CSE 1030

Yves Lespérance

Lecture Notes

Week 4 — Implementing Aggregation and Composition

Recommended Readings:
Van Breugel & Roumani Ch. 4 and Savitch Ch. 4 & 5

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 callled 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.

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.

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.

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.

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.

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.

Collections

In many case, an object has a whole *collection* of components, e.g. a `Portfolio` has a collection of `Investment`s (seen in Java By Abstraction), a `Course` has a collection of `Student`s, a `Student` has a collection of `Courses` it has taken.

In math, the main corresponding structuring mechanism is sets.

In Java, there are several mechanisms to deal with collections, in particular, arrays and the classes in the Collection Fremework.

Can use UML class diagrams to represent *having a collection of components*.

The multiplicity of has-a relationship can be *, i.e. 0 or more components.

The Collection Framework

Interface `Set<T>` implemented by `HashSet<T>` and `TreeSet<T>`; order does not matter and no duplicates.

Interface `List<T>` implemented by `ArrayList<T>` and `LinkedList<T>`; order matters and duplicates are allowed.

Interface `Map<Tkey,Tval>` implemented by `HashMap<Tkey,Tval>` and `TreeMap<Tkey,Tval>`, representing a function from Tkey to Tval.

Iteration over Collections

Often you need to do some operations on each element of a collection. This is called *iterating* over the collection.

Classes that have collections as part provide ways to iterate over their elements.

`iterator()` method of `Set<T>` and `List<T>` returns an `Iterator<T>` object.

Can get successive elements by calling `next()` on iterator.

Can check whether there are more elements by calling `hasNext()` on iterator; returns true iff there are more elements.