

CSE 1030

Yves Lespérance

Lecture Notes

Week 5 — Aggregation

Recommended Readings:

Van Breugel & Roumani Ch. 4 and Savitch Ch. 4 & 5

What is Composition ?

Composition is a special type of aggregation. An instance of one class *owns* an instance of another class as one of its parts.

E.g. a `Student` *owns* a `Date` as the date when he/she joined the university.

E.g. a `Person` *owns* a `House`; a `Person` *has* a `Person` as a friend, but does not *own* this `Person`.

Composition can also be represented in UML diagrams.

See how the second version of the `Student` class is implemented in Sec. 4.2 of the lecture notes.

What is Aggregation ?

Aggregation is a relation between classes. An instance of one class *has an* instance of another class as one of its parts.

Also called the *has a* relation. The part is called an *aggregate* of the whole.

E.g. a car has an engine, it has 4 tires, etc.

E.g. a `Student` has a `String` as its id, a `String` as its name, and a `URL` as its homepage.

Aggregation can be represented in UML diagrams. The multiplicity of the aggregation can also be indicated.

See how the `student` class is implemented in Sec. 4.1 of the lecture notes.

Composition vs Simple Aggregation

The main difference between composition and simple aggregation is that in composition the part belongs to its owner, and *cannot be changed without the owner's permission*.

A composed part could even be completely private.

This becomes an issue when the part is mutable, e.g. `Date` for `Student`.

If an accessor returns a reference to the actual part, the client can mutate it. This is the case even if the attribute, e.g. `joinDate` is private.

This results in a *privacy leak*.

To avoid this, the accessor can return a *copy* of the part rather than the original.

The client can then mutate the copy, but not the original part.

5

In summary, to implement composition where the owned part is a mutable object:

- 1) we make the attribute for the owned part private;
- 2) we make the mutators/constructors make a copy of the argument object and assign this copy to the attribute; and
- 3) we make the accessors return a copy of the owned attribute object.

7

A privacy leak also occurs if a mutator or constructor assigns a mutable object that the client has a reference to to an owned attribute, e.g. `Date` for `Student`.

Then the client can still mutate the object, without the owner's permission.

To avoid this, the mutator/constructor can make a *copy* of the object before assigning it to the owned attribute.

Then the client does not have a reference to the copy of the object in the owned attribute, so it cannot mutate it.

6

Deep vs Shallow Copying

When making a copy of an object that owns other mutable objects, it is important to make copies of the parts as well, so that the copy and the original do not share parts.

This is called making a *deep copy*.

If you only copy the references to the parts, you are making a *shallow copy*, and the copy and the original share the parts.

Then mutating one part mutates the other.

8