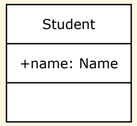
Object-Oriented Programming

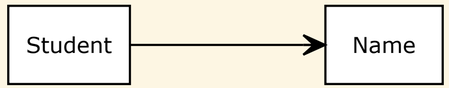
UML Association

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Properties: Attributes & Association (review)



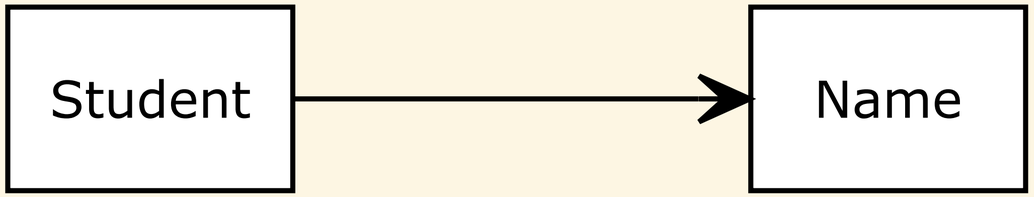


- Another notation for *property* (alternative to *attribute*)

- Use attributes for properties that are *datatypes*, e.g., primitive types or whatever is considered a data type in your design

- Use an association for a *class*, especially if the methods of the class are of interest at the design level

Association: One class has an element of another class



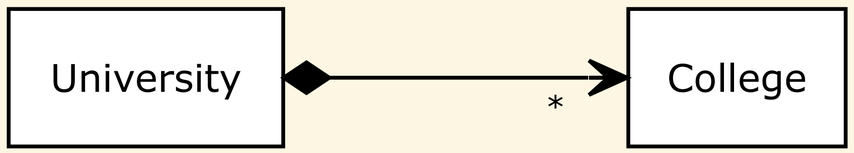

```cpp
#include "Name.hpp"

class Student {
private:
    Name name;
};
```

UML Class Relationships

Relationship	**Description**
Dependency	*uses a*
Generalization	*is a*
Association	*has a*
Aggregation (Association)	*has a*
Composition (Association)	*has a*

Composition



- Target belongs **only** to the source
- Source (e.g., *University*) controls the *lifetime* of the target (e.g., *College*)

Composition Code

```cpp
#include "College.hpp"

class University {
public:
    void addCollege(const College& college);
private:
    std::vector<College> colleges;
};
```

- Code: fields/data members of

  direct type

  containers

  pointers/references to objects created (and destroyed) internally (discussed later)

Aggregation

- *part of* relationship
- A special kind of association where roles and multiplicities are often used
- Targets may be shared, e.g., a member can belong to many different clubs
- The source (e.g., *Club*) doesn't control the lifetime of the target (e.g., *Member*)
- Code: Fields (data members) are pointers/references to this type, and the objects are created externally

Aggregation: C++ References

```
class Club {
public:
    Club(Member& member)
        : president(member)
    {}
private:
    Member& president;
};

Club* club = nullptr;
{
    Member newpres;
    club = new Club(newpres);
}
// Does newpres still exist?
```

- Simpler syntax than pointers
- Sure (?) that the object exists
- Can only be changed in a member initialization list
- Cannot be directly part of a container (workaround: `std::reference_wrapper`)
- References to local objects whose lifetime may disappear

Aggregation: C++ Pointers

```
class Club {
public:
    Club(Member* member)
        : president(member)
    {}
private:
    Member* president;
};

Member* newpres = new Member();
Club club(newpres);
// ...
// Who deletes? Club? or client code?
```

- Much more flexible, can be changed at any time

- More complex syntax

- May be `nullptr`, so have to check before use

- More difficult to manage lifetime, *Who deletes?*

Aggregation: std::shared_ptr<>

```
class Club {
public:
    Club(std::shared_ptr<Member> member)
        : president(member)
    {}
private:
    std::shared_ptr<Member> president;
};

Club* club = nullptr;
{
    std::shared_ptr<Member> newpres(new Member());
    club = new Club(newpres);
}
// Safe delete in destructor
```

- Similar to `std::unique_ptr<>`, but allows multiple clients
- As the last client is deleted, then the pointer is deleted
- Easier to manage lifetime, as the pointer is shared
- Suggest as the first choice

Composition Can Occur Even With:

References	`Policy& policy;`
Pointers	`Policy* policy;`
Smart Pointers	`std::unique_ptr<Policy> policy;`

Composition with References

```
class Club {
public:
    Club()
        : size(0),
          policy(emptyPolicy)
    {}

    Club(int members)
        : size(members),
          policy(fullPolicy)
    {}

    ~Club() {}
private:
    int size;
    EmptyPolicy emptyPolicy;
    FullPolicy fullPolicy;
    Policy& policy;
};
```

- References to other fields

- Still composition since the class `Club` controls the lifetime of the object being referenced, e.g., `emptyPolicy` and `fullPolicy`

Composition with Pointers

```
class Club {
public:
    Club()
        : size(0),
          policy(new EmptyPolicy())
    {}

    Club(int members)
        : size(members),
          policy(new FullPolicy())
    {}

    ~Club() {
        delete policy;
    }
private:
    int size;
    Policy* policy;
};
```

- Pointers to objects created and destroyed by the class
- Typically allocated in the constructor (or later on) and deleted in the destructor
- The class `Club` controls the lifetime of `policy`
- This is still *composition*

Composition with Smart Pointers

```
class Club {
public:
    Club()
        : size(0),
          policy(new EmptyPolicy())
    {}

    Club(int members)
        : size(members),
          policy(new FullPolicy())
    {}

    ~Club() {}
private:
    int size;
    std::unique_ptr<Policy> policy;
};
```

- When the `Club` destructor is called, the `std::unique_ptr<>` handles the `delete` of the allocated memory from the objects

- This is still composition

Composition vs. Aggregation

- Prefer *Composition* to *Aggregation* as the object's lifetime is clearer

- Aggregation has particular roles in more complex relationships, especially with polymorphic data members

- Two are often confused, and terms are used interchangeably