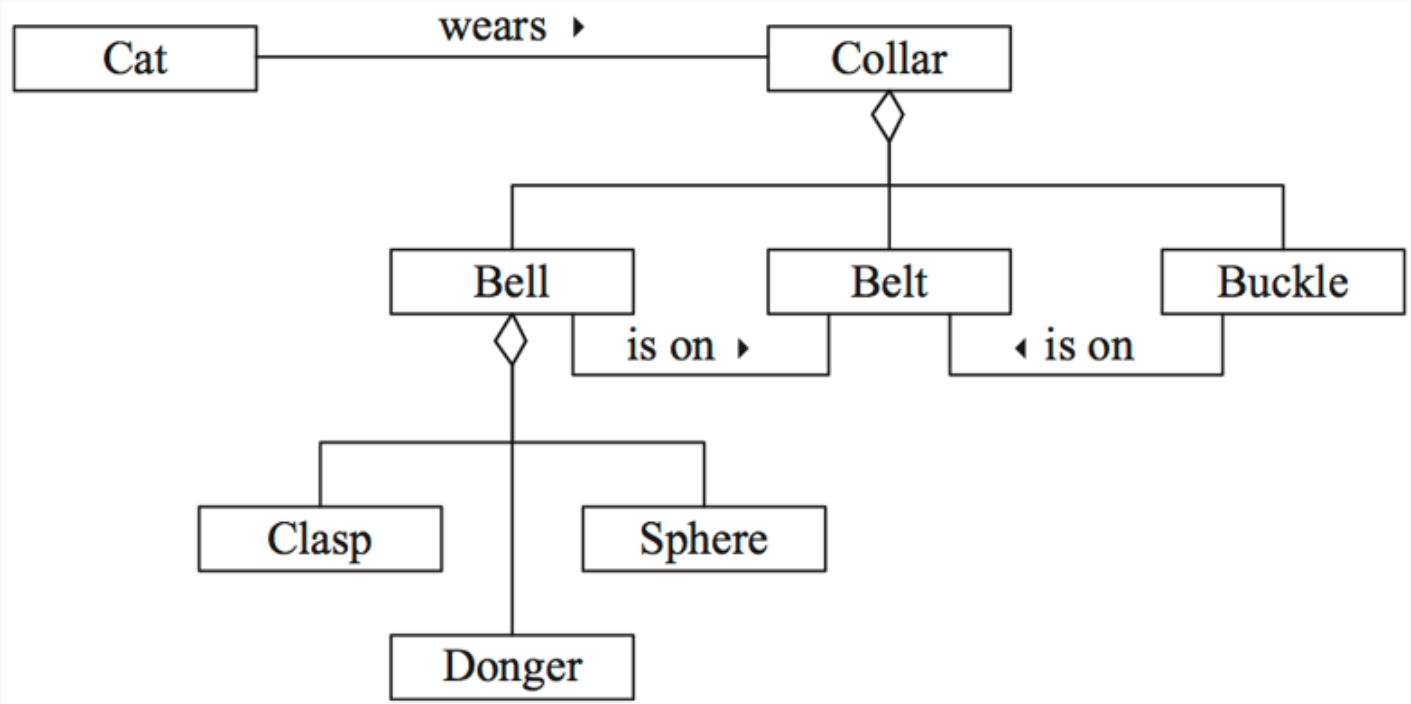
- There are also a few useful other forms of relationships
  - is-a-kind-of (a.k.a. is-a-type-of) – Inheritance hierarchies.
  - is-made-up-of or is-part-of – Aggregation.
  - owns-a – Composition.

Aggregation and Composition

Two types of is-part-of relationships (not inheritance hierarchies)

- Aggregation
  - Child can exist independent of parent.
  - Example: A module can exist independent of a degree course,
    - but a module is part of an degree course.
- Composition
  - Child cannot exist without the parent.
  - Example: a square of a Chess board cannot exist without the board.
  - If the parent is deleted then all the children are deleted.

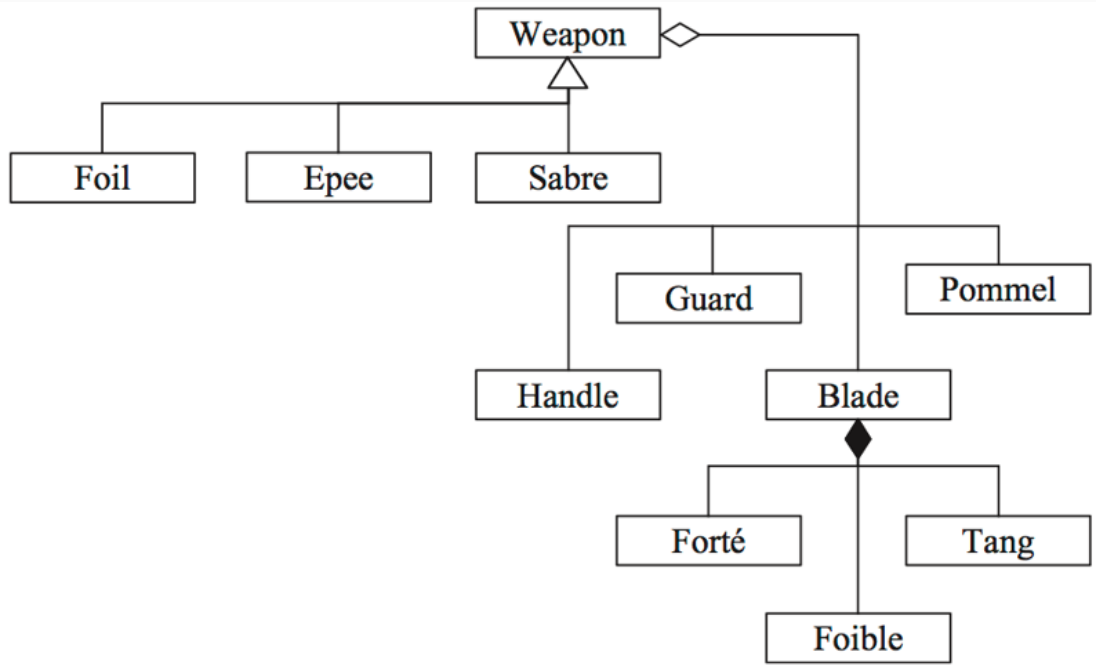
Aggregation

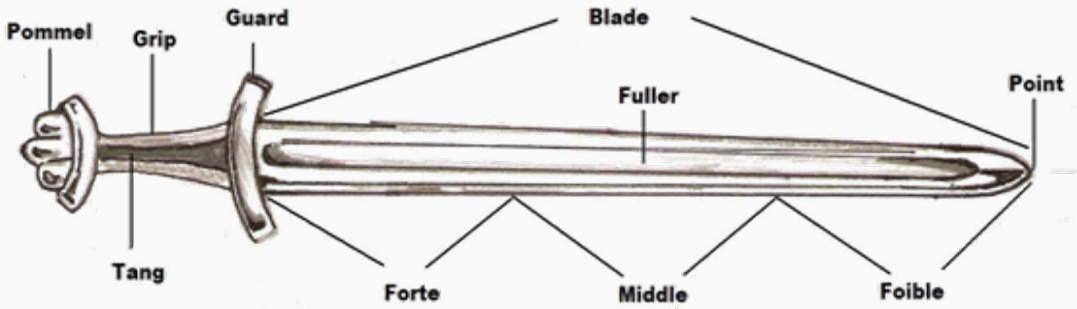


- Aggregation allows assemblies and structures to be modelled.
- Aggregation is shown by adding an unfilled diamond. It is read as "is made up of" (or "has a").
- Collar is a made up of Bell, Belt and Buckle.
- Remember: an omitted multiplicities is assumed to be 1.

Composition

- Composition is similar to aggregation.
- Composition is shown by adding a filled diamond. It is read as "owns a".
- Very subtle difference.
- Composition – made up of components that cannot exist independently of the "parent".





Collaborations to Code

- Collaborations are realised through public method calls.
- We discussed these calls in Java a fair bit last class.
- When listing the collaborations of a class think:
  - What information does this class store?
  - What information does this class need?
  - If there is missing information, where in the design is it stored?
  - Method calls to those classes indicate collaborations.

Review

1. What is...
2. Association?
3. Composition?
4. Aggregation?