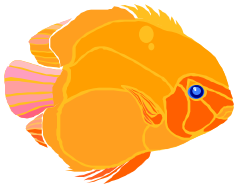
IX. Class Diagrams

Role of Class Diagrams in Requirements Analysis
Classes, Attributes and Operations
Generalization and Inheritance
Associations and Multiplicity
Aggregation and Composition



REMEMBER,

we are modeling the environment within which the system will operate, and how that environment interacts with the system

NOT

the internals of the system (that's system design)

What Must a Requirements Model Include?

- Must contain an overall description of functions.
- Must represent people, physical things and concepts important to the analyst's understanding of what is going on in the application domain
- Must show connections and interactions among these people, things and relevant concepts.
- Must show the business situation in enough detail to evaluate possible designs.
- Should be organized in such a way that it is useful later on during design and implementation of the software.
- Hence a need for more detailed models than use cases!!

==> Hence the need for Class Diagrams!

Classes

- A class describes a group of objects with
  - similar properties (attributes),
  - common behaviour (operations),
  - common relationships to other objects,
  - and common meaning ("semantics").

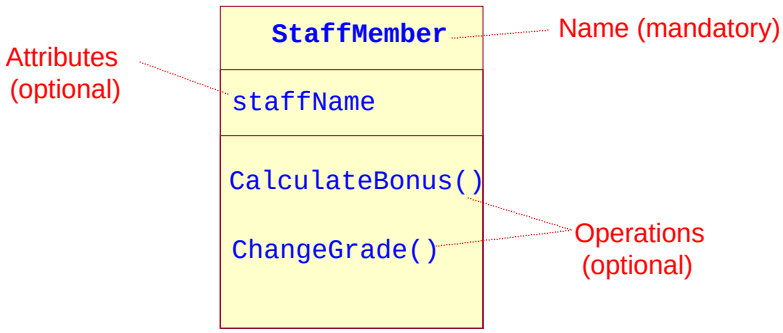
- For example, "employee: has a name, employee# and department; an employee is hired, and fired; an employee works in one or more projects"

Finding Classes

- Finding classes in use case:
  - Look for nouns and noun phrases in the description of a use case;
  - These are only included in the model if they explain the nature or structure of information in the application.
- Don't create classes for concepts which:
  - Are beyond the scope of the system;
  - Refer to the system as a whole;
  - Duplicate other classes;
  - Are too vague or too specific (few instances);
- Finding classes in other sources:
  - Reviewing background information;
  - Users and other stakeholders;
  - Analysis patterns;
  - CRC (Class Responsibility Collaboration) cards.

`StaffMember` Class for Agate

- For example, Agate will want to store information about all its staff members, including current and newly hired staff members.
- The class `StaffMember` is a way of organizing all these objects ("class instances") and defining the set of attributes and operations that apply to all staff.



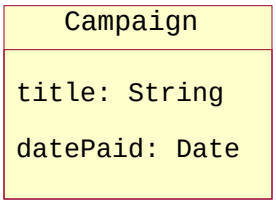
Names

- Every class must have a unique name

- In the Agate system, we shall use ***instances*** of these classes.
- For example, when we `assignStaff` to work on a campaign, we shall use instances of the classes `Campaign` and `StaffMember`
- For each instance of `Campaign` there will be several instances of `StaffMember`.

Attributes

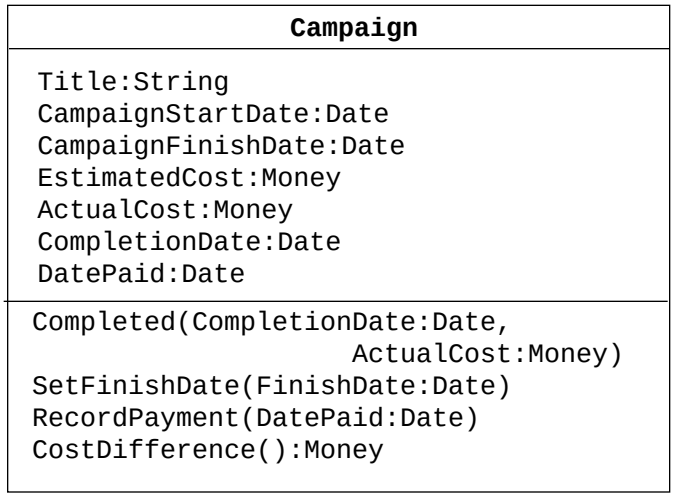
- Each class can have ***attributes*** which represent useful information about instances of a class.
- Each attribute has a ***type***.
- For example, `Campaign` has attributes `title` and `datePaid`.



Operations

- Often derived from actions verbs in use case descriptions.
- Some operations will carry out processes to change or do calculations with the attributes of an object.
- For example, the directors of Agate might want to know the difference between the estimated cost and the actual cost of a campaign
  - campaign would need an operation `CostDifference()`

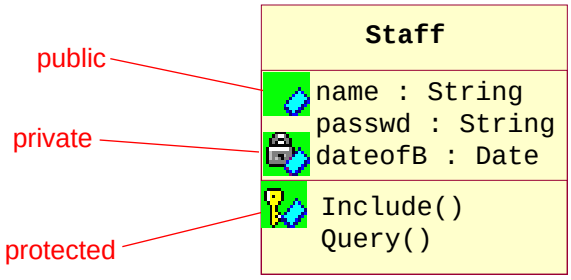
Operations



- Each operation has a ***signature***, which specifies the types of its parameters and the type of the value it returns (if any).

Visibility

- As with Java, attributes and operations can be declared with different visility modes:
  - \+ **public**: any class can use the feature (attribute or operation);
  - # **protected**: any descendant of the class can use the feature;
  - \- **private**: only the class itself can use the feature.



Relationships

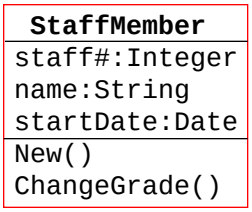
- Classes and objects do not exist in isolation from one another
- A relationship represents a connection among things.
- In UML, there are different types of relationships:
  - ✓ Generalization
  - ✓ Association
    - ✓ Aggregation
    - ✓ Composition
  - ✓ Dependency
  - ✓ Realization
- Note: The last two are not useful during requirements analysis and will be discussed later.

Generalization Relationship

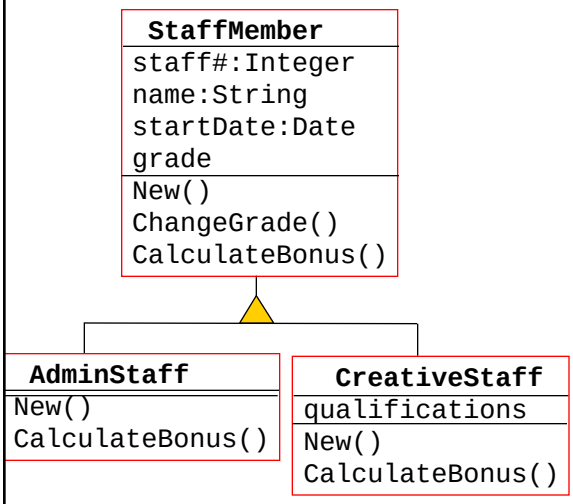
- Generalization relates two classes when the concept represented by one class is more general than that represented by the other.
- For example, `Person` is a generalization of `Student`, and conversely, `Student` is a specialization of `Person`.
- The more general class participating in a generalization relationship is also called the superclass or parent, while the more specialized class is called subclass or child.
- The child always inherits the structure and behavior of the parent. However, the child may also add new structure and behavior, or may modify the behavior of the parent..

Generalization

- It may be that in a system like Agate's we need to distinguish between different types of staff:
  - creative staff and administrative staff;
  - and to store different data about them.
- For example,
  - Administrative staff cannot be assigned to work on or manage a campaign;
  - Creative staff have qualifications which we need to store;
  - Creative staff are paid a bonus based on the work they have done;
  - Administrative staff are paid a bonus based on a percentage of salary.



Generalization



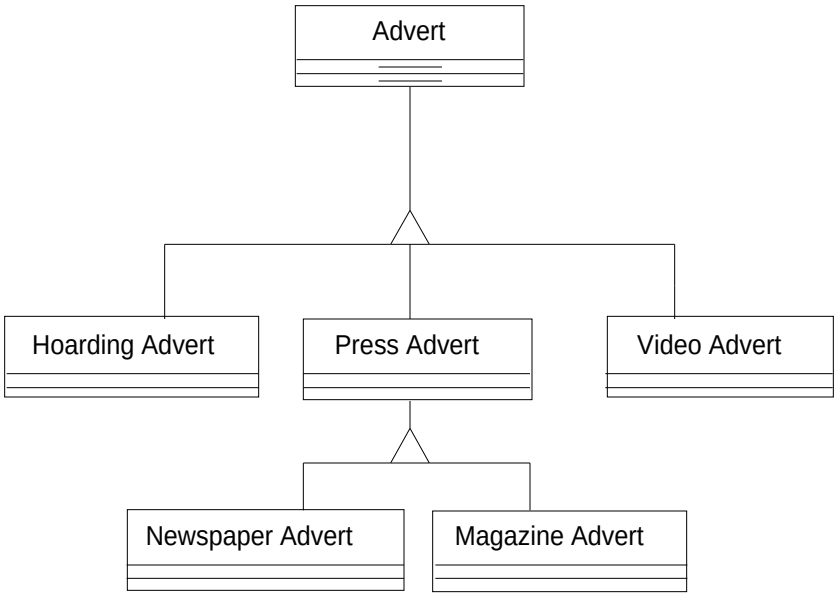
The triangle linking the classes shows inheritance; the connecting line between `AdminStaff` and `CreativeStaff` indicates that they are mutually exclusive. However, all instances of `AdminStaff` and `CreativeStaff` will have a `staff#`, `name`, `startDate`, while `CreativeStaff` will also have a `qualification` attribute.

Generalization

- Similarly, the operation `CalculateBonus()` is declared in `StaffMember`, but is ***overridden*** in each of its sub-classes.
- For `AdminStaff`, the method uses data from `StaffGrade` to find out the salary rate and calculate the bonus.
- In the case of `CreativeStaff`, it uses data from the campaigns that the member of staff has worked on to calculate the bonus.
- When the same operation is defined differently in different classes, each class is said to have its own ***method*** of defining the operation.

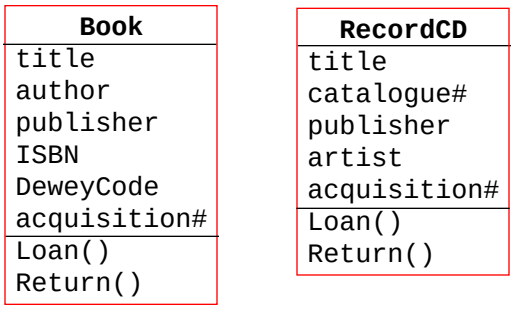
Finding Inheritance

- Sometimes inheritance is discovered top-down: we have a class, and we realize that we need to break it down into subclasses which have different attributes and operations.
- Here is a quote from a director of Agate: "*Most of our work is on advertising for the press, that's newspapers and magazines, also for advertising hoardings, as well as for videos*."

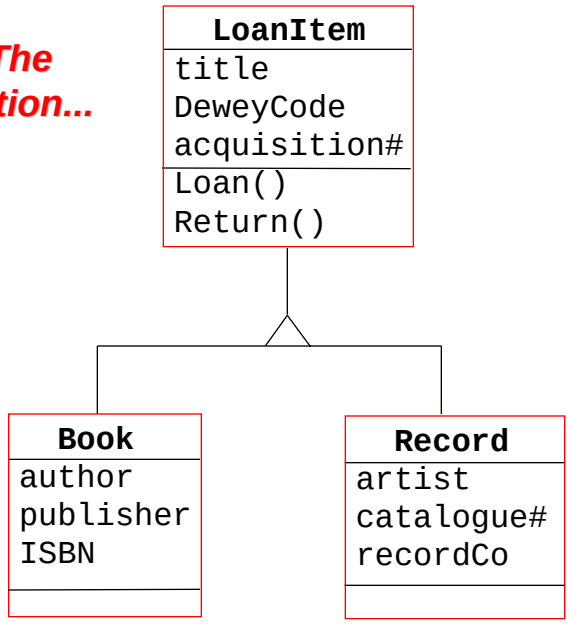


Finding Inheritance

- Sometimes we find inheritance bottom-up: we have several classes and we realize that they have attributes and operations in common, so we group those attributes and operations together in a common super-class.
- Define a suitable generalization of these classes and redraw the diagram

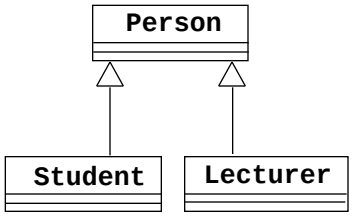


...The Solution...

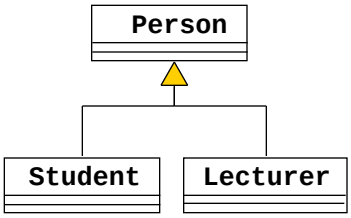


Generalization Notation

Possibly overlapping
e.g., `Maria` is both `Lecturer` and `Student`



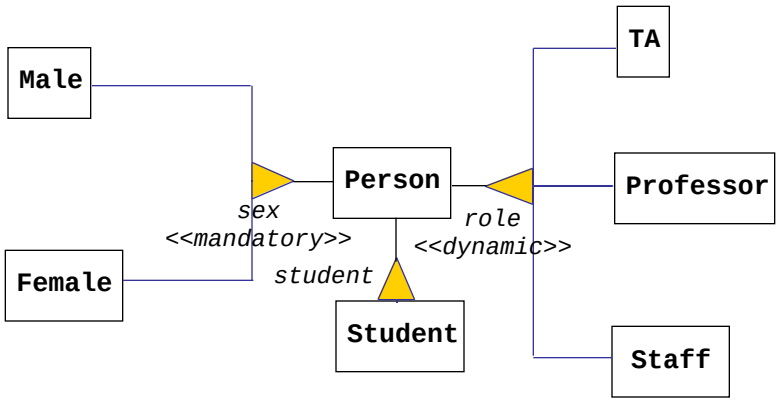
Mutually exclusive
I.e., a lecturer can't be a student and vice versa



Multiple and Dynamic Classification

- Classification refers to the relationship between an object and the classes it is an instance of.
- Traditional object models (e.g., Smalltalk, C++,…) assume that classification is **single** and **static**. This means that an object is an instance of a single class (and its superclasses) and this instance relationship can't change during the object's lifetime.
- Multiple classification allows an object to be an instance of several classes that are not is-a-related to each other; for example, `Maria` may be an instance of `GradStudent` and `Employee` at the same time.
- If you allow multiple classification, you want to be able to specify which combinations of instantiations are allowed. This is done through **discriminators**.
- **Dynamic** classification allows an object to change its type during its lifetime.

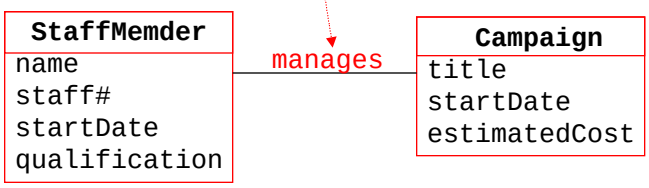
Multiple Classification



- Mandatory means that every instance of person must be an instance of Male or Female.
- <<Dynamic>> means that an object can cease to be a TA and may become a Professor.

Association Relationship

- An association is a structural relationship which represents a binary relationship between objects..
- For example, a person is the child of another person, a car is owned by a person, or, a staff member manages a campaign.
- An association has a **name**, and may be specified along with zero, one or two ***roles***



Association Multiplicity

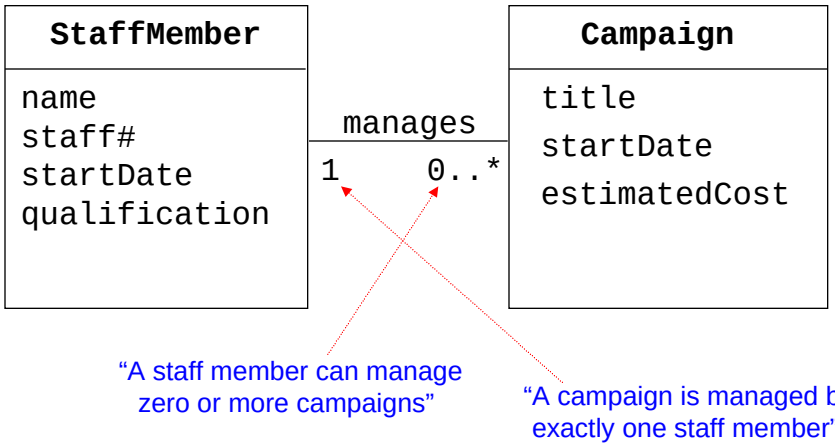
- Can a campaign exist without a member of staff to manage it?
- If yes, then the association is optional at the Staff end - zero or one
- If a campaign cannot exist without a member of staff to manage it
  - then it is not optional
  - if it must be managed by one and only one member of staff then we show it like this - exactly one
- What about the other end of the association?
- Does every member of staff have to manage exactly one campaign?
- No. So the correct multiplicity is zero or more.
  - Kerry Dent, a more junior member of staff, doesn't manage any campaigns
  - Pete Bywater manages two

Multiplicity

- Some examples of specifying multiplicity:

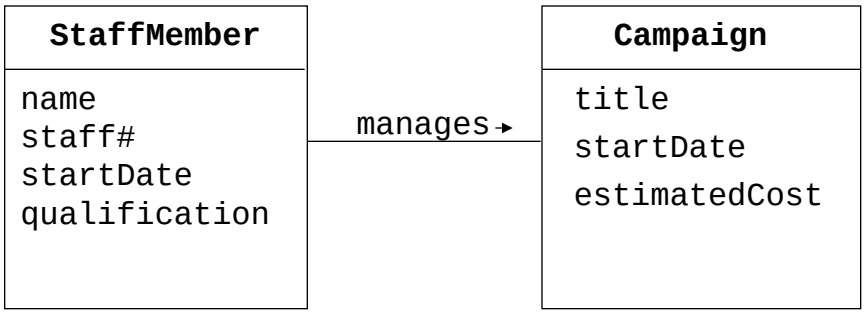
Optional (0 or 1)	`0..1`
Exactly one	`1 = 1..1`
Zero or more	`0..* = *`
One or more	`1..*`
A range of values	`1..6`
A set of ranges	`1..3,7..10,15,19..*`

Associations with Multiplicity



Direction of an Association

- You can specify explicitly the direction in which an association is to be read. For example,

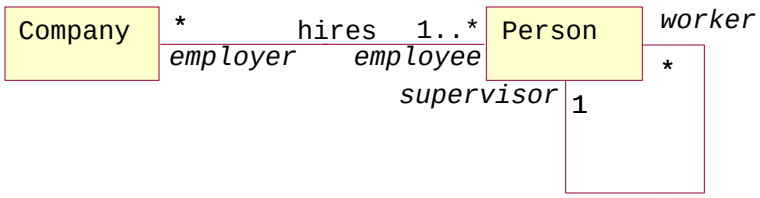


Association Navigation

- Sometimes we want to model explicitly the fact that an association is uni-directional.
- For example, given a person's full name, you can get the person's telephone number, but not the other way around.

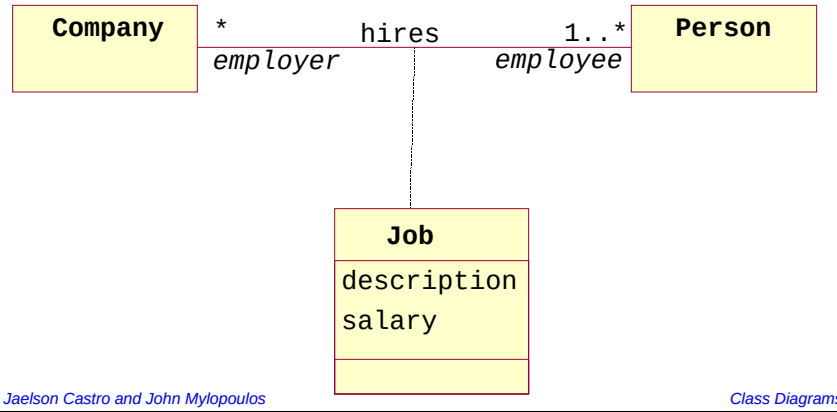
Association and Role

- We can name explicitly the role a class in an association.
- The same class can play the same or different roles in other associations.



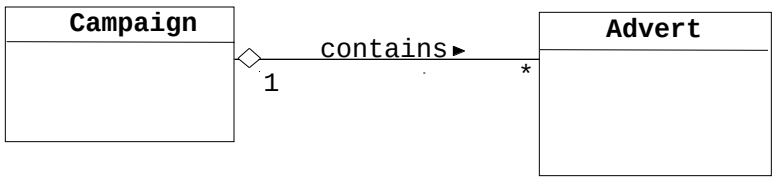
Association Classes

- Sometimes we want to treat an association between two classes, as a class in its own right, with its own attributes and operations.



Aggregation Relationship

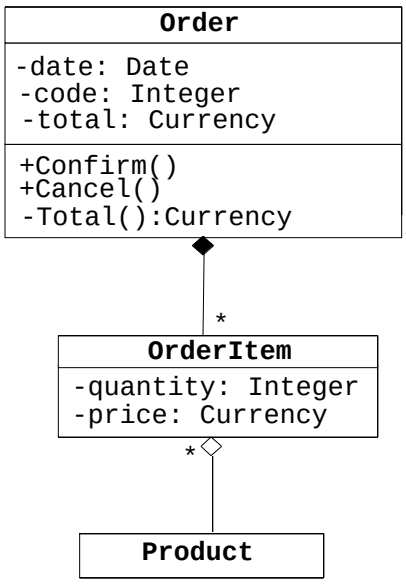
- This is the `Has-a` or `Whole/part` relationship, where one object is the "whole", and the other (on of) the "part(s)".



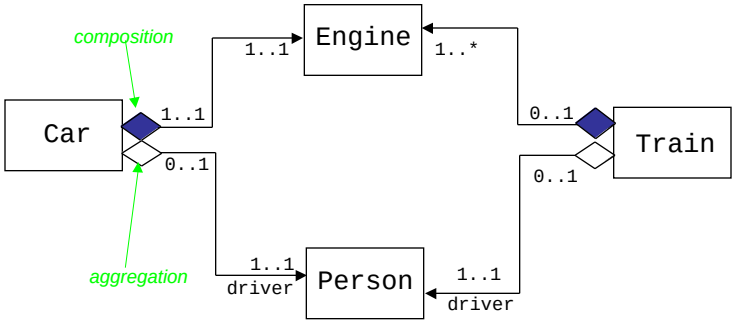
Composition Relationship

- It is a special case of the aggregation relationship.
- A composition relationship implies strong ownership of the part and the whole. Also implies that if the whole is removed from the model, so is the part.
- For example, the relationship between a person and her head is a composition relationship, and so is the relationship between a car and its engine.
- In a composition relationship, the whole is responsible for the disposition of its parts, i.e. the composite must manage the creation and destruction of its parts.

An Example

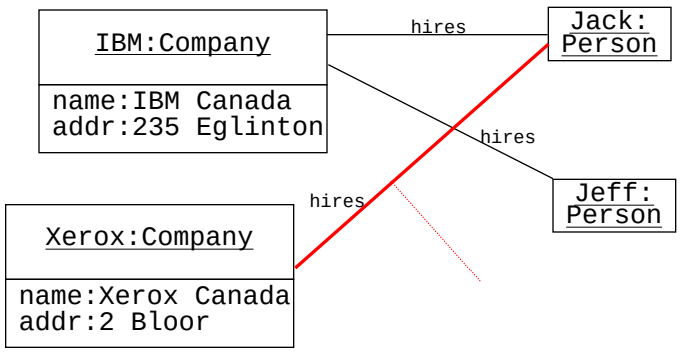


Another Example



Object Diagrams

- These are like class diagrams, except now we model objects, i.e., instances of the classes defined in class diagrams.



Additional Readings

- [Booch99] Booch, G. et al. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999. (Chapters 4, 5, 8, 9, 10.)
- [Fowler97] Fowler, M. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
- [Bellin97] Bellin, D et al. *The CRC Card Book*. Addison-Wesley, 1997.