Advanced OO Design Principles

Unit 1 - Design Fundamentals

Pratian Technologies (India) Pvt. Ltd.
www.pratian.com

PRATIAN
TECHNOLOGIES

Topics

- Classification
  - Class diagram and UML notation
    - Designing Encapsulation
  - Interface vs. Abstract class
- Relationships
  - Generalization & Realization
  - Association, Aggregation & Composition
  - Dependency
- Exercises

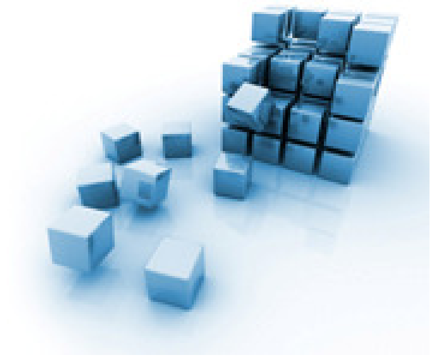
Classification - Identifying Classes

Identifying Classes

Develop an Object Model for the following requirement in an ILT (instructor-led training):

“A trainer trains many trainees on a given course. Every course has many modules – each module is comprised of different units and each unit has many topics”.

- Identify the different classes from the above problem statement



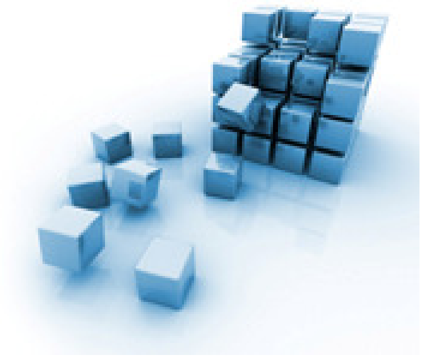
Identifying Classes

- **Recommended Class Identification Technique**
  - Focus on identifying **NOUNS (Data Driven approach)**
  - We will look at other techniques subsequently

- Trainer
- Trainee
- Course
- Module
- Unit
- Topic

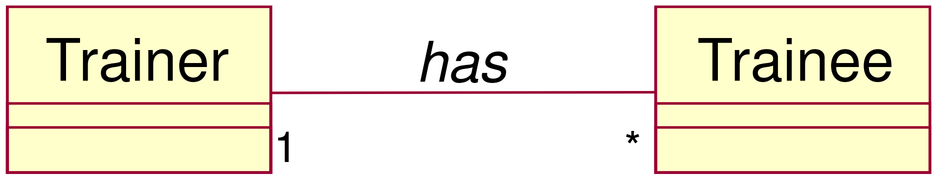
Identify the different connections (relationships) between the above classes

- Connections between classes established through Relationships
- Focus on 'Is-a' and 'Has-a' relationships

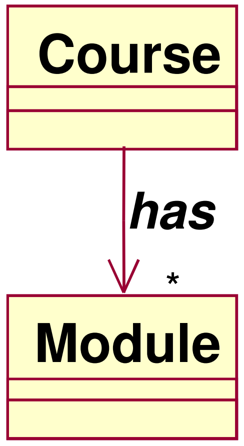


Identifying Relationships

- Trainer - Trainee
  - Trainer 'HAS' many Trainees
  - Every Trainee 'HAS' a Trainer



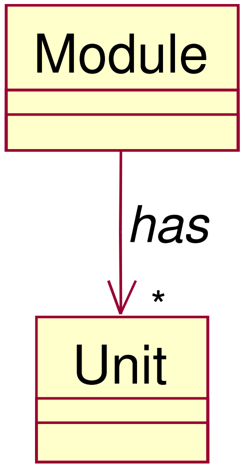
- Course - Module
  - Course 'HAS' many Modules



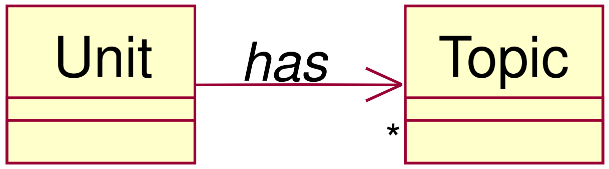
PRATIAN TECHNOLOGIES

Identifying Relationships

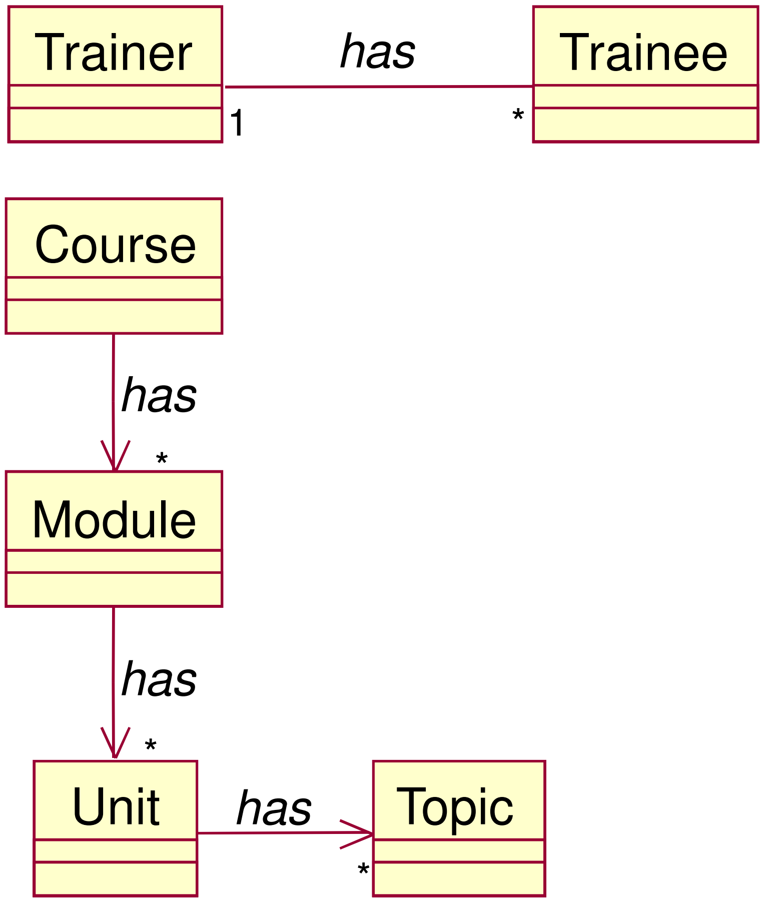
- Module – Unit
  - Module 'HAS' many Units



- Unit – Topic
  - Unit 'HAS' many Topics



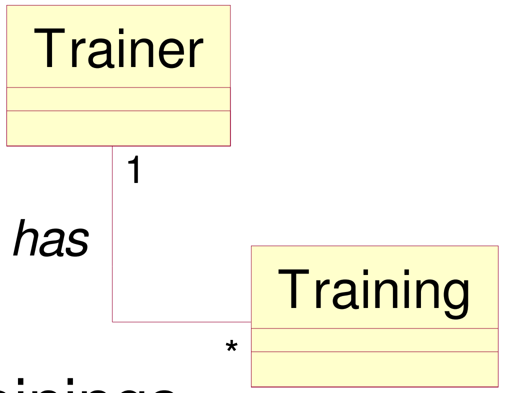
The OO Model



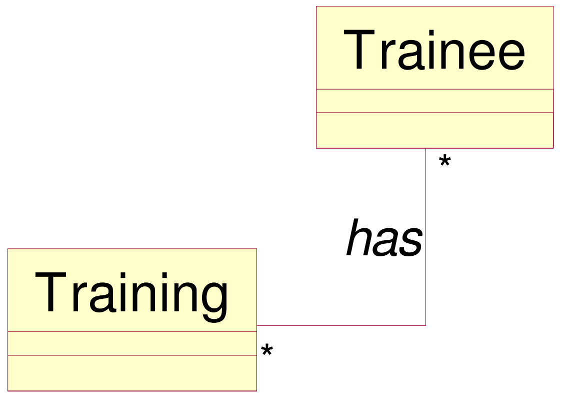
How do you relate the Trainer & Trainee to the Course?

Conceptual Entity - Training

- Trainer – Training
  - A Trainer (HAS) conducts many Trainings
  - A Training HAS a Trainer

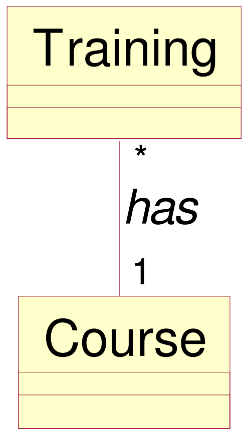


- Trainee – Training
  - A Trainee (HAS) attends many Trainings
  - A Training HAS many Trainees



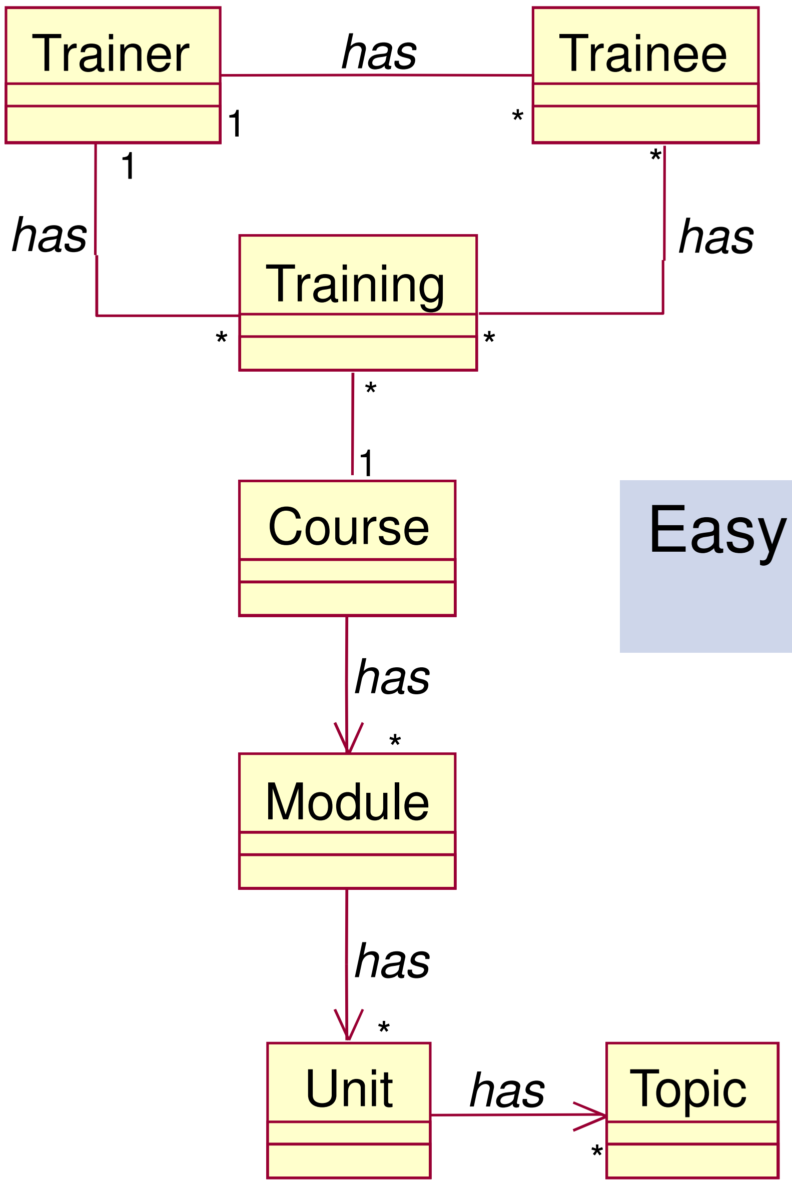
Conceptual Entity

- Training - Course
  - The Training (HAS) (is conducted for) a Course
  - A Course HAS (can have) many Trainings

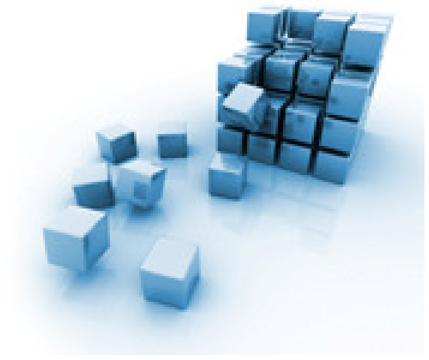


PRATIAN TECHNOLOGIES

Solution



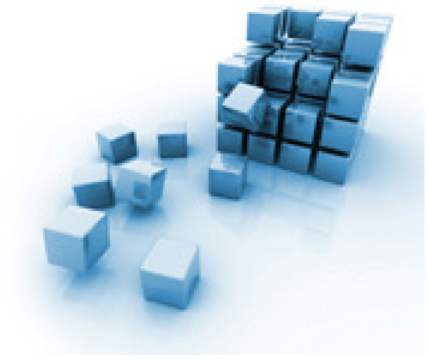
Easy to model real-world problems using the OO approach



Exercise

A company sells different items to customers who have placed orders. An order can be placed for several items. However, a company gives special discounts to its registered customers.

- Identify the different classes from the above problem statement
- Identify the different connections (relationships) between the above classes



Exercise

In the SkillAssure Assessment Framework,

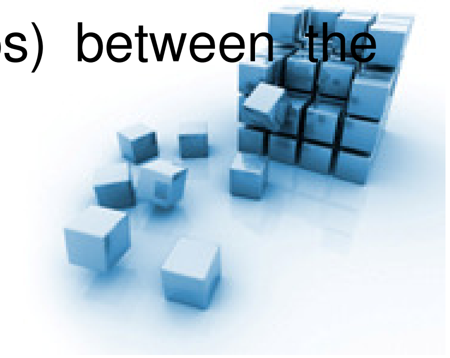
Every course can have assessments

An iteration has many courses and can also have additional assessments

The training model comprises of 4 iterations.

An assessment can be of multiple-choice type, hands-on exercise or a project.

- Identify the different classes from the above problem statement
- Identify the different connections (relationships) between the above classes

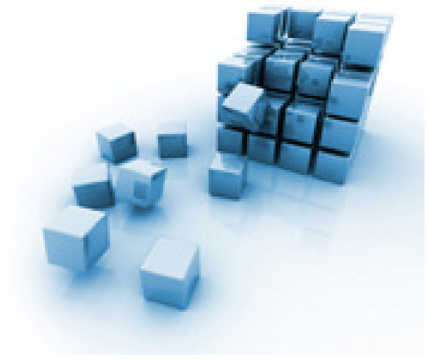


PRATIAN TECHNOLOGIES

Exercise

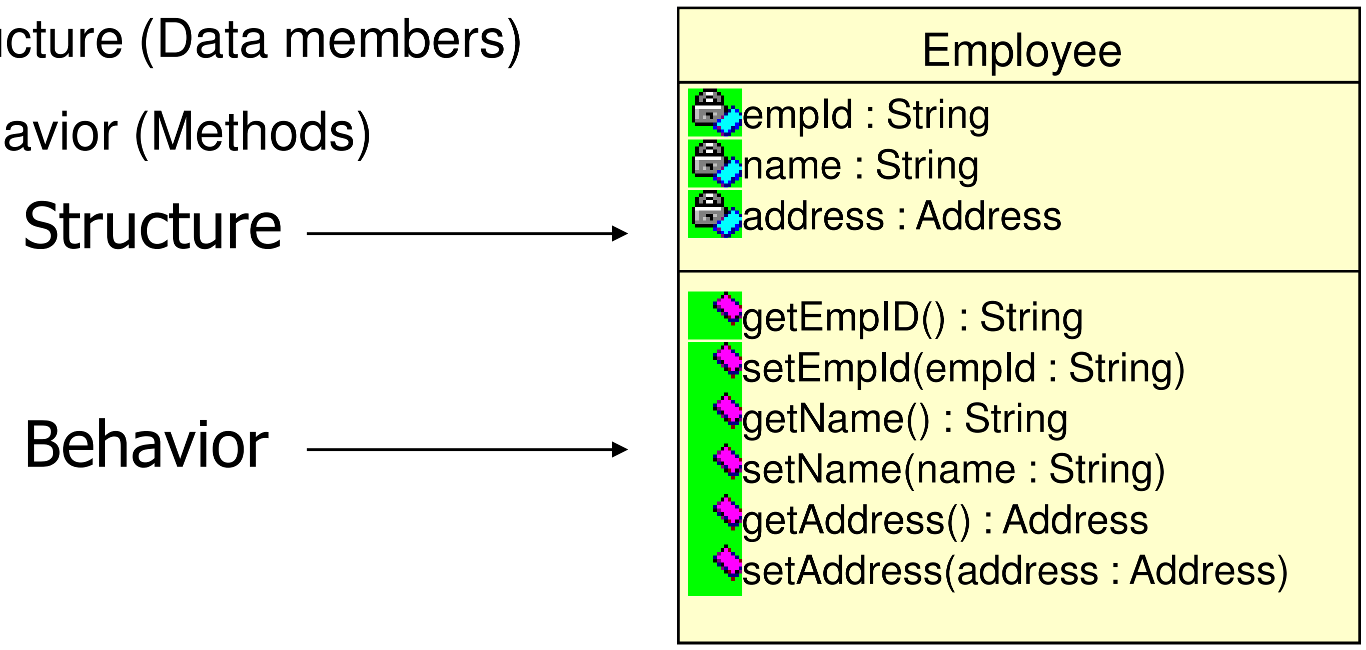
There are many programming languages. Java and C# are object-oriented programming languages. C is a procedural programming language.

- Identify the different classes from the above problem statement
- Identify the different connections (relationships) between the above classes



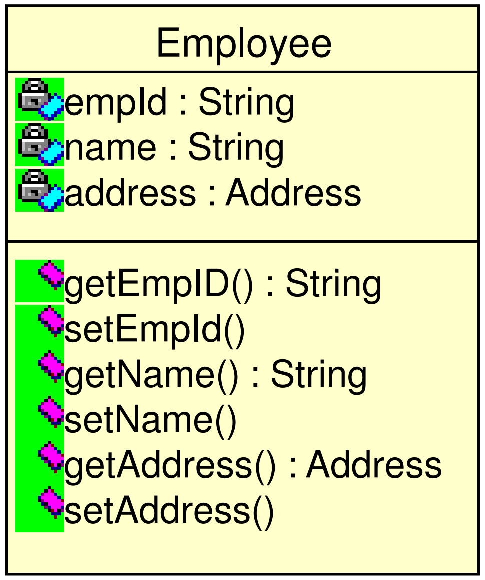
Class Diagram

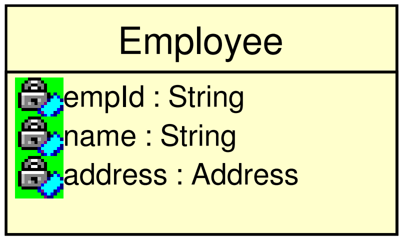
- Every class has three compartments
  - Name of the class
  - Structure (Data members)
  - Behavior (Methods)

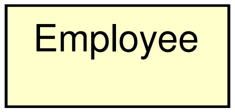


Class Diagram

- Other representations



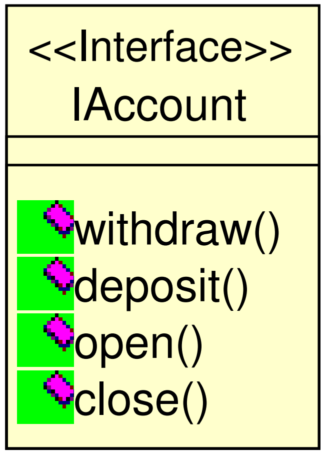




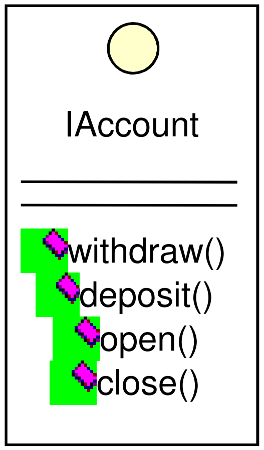
Interface

- Is an extension of the class symbol in UML
- Standard UML offers no different symbol for interface
  - Instead a stereotype is used to extend the class symbol to represent an interface
- Ideally an interface should have only abstract behavior (key abstractions) and no structure

UML Representation

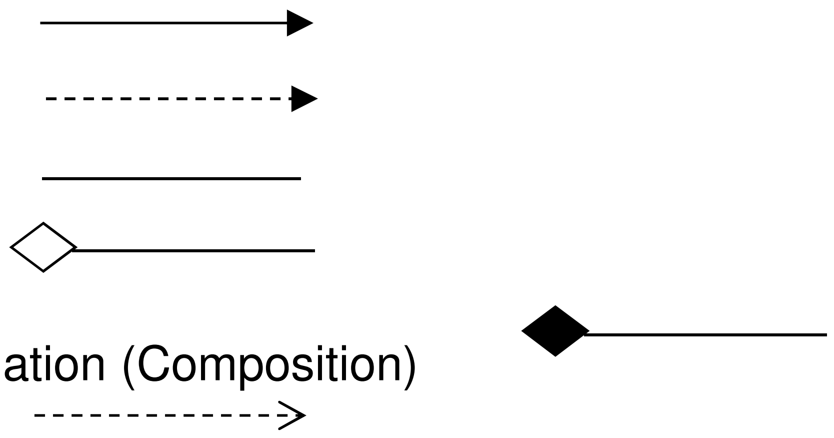


Rational Rose representation

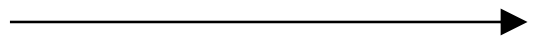


Classes and Relationships

- Six types of relationships can exist between classes
  - Generalization
  - Realization
  - Association
  - Aggregation
  - Composite Aggregation (Composition)
  - Dependency



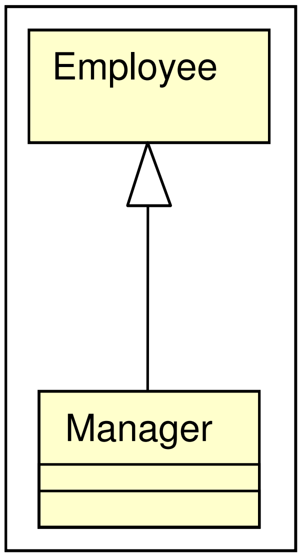
Generalization

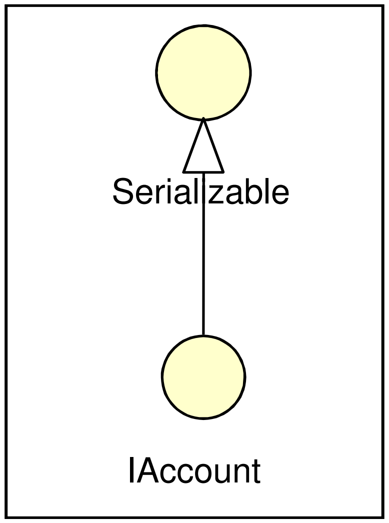
Generalization 

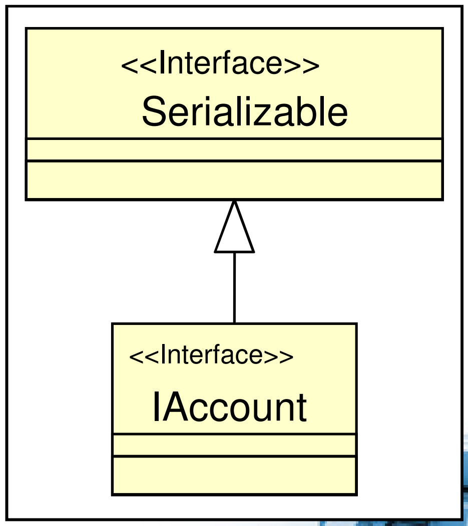
- Relationship between two classes or two interfaces
- Represented by ‘public’ Inheritance in C++
- Represented by ‘extends’ in Java

class Manager : public Employee { }

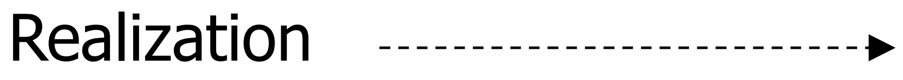
struct IAccount : Serializable { }





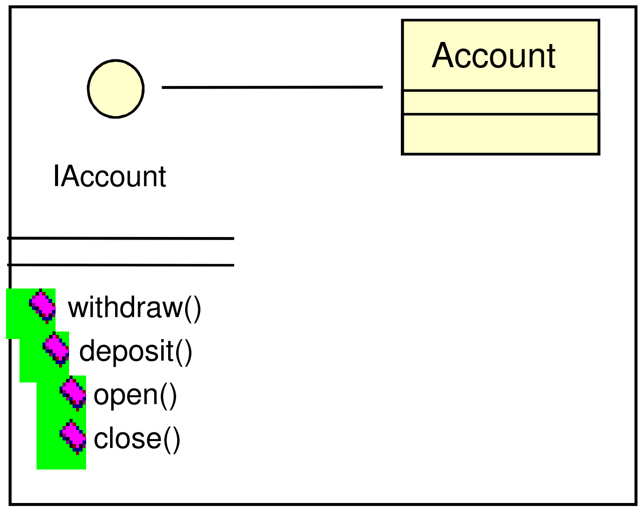


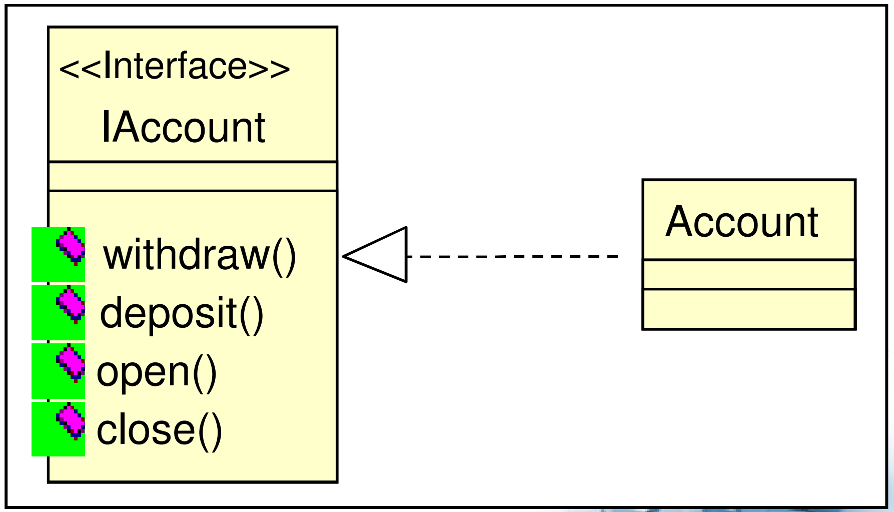
Realization



- Relationship between a class and an interface
- Represented by ‘implements’ in Java
- Represented by ‘public’ inheritance in C++
- Realizing an interface would require a class to provide an implementation for all the inherited method declarations failing which the class becomes abstract

```
class Account : public IAccount { }
```

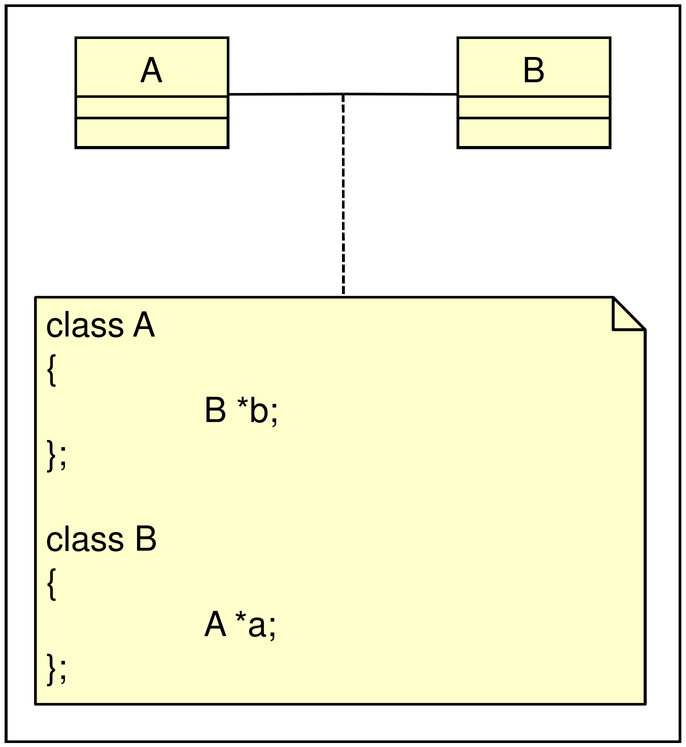




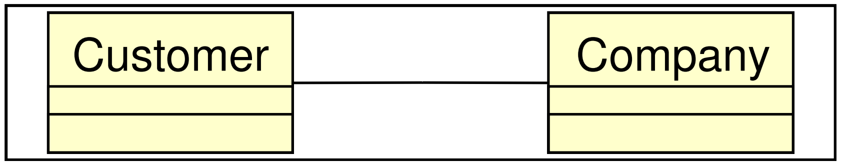
Association

Association

- 'Has-a' relationship
- Semantic relationship between two or more classifiers that involve connections among their instances



```
class A
{
        B *b;
};

class B
{
        A *a;
};
```

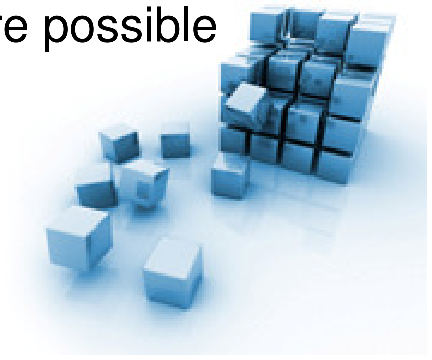


A Customer is associated with a Company is essentially a 'has-a' relationship between a Customer and a Company

```
class Customer
{
        Company* company;
};
```

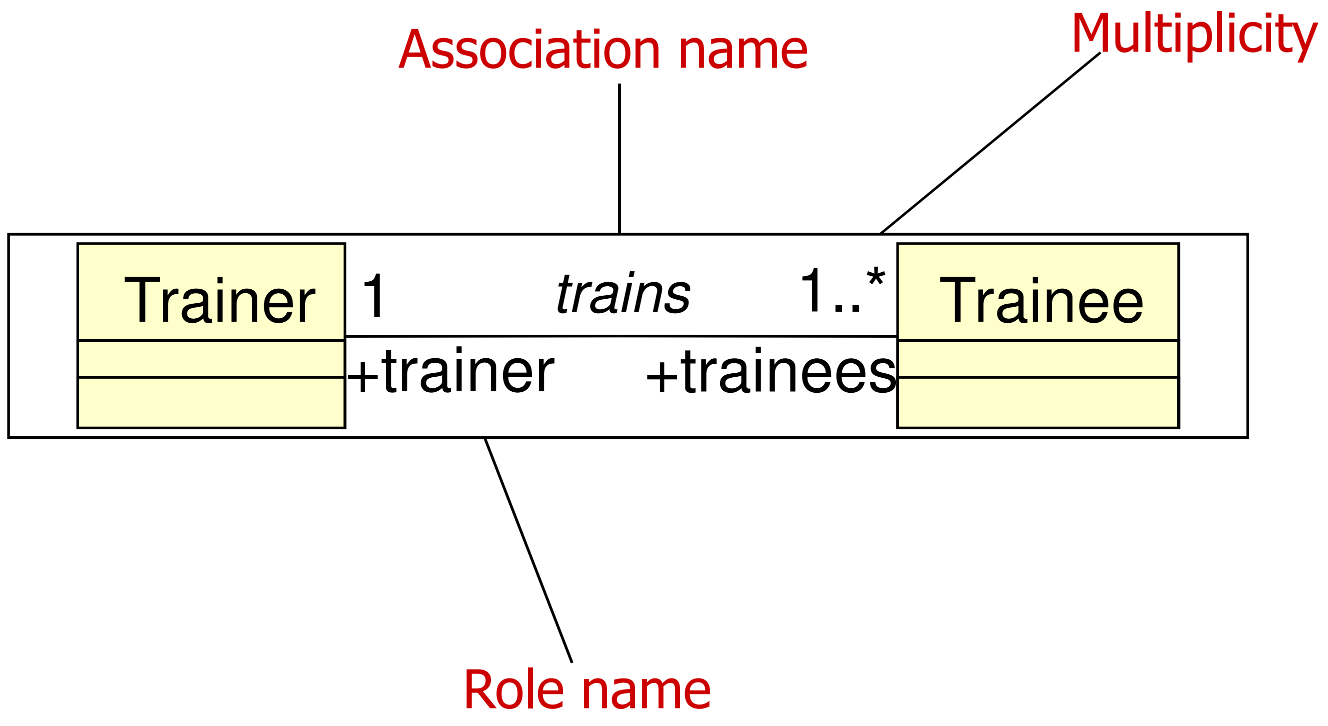
Association

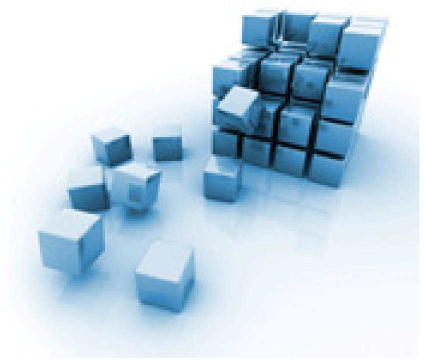
- The associations are qualified by
  - Multiplicity
    - The number of instances with which a class is associated
    - Can be 1, 0..1, *, 1..*, 0..*, 2..*, 5..10, etc.
    - Multiplicity is by default 1
  - Navigability
    - Can be unidirectional or bidirectional
    - Navigability is by default bi-directional
  - Role name
    - The name of the instance in the relationship
    - Multiple associations based on different roles are possible



Association

- Role name, navigability and multiplicity

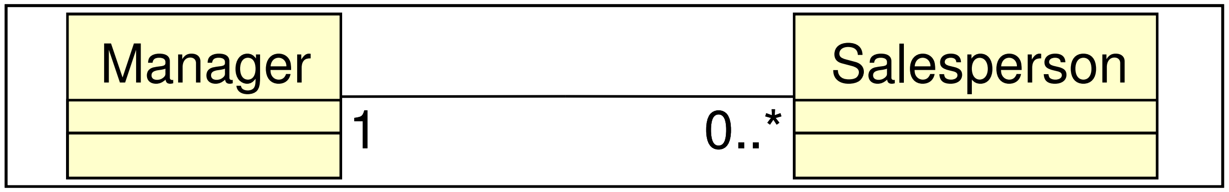


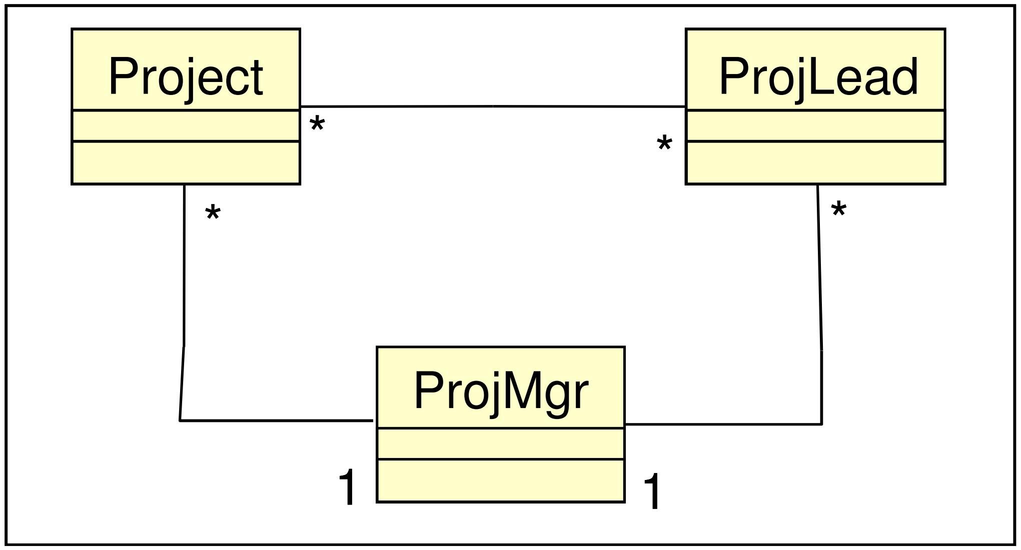


PRATIAN TECHNOLOGIES

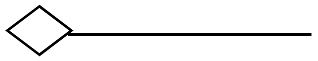
Association

- Examples

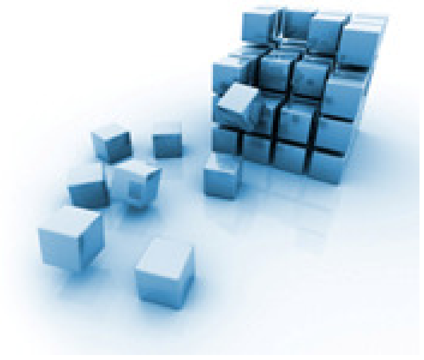




Aggregation

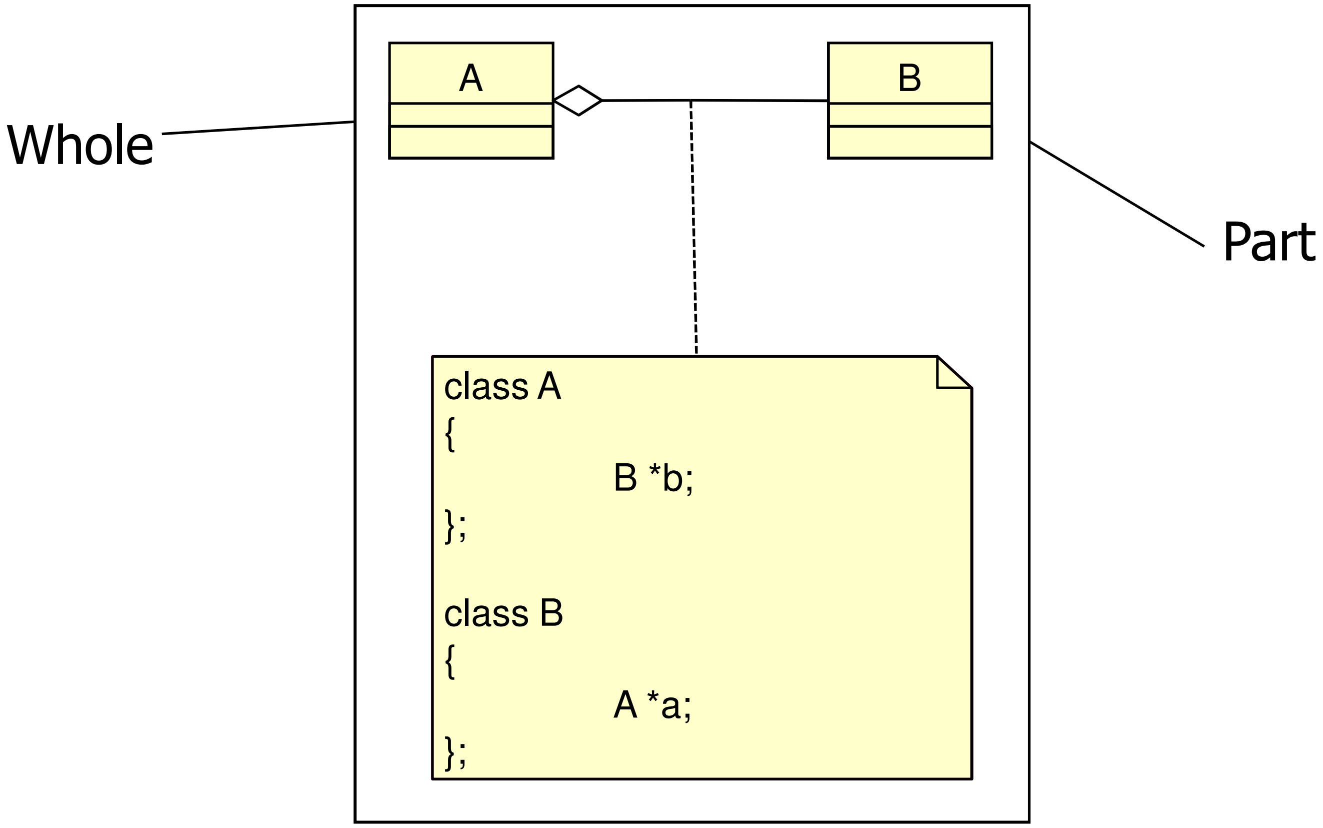
Aggregation 

- ‘Has-a’ relationship
- Is a special (stronger) form of association which conveys a whole part meaning to the relationship
  - Also known as Aggregate Association
- Has multiplicity and navigability
  - Multiplicity is by default 1
  - Navigability is by default bi-directional



Aggregation

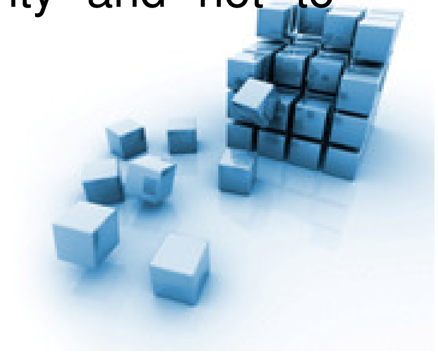
- The hollow diamond is placed towards the whole



```
class A
{
        B *b;
};

class B
{
        A *a;
};
```

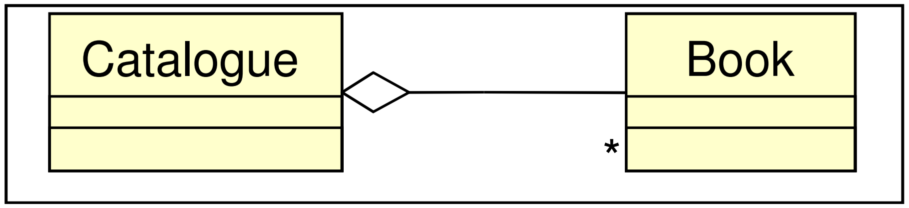
Aggregation

- So what's the difference between Association and Aggregation?
  - When it is comes to code – NOTHING!
- Aggregation is a special meaning derived out of Association based on the context
  - Whole Part
  - Lack of independent use and existence
  - Scope of verb is constrained to only 'has' or 'contains'
- **When in doubt, leave aggregation out!**
  - Association and Aggregation exist to add clarity and not to introduce ambiguity

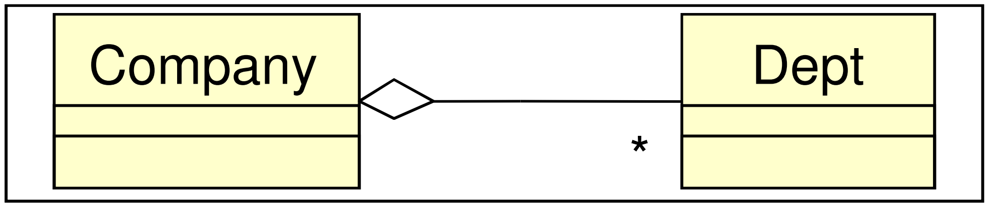


Aggregation

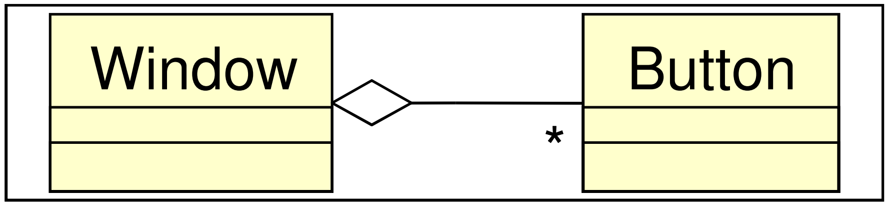
- The Whole-Part nature

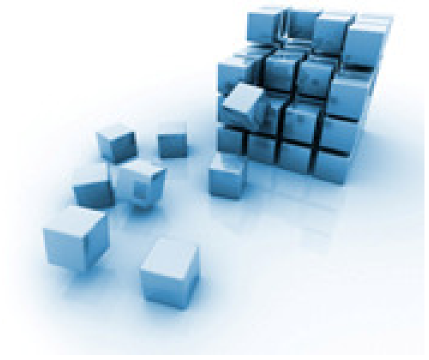


- Independent existence / use



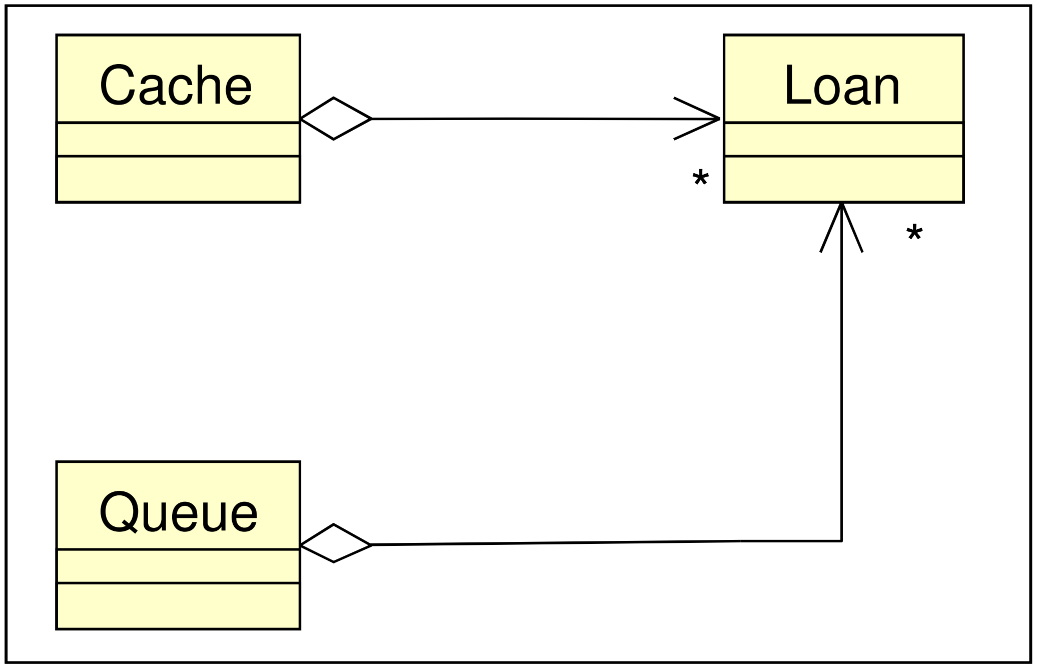
- Scope of verb is constrained to only ‘has’ or ‘contains’





Aggregation

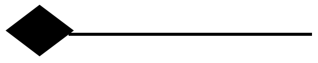
- A part can be shared between many wholes
  - Shared aggregation



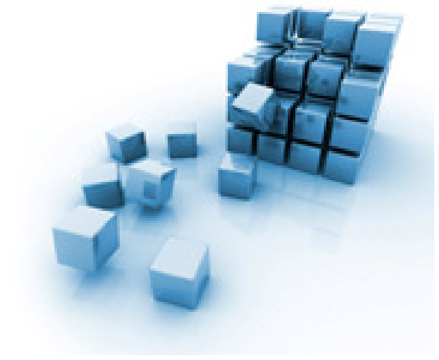
PRATIAN TECHNOLOGIES

Composite Aggregation

Composite Aggregation

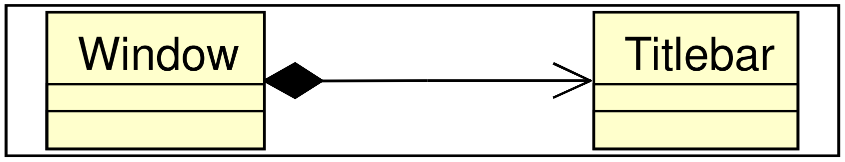


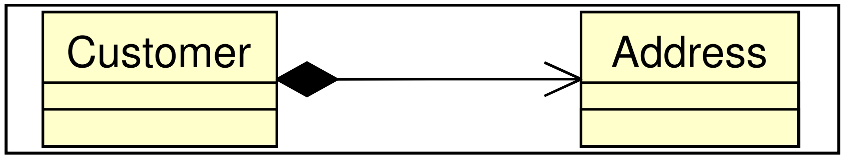
- 'Has-a' relationship
- Is a stronger form of aggregation
  - Also known as Composition or Containment By Value
- A part cannot be shared between many wholes
  - Unshared aggregation
- Explicit lifetime control
- Exclusive ownership
- Has multiplicity and navigability
  - Multiplicity at the whole end should always be 1
  - Navigability is by default bi-directional

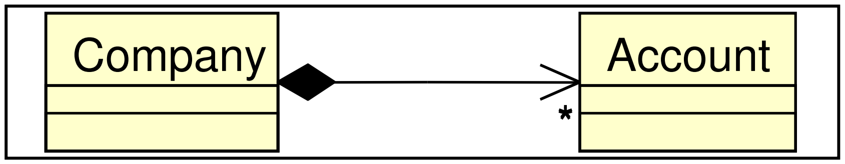


Composite Aggregation

- Examples



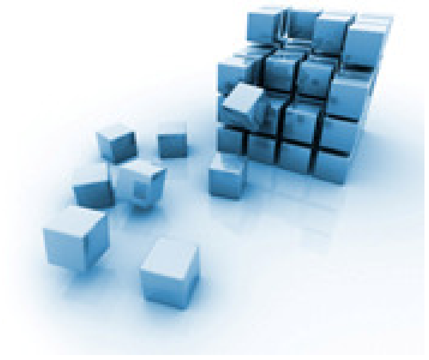




Dependency

Dependency - - - - - - - - - ->

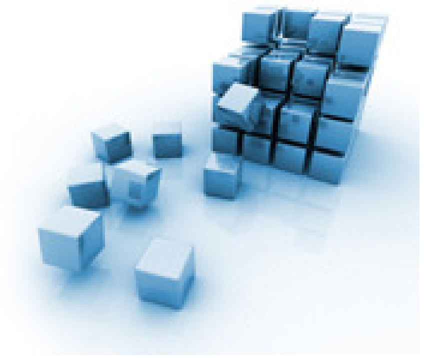
- 'Uses' relationship
- Behavioral relationship
  - Loose coupling
  - Has no impact on class structure (data members)
- A class references another class only within its methods for the purpose of:
  - Invoking a static method
  - Local instantiation
  - Formal argument use
  - Return type



Dependency

- **Invoking a static method**

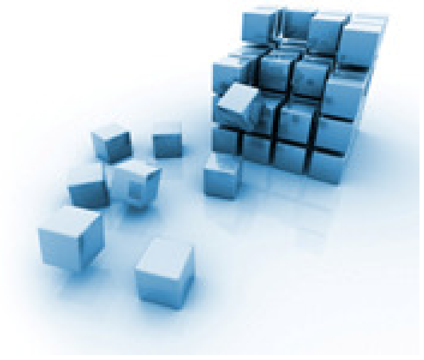
```
class B {
public:
        static void method1() { }
};

class A {
public:
         void f1() {
                  B::method1();
        }
};
```



Dependency

- **Local instantiation**

```
class B {
public:
        void method2() { }
};

class A {
public :
        void f1() {
                B *b1 = new B;
                b1->method2();
        }
};
```



Dependency

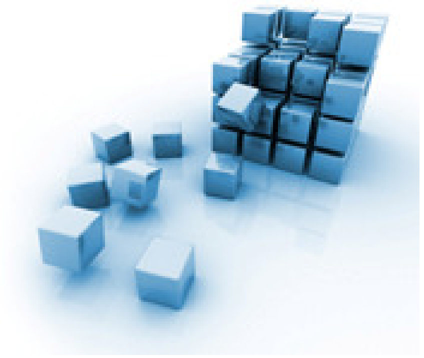
- **Formal argument use**

```
class B {
public:
        void method2() { }
};


class A {
public:
        void f1(B &b) {
                b1.method2();
        }
};
```

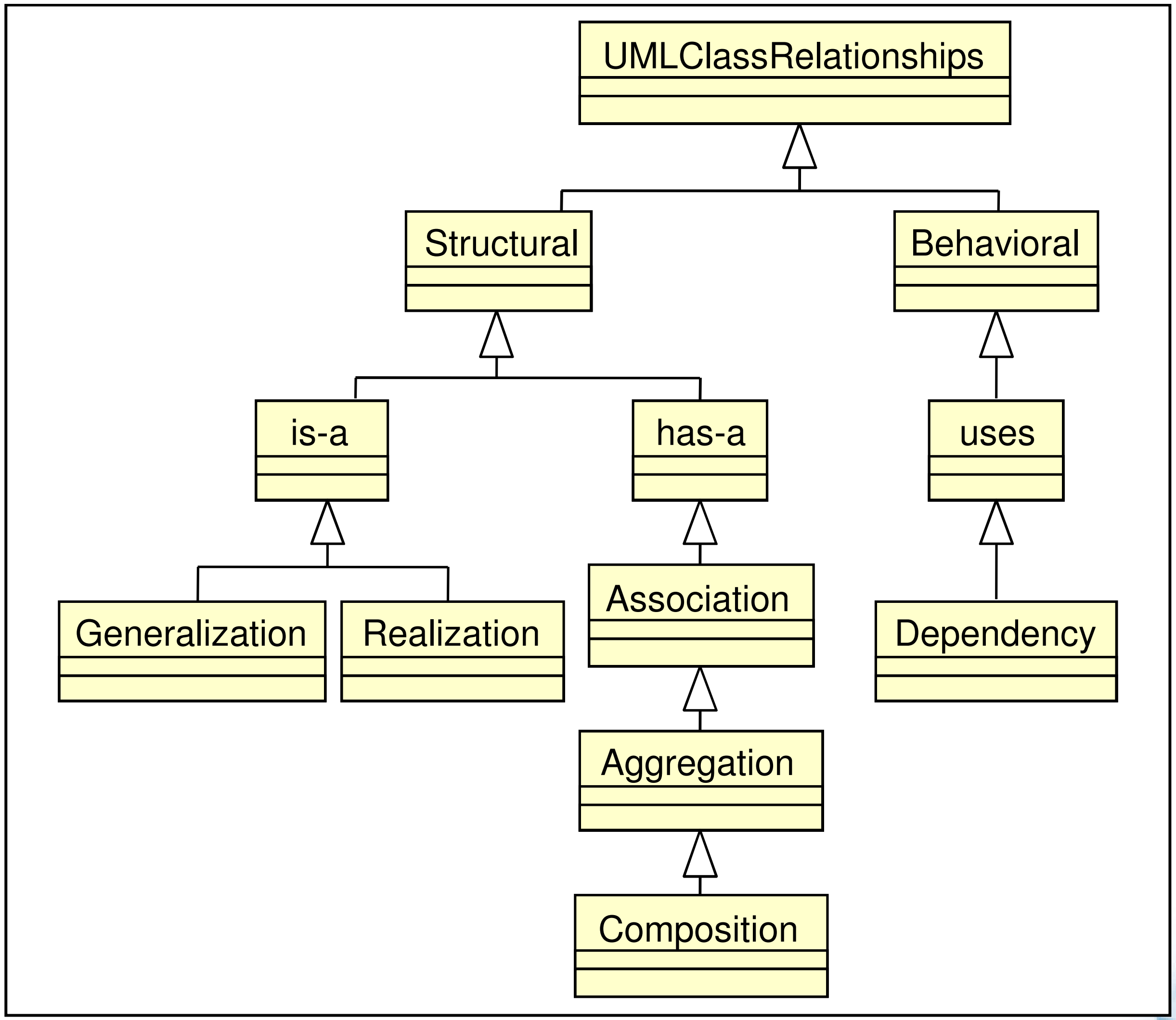
Dependency

- **Return type**

```
class B { };

class BFactory {
public:
        B* create() {
                // check some condition
                return new B;
        }
};
```



Relationship Meta Model



Question Time

Please try to limit the questions to the topics discussed during the session.

Thank you.