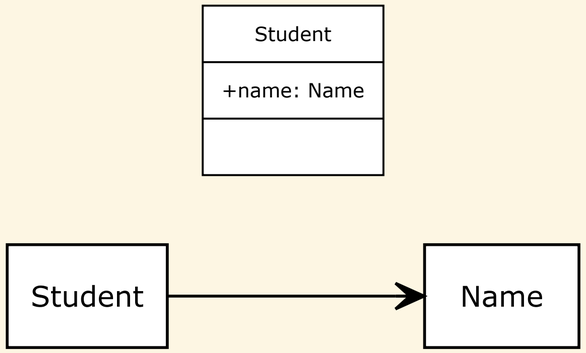
Object-Oriented Programming

UML Class Relationships

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Properties: Attributes & Association



- 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

- Is String a *class* or a *datatype*?

DataType	Class
Primitives	Objects
Methods are familiar	Methods are not familiar
Operators do not change	Operators can change
New developers know precisely what you mean	New developers have only a vague idea
Tend to be scalar (single value)	Tend to be complex types
Equivalent values ⟹ same identity	Two distinct objects can have the same value
Probably not a base for generalization (inheritance)	Maybe a base for generalization (inheritance)

Class Relationships

- Object-Oriented Design: *What classes make up this system?*

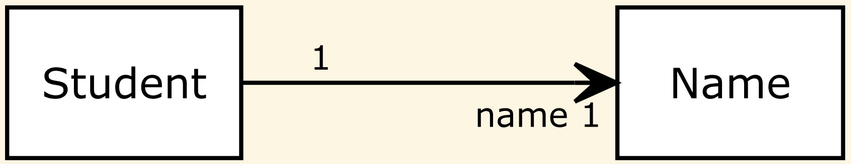
- Multiple classes ⇨ relationships between classes

- Existence and kinds of relationships between classes are the central part of a UML Class Design

UML Class Relationships

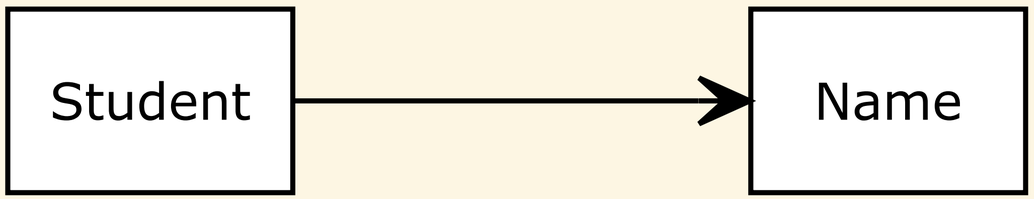
Relationship	Description
Dependency	*uses a*
Association	*has a*
Aggregation (Association)	*has a*
Composition (Association)	*has a*
Generalization	*is a*

Relationship



- directionality
- multiplicity
- labels

Association



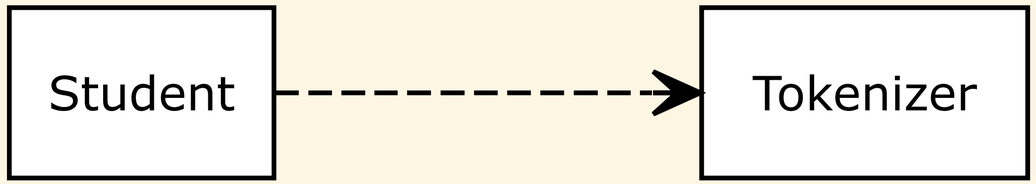

```cpp
class Student {
private:
    Name name;
};

class Name {};
```

Association

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

class Name {};
```

- One class has a(n) element of another class
- Represented in UML by a solid line with an arrow

Dependency



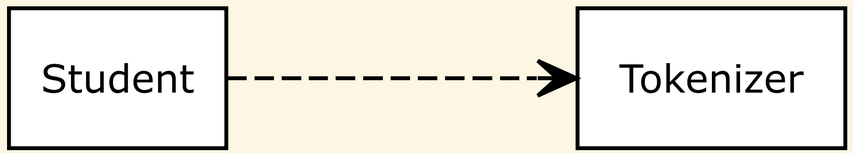

```
class Student {
    void importName(Name name) {
        Tokenizer t;
        // ...
    }

    void import(Tokenizer t) {}
private:
    Name name;
};

class Tokenizer {};
```

Dependency

```
class Student {
    void importName(Name name) {
        Tokenizer t;
        // ...
    }

    void import(Tokenizer t) {}
private:
    Name name;
};

class Tokenizer {};
```

- Class used as a parameter or local variable only (**not** a field/data member) results in a dependency

Dependency Discussion

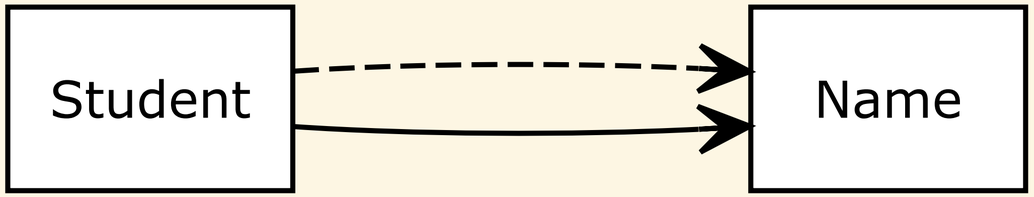


- Represented in UML by a dashed line with an arrow

- Changes to the definition of the supplier/source (e.g., Tokenizer) may cause changes to the client/target (e.g., Student)

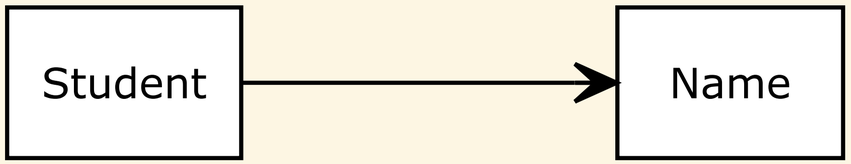
- Dependencies are weaker than Associations and **therefore preferred**

Dependency & Association




```
class Student {
    void importName(Name name) {
        Tokenizer t;
        // ...
    }

    void import(Tokenizer t) {}
private:
    Name name;
};

class Tokenizer {};
```

Dependency & Association



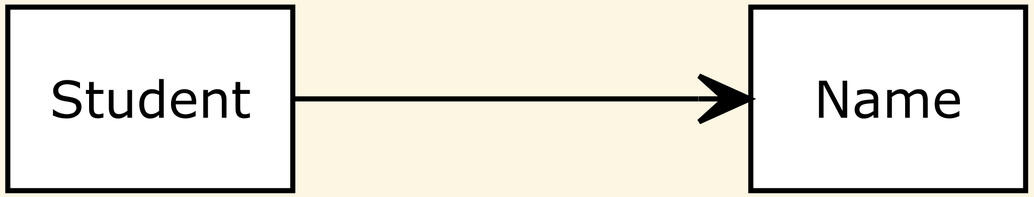
- Show the strongest relationship only

Strong Relationships

Strength	UML Relationship
Strongest	Generalization (Inheritance)
Stronger	Composition
Strong	Aggregation
Strong	Association
Weak	Dependency
Weakest	Realization (Implementation)

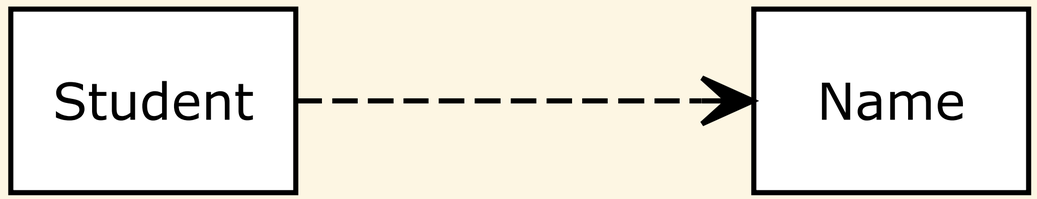
- Can rank the strength of relationships
- *Association* is a stronger relationship than *Dependency*
- Strong relationships are **not good**
- Any relationship implies *coupling*, overly *strong coupling* is not good design
- The stronger the relationship, the more constrained and rigid the design is
- Prefer *Dependency* over *Association*

Scenario: Original Design




```
class Student {
    void importName(Name name) {
        Tokenizer t;
        // ...
    }

    void import(Tokenizer t) {}
private:
    Name name;
};

class Tokenizer {};
```

Scenario: Name removed



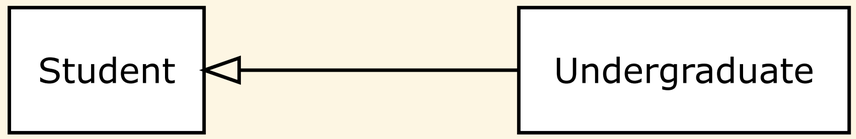

```
class Student {
    void importName(Name name) {
        Tokenizer t;
        // ...
    }

    void import(Tokenizer t) {}
private:
//    Name name;
};

class Tokenizer {};
```

Substitutability: Liskov Substitution Principle

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then, $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

- We can substitute type *S* objects for type *T* objects
- Not just interface (syntax), but behavior (semantics)
- Typically referred to as an "is a" relationship
- [Liskov,Wing'94]

Generalization



- Expression of **is a** relationship: Substitutability
- Maps directly to inheritance in most OOP languages
- Subclass/subtype/derived class is a *generalization* of superclass/supertype/base class
- Highest semantically defined relationship
- Represented in UML by a solid line with a hollow arrowhead

Inheritance

```
class Student {};

class Undergraduate : public Student {};

void pass(Student& student) {}

Student student;
pass(student);

Undergraduate ugrad;
pass(ugrad);
```

- In C++, Generalization is public inheritance

Problems with Generalization

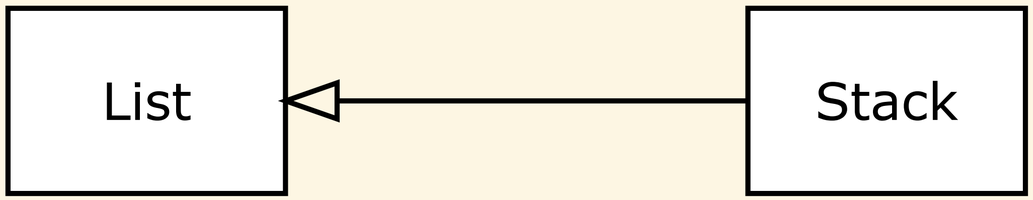
The purpose of generalization is to solve design problems. If you don't have a design problem, don't use generalization.

- Inheritance and inheritance hierarchies are more challenging to get right than realized

- The developer needs to create classes with inheritance in mind. Most classes are not designed for this, e.g., C++ standard library

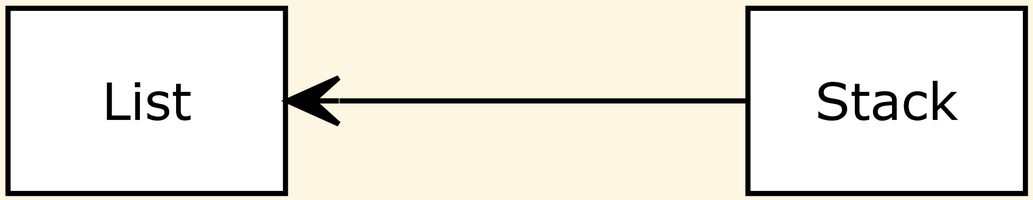
- Often overused. In general, prefer *association* over *generalization*

Substitutability Violation #1



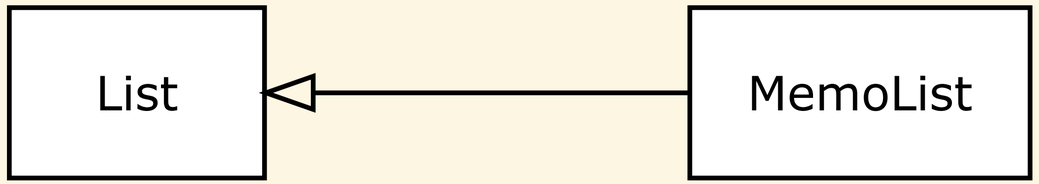

```
class List {
public:
    void push_back(int);
    void push_front(int);
};

class Stack : public List {
public:
    void push(int);
    int pop();
};

List l;
l.push_front(4);
l.push_back(4);

Stack s;
s.push(1);
s.pop();
s.push_back(4);
s.push_front(5);
```

Violation Solution #1



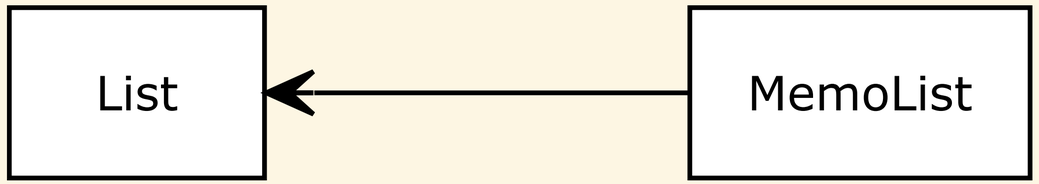

```
class List {
public:
    void push_back(int);
    void push_front(int);
};

class Stack {
public:
    void push(int);
    int pop();
private:
    List list;
};

List l;
l.push_front(4);
l.push_back(4);

Stack s;
s.push(1);
s.pop();
// s.push_back(4);
// s.push_front(5);
```

Substitutability Violation #2



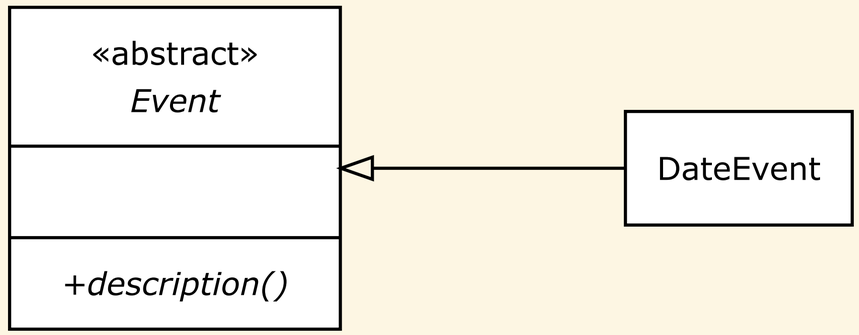

```
class List {};

class MemoList : public List {};
```

Violation Solution #2




```
class List {};

class MemoList {
private:
    List list;
};
```

Abstract Classes/Methods



- Abstract classes do not support direct instantiation, e.g., you can create a *DateEvent* object but not an *Event* object

- An abstract class typically has one or more abstract methods. An abstract method is a method that is declared but not defined in the class, and a derived class must provide a definition.

- Displayed (both classes and methods) as Event or with a label *{abstract}*, or abbreviated label *{A}*

Interfaces

- All operations are public, and no operation has a method body
- Cleanly models interfaces in Java, C#, COM, CORBA
- Indicated as keyword «interface», or with a label *{interface}*, or abbreviated label *{I}*
- In this case, *inheritance* means *implementation*
- We can also have a dependency relationship with an interface