Aggregation and Composition. Inheritance

Multiplicity
Abstract Classes
Single and Multiple Inheritance
Interfaces
Examples
Aggregation

An **aggregation** is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.

Aggregation is used to model a compositional relationship between model elements. Containment of the aggregated class is **by reference.**
Aggregation – Examples

Examples:
- a **Library** contains **Books**
- within a company **Departments** are made-up of **Employees**
- a **Computer** is composed of a number of **Devices**.

To model this, the aggregate (**Department**) has an **aggregation** association to the its constituent parts (**Employee**).
Aggregation Example.
Shared Aggregation

**Example:** an *Customer* has an *Address*. We use aggregation because the two classes represent part of a larger whole. We have also chosen to model *Address* as a separate class, since many other kinds of things have addresses as well.

An aggregate object can hold other objects together

**Shared Aggregation**
An aggregation relationship that has a multiplicity greater than one established for the aggregate is called *shared*, and destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph, or a tree with many roots.
Shared Aggregation Usage. Example

**Shared aggregations** are used in cases where there is a strong relationship between two classes, so that the same instance can participate in two different aggregations.

*Example:* Consider the case where a person has a home-based business. Both the **Person** and the **Business** have an address; in fact it is the *same* address. The **Address** is an integral part of both the **Person** and the **Business**. Yet the **Business** may cease to exist, leaving the **Person** hopefully at the same address.
Composition

**Composition** is a form of aggregation with strong ownership and coincident lifetime of the part with the aggregate:

- The multiplicity of the aggregate end (in the example, the **Order**) may not exceed one (*i.e. it cannot be shared*).
- The aggregation is also **unchangeable**, that is once established, its links cannot be changed.
- By implication, a composite aggregation forms a "tree" of parts, with the root being the aggregate, and the "branches" the parts.

A compositional aggregation should be used over "plain" aggregation when there is strong inter-dependency relationship between the aggregate and the parts; where the definition of the aggregate is incomplete without the parts. Containment of the aggregated class is **by value**.
Composition - Examples

Example 1: why to have an Order, if there is nothing being ordered (i.e., empty Order without any Line Items)?

Example 2: the Customer Interface is composed of several other classes. Here, the multiplicities of the aggregations are not yet specified. A Customer Interface object knows which Display, Receipt Printer, KeyPad, and Speaker objects belong to it.
Using Aggregation to Model Class Properties

The **Customer** class can have a set of address attributes OR an aggregated **Address** class. How to decide:

- Do the 'properties' need to have independent identity, such that they can be referenced from a number of objects?
- Do a number of classes need to have the same 'properties'?
- Do the 'properties' have a complex structure and properties of their own?

If so, use a class (or classes) and aggregation. Otherwise, use attributes.
Using Composition – Example

Example: the **Customer Interface** keeps track of the current Customer and his/her PIN
Self-Aggregations

Sometimes, a class may be aggregated with itself - one instance if the class is an aggregate composed of other instances of the same class. In the case of **self-aggregations**, role names are essential to distinguish the purpose for the association.

A product may be composed of other products; if they are, the aggregated products are called *sub-products*.

**Aggregation or Association?**

Aggregation should be used only in cases where there is a compositional relationship between classes, where the "parts" are incomplete outside the context of the whole. If the classes can have independent identity, if they are not parts of some greater whole, then the association relationship should be used.
## Generalization

| Generalization | A **generalization** is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element, and contains additional information. An instance of the more specific element may be used where the more general element is allowed. |

A generalization shows that one class inherits from another. The inheriting class is called a *descendant*. The class inherited from is called the *ancestor*. Inheritance means that the definition of the ancestor - including any properties such as attributes, relationships, or operations on its objects - is also valid for objects of the descendant. The generalization is drawn from the descendant to its ancestor.
Single and Multiple Inheritance

A class can inherit from several other classes through multiple inheritance, although generally it will inherit from only one.
Problems with Multiple Inheritance. Repeated Inheritance

• If the class inherits from several classes, you must check how the relationships, operations, and attributes are named in the ancestors. If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.
• If repeated inheritance is used; in this case, the same ancestor is being inherited by a descendant more than once. How many copies of the attributes of Window are included in instances of the last descendant?
Abstract and Concrete Classes

A class that is not instantiated and exists only for other classes to inherit it, is an **abstract** class. Classes that are actually instantiated are **concrete** classes. Note that an abstract class must have at least one descendant to be useful.

A Pallet Place in the Depot-Handling System is an abstract entity class that represents properties common to different types of pallet places. The Pallet Place is not instantiated on its own.
**Inheritance to Support Polymorphism - Subtyping**

**Subtyping** means that the descendant is a subtype that can fill in for all its ancestors in any situation. Subtyping is a special case of polymorphism; it ensures that the system will tolerate changes in the set of subtypes.

*Example*: all the Circle, Rectangle and Triangle classes inherit from the AbstractFigure class. Thus, other objects can call the Draw() method without taking care whose is that method.

You should use generalizations only between classes of the same stereotype.
**Subclassing** constitutes the reuse aspect of generalization. When subclassing, you consider what parts of an implementation you can reuse by inheriting properties defined by other classes. Subclassing saves labor and lets you reuse code when implementing a particular class.

*Example:* in the Smalltalk-80 class library, the class Dictionary inherits properties from Set. Even though a Dictionary can be seen as a Set (containing key-value pairs), Dictionary is not a subtype of Set because you cannot add just any kind of object to a Dictionary (only key-value pairs).
## Interfaces

A model element which defines a set of behaviors (a set of operations) offered by a **classifier** model element (specifically, a class, subsystem or component). A classifier may **realize** one or more interfaces. An interface may be realized by one or more classifiers. Any classifiers which realize the same interfaces may be substituted for one another in the system. Each interface should provide an unique and well-defined set of operations.

**Naming and Describing Interfaces**

**Defining Operations**

**Documenting Interfaces**
More about Interfaces

- An interface specifies the externally-visible operations of a class and/or component, and has no implementation of its own. An interface typically specifies only a limited part of the behavior of a class or component.

- Interfaces belong to the logical view but can occur in both class and component diagrams.
  - In component diagrams - an interface in a component diagram is displayed as a small circle with a line to the component that realizes the interface.
  - In class diagrams - an interface in a class diagram is represented by a class icon with the stereotype “interface.” Thus, it is a 3-part box, with the interface name in the top part, a list of attributes (usually empty) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.

- The attribute and operation sections of the interface class box can be suppressed to reduce detail in an overview. Suppressing a section makes no statement about the absence of attributes or operations, but drawing an empty section explicitly states that there are no elements in that part.
The University Course Registration (UCR) Case Study

Navigation in aggregation

Containment by-value and by-reference (composition versus aggregation)
Inheritance versus aggregation
Inheritance versus aggregation