Modelling Relationships

Packages

- Packages and Namespaces are logical groupings.
- Packages contain classes.
- Classes can have relationships amongst each other within the package and across the packages.
- Needed so that, eg, classes with the same name do not clash and confuse the compiler.

Logical vs. Physical Grouping

- Logical → a placeholder, or a context in which you put classes in.
  - Package is a logical grouping.
- Physical → where do these classes physically reside?
  - In C++, we place them in libraries, DLLs, or libs, or shared objects.
  - In Ruby, we put them in gems.
  - In Java, we put them in JAR files.
  - In C#, we put them in assemblies.
  - Physical grouping may contain multiple packages or namespaces. A package or a namespace may potentially reside across multiple physical groupings.

Associations

- It’s a relationships between two classes.
- It’s in a class diagram.
- Represented in the class in the code.

Link

- A link is an instance of an association.
- An association can have multiple instances, that is,

- Often appear as verbs. Eg. John *works for* XYZ company.
- Can give names to associations. Eg. `marriage` between a `Man` object and a `Woman` object.
- Can provide more than one relationship between two classes.
- Can provide names for the endpoints. Eg. `Man` object has a ‘wife’, and the `Woman` object has a ‘husband’.

multiple links.

- It’s in an object diagram.
- We traverse the link of an object at runtime.
- Eg. Joe can be an instance of `Man`, Susan can be an instance of `Woman`, then the relationship of marriage between them is a link.

Bi-directional vs. Unidirectional

- All relationships are bi-directional, unless specifically stated.
  - Eg. if `Man` is married to `Woman`, then `Woman` is married to the `Man` as well.
- Sometimes relationships are unidirectional.
  - Eg. ‘I love this book.’ The book doesn’t love you back.

Direction of Navigation

- Bi-directional navigation → given one one object, you can find the other.
- Sometimes there isn’t bi-directional navigation.
  - Eg. a `Teacher` may have the foreign key of all the students, but the `Student` may not have the foreign key of the teacher. Finding a `Student` given a `Teacher` would be easy — just go to the `Teacher` (joining) table and check the foreign key. However, finding the `Teacher` given a `Student` is not easy — you’d have to go the `Teacher` tables and figure which `Teacher` this `Student` belongs to. Here, navigating in one direction is easy, but navigating in the other direction is not so easy.

Ternary and Higher Order Associations

- Binary → relationship between just two objects.
  - Eg. a marriage has only two partners involved.
- Ternary → three objects in a relationship.
  - Eg. a couple and their kid.
- n-ary
  - Complicated.
  - Hard to implement.
  - Better to avoid.
  - Model as binary. Multiple binaries.
    - Eg. here's a man with a daughter, here's a woman with a daughter, and the man and the woman are married.

Cardinality of Associations

- Multiplicity of a relationship.
- One → one object on either side.
  - For every class level association, there's a link at the object level.
- Optional → may or may not have an object at the instance level.
  - If there's no class level association → no instance level association possible.
  - If there's class level association → there may or may not be an instance level association.
- Many → represented usually by a star `*`.
- Zero or more → represented by `0..*`
- One or more → represented by `1..*`

Link Attributes and Association Classes

- Association → between two objects of equal stature.

    - No subordinate relationship.
    - No part-whole relationship.
- Self-Association → one instance of a class is related to another instance of the same class.
    - Eg. a `Man` could have a mentor who is also a `Man`.
    - In a rare situation, one instance could be associated with the same instance.
        - Eg. a `Man` praising himself (haha).
- Link Attributes → part of an association class.
    - Attributes that belong to the relationship, not to one side.
    - Present when there's many-to-many relationships.
- Association Class → stores the link attributes.

Qualifiers

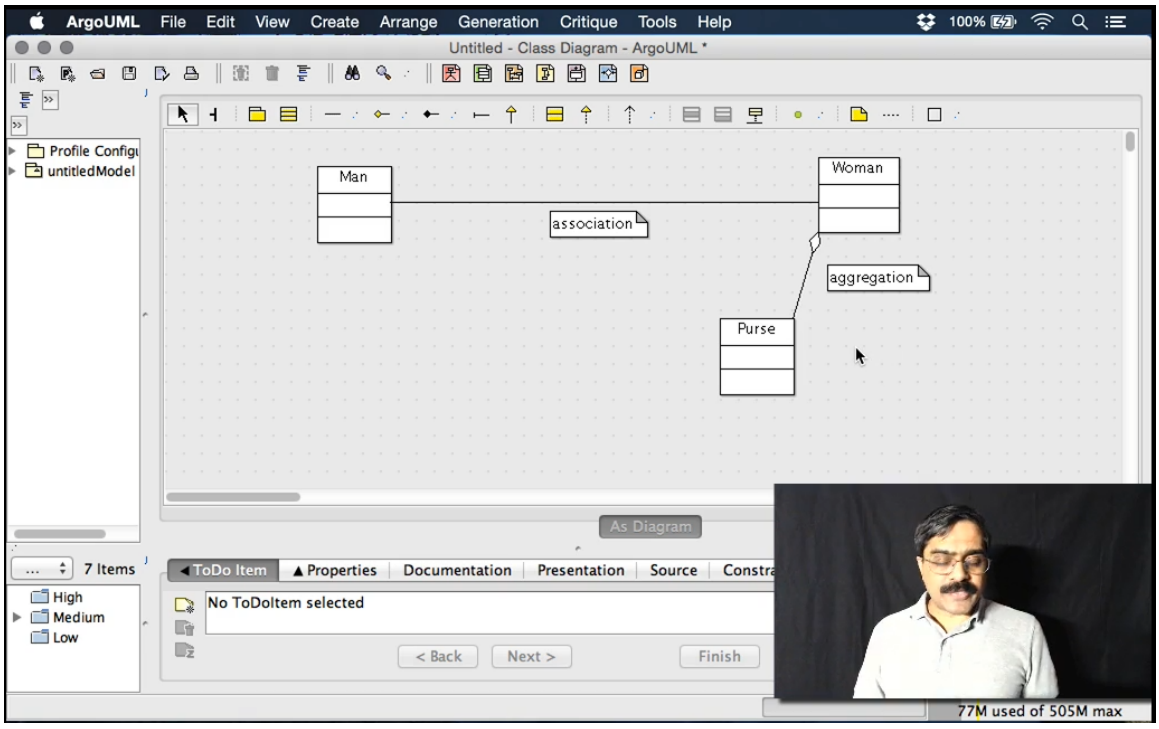
- Used when a collection of objects are being used.
- Distinguishes among set of associated objects.
- Modelled as associate arrays, dictionaries (hash map).

```
Map<String, Student> students
```

Association vs. Aggregation

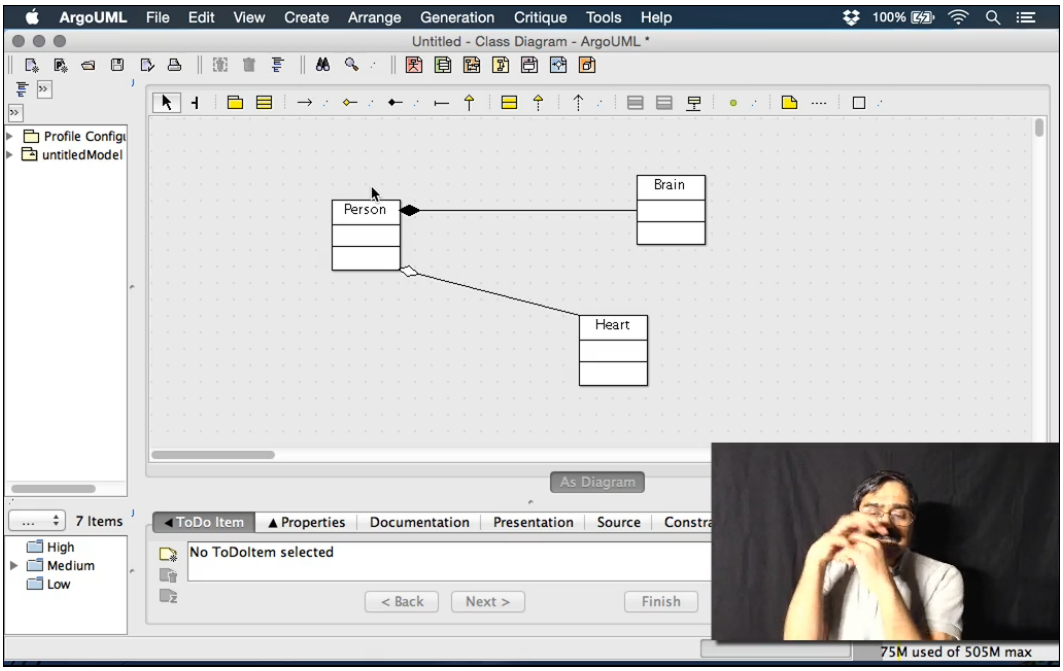
- Association → two object of **equal stature** are related to each other.
    - Eg. relationship between `Man` and `Woman` object.
- Aggregation → typically a '**has a**' or a '**part-whole**' relationship.
    - In the UML diagram, typically, there's a diamond on the side of the 'owner'.
    - Eg. `Man` has a `Car`.
    - Eg. `Woman` has a `Purse`.

- The diamond is on the `Woman` side.



Aggregation vs. Composition

- Composition → a particular object owns another object, but their lifetimes are tied together.
  - Destroying the owning object would destroy the owned object.
  - In UML diagram, there's a shaded diamond.




```
class Brain {};
class BetterBrain : public Brain {};
class Heart {};
class GenerousHeart : public Heart {};

class Person {
    Brain brain; // creation of Person will create Brain object
    // you never instantiate the object of Brain separately.

    // this person cannot have Better Brain because of compositi
    // you are stuck with Brain

    Heart* pHeart;

    // Brain is fully attached to Person, while Heart isn't.
    // * Relationship between Person & Heart is that of aggregat
            // Person owns the Heart, but can change it.
    // * Relationship between Person & Brain is that of composit
            // Person is composed of Brain. The lifetime is tied
            // you cannot change the Brain.
```

```
public:
    Person() {
        pHeart = new Heart();
    } // heart can be replaced with a better heart
    void changeHeart(Heart* pHeart2) {
        delete pHeart;
        pheart = pHeart2;
    }
};
```

```
class Brain {}
class BetterBrain extends Brain {}
class Heart {}
class GenerousHeart extends Heart {}

class Person {
    Brain brain; // this is a reference or pointer
    Heart Heart; // this is a reference also

    public Person(Heart* theHeart){
        brain = new Brain();
        heart = theHeart;
        // OR
        heart = new Heart();
    }
    // no method to change brain. This comes composition.

    public void changeHeart(...){}

};
```

```
class Woman {
    Purse purse;
```

```
    public void skate(Skateboard board){
        // the woman doesn't carry the skateboard all the time.
        // woman depends on the skateboard temporarily, within t

        // this is a simple dependency, not an association or ag
    }
}
```