**Object-Oriented Programming (OOP)**

- 3 fundamental concepts of OOP:
  - Classes and instances: Encapsulation of code and data
  - Inheritance
  - Polymorphism and dynamic method dispatch
- Classes provide a type system
  - Type conformance issues
  - Method signatures provide a well-defined interface
    (at least at the syntactic level)
- Is OOP the ideal platform for implementing software components?

**Interfaces**

- Interfaces = means by which components connect:
  - Set of named operations that can be called by clients
  - With specified semantics
  - To be respected both by component provider and by client
- Direct interfaces
  - Procedural interfaces of traditional libraries
  - Dynamic method dispatch
- Indirect interfaces
  - Object interfaces
  - Dynamic method dispatch – potentially routed to a third component…

**Contracts** [Meyer'88]

- A contract is the set of requirements that a use of an object has on a declaration of its class.
- Functional specification for each module before coding
- Class conformance / Syntactic substitutability [Liskov'92]:
  - A module Y is conformant to a module X if it can safely replace X in the system.
  - A subclass Y is conformant to a superclass X if all objects of Y can safely replace all objects of X.
  - An interface is a contract between the client of the interface and the provider of the implementation

**Syntactic substitutability**

- Given a declaration of a variable or parameter of type X:
  - X x;  
  - foo (...) {...

any instance of a class Y that is a descendant of X may be used as the actual value of x without violating the semantics of the declaration of its use:

- Y y;  
- x = y;  
- call foo (..., y, ...);

- Because an Y instance understands all methods that an X instance must have.
- But x.bar(...) and y.bar(...) do not necessarily call the same method! (polymorphism)
  - syntactic, but not semantic substitutability
- X is a supertype of Y (e.g., a superclass)
- Y is a subtype of X (e.g., a subclass)
- Also called Liskov substitutability (attributed to B. Liskov, MIT)

**Interfaces in UML**

- Component diagram
  - Provided interfaces
  - Required interfaces

**Wiring of components**

- Dictionary
  - spellCheck
  - synonyms
  - supplement

- WordProcessor

- Provided interfaces
  - spellCheck
  - synonyms
  - supplement

- Required interface

**Dictionary**

- WordProcessor
  - spellCheck
  - synonyms
  - supplement
Design by contract: Interfaces and OOP

- Subtype (subclass) as "subcontractor":
  - Conformance rules for polymorphic method interfaces in OO languages
  - Provider of a subclass (B) may expect less than the contract guarantees
- For input parameters: Contravariance of types
  - Provider of subclass can substitute a supertype (e.g., superclass W) for a parameter type -> more general type accepted, overfulfills contract
- For output parameters: Covariance of types
  - Provider of subclass can substitute a subtype (e.g., subclass Z) for a result type -> more specific type returned, overfulfills contract
- Remark: Specialization as inheritance may break contravariance rule
  - Workaround may require downcasts with dynamic type checking

```
class A {   Y foo ( X x ) /*contract*/ {...}  ... }
class B extends A {  Z foo ( W w ) {...};  ...  }
class Z extends Y {  ...  }
class X extends W {  ...  }
```

Background

- Covariance: arguments, return values, or exceptions of overriding methods can be of subtypes of the original types.
- Contravariance: arguments, return values, or exceptions of overriding methods can be of supertypes of the original types.
- Invariance: arguments etc. have to be of exactly the same type.

### Background

#### Covariance

- Arguments, return values, or exceptions of overriding methods can be of subtypes of the original types.

#### Contravariance

- Arguments, return values, or exceptions of overriding methods can be of supertypes of the original types.

### Background

#### Invariance

- Arguments etc. have to be of exactly the same type.

```
interface View {
  Model getModel();
}
interface TextView extends View {
  TextModel getModel();
}
interface GraphicsView extends View {
  GraphicsModel getModel();
}
```

Contracts – beyond type conformance

- Semantic substitutability = conformant types = ...
- Hoare triplets: (precondition) operation (postcondition) [Hoare'69]
- Preconditions of an operation:
  - True on invocation
  - Callee’s / provider’s requirements
- Postconditions of an operation:
  - True on return
  - Callee’s / provider’s promise to caller / client
- May be formulated e.g. in UML-Object Constraint Language OCL
- "Demand no more, provide no less":
  - Contract precondition must imply provider’s precondition
  - Provider’s postcondition must imply contract postcondition

```
interface TextModel {
  int max();     // maximum length this text can have
  int length();  // current length
  char read ( int pos );  // character at position pos
  void write ( int pos, char ch );  // insert ch at pos
  // [ len: int, txt: array of char ::
  //  pre len := this.length();
  //  (all i: 0<=i<len:  txt[i] := this.read( i ))
  //  len < this.max() and 0 <= pos <= len
  //  post this.length() = len + 1
  //  (all i: pos<i<this.length():  this.read( i ) = txt[i-1] )
  //  and (all i: pos=0-pos=txt.length(): this.read( i ) = txt[i-1] )
}
```

Example with pre- and postconditions

```
interface TextModel {
  int max();     // maximum length this text can have
  int length();  // current length
  char read ( int pos );  // character at position pos
  void write ( int pos, char ch );  // insert ch at pos
  // [ len: int, txt: array of char ::
  //  pre len := this.length();
  //  (all i: 0<=i<len:  txt[i] := this.read( i ))
  //  len < this.max() and 0 <= pos <= len
  //  post this.length() = len + 1
  //  (all i: pos<i<this.length():  this.read( i ) = txt[i-1] )
  //  and (all i: pos=0-pos=txt.length(): this.read( i ) = txt[i-1] )
}
```
Syntactic FBCP example (cont.)

Simpler example of virtual method call, also in C++:

```cpp
class B { // base class
    public:
        virtual void foo() {} ...
        virtual void bar() {} ...
};

class C : B { // subclass of B
    virtual void foo() {} ...
    virtual void bar() {} ...
    virtual void m() {} ...
    virtual void n() {} ...
};
```

Solution 1: (IBM SOM)

- Ideally, recompilation should not be necessary in case of purely syntactic changes of superclasses' interfaces.
  - Example: (refactoring)
    - Base class method moves upwards in the class hierarchy
      - No syntactic change (i.e., in method's signature)
      - Method dispatch table entries change
      - Compiled old subclass code may access wrong/invalid locations

Solution 2: (Java VM)

- Generally look up all virtual methods at run time, even if they could be bound statically e.g. after analysis

Fragile Base Class Problem

- Superclasses (e.g., system library classes) may evolve
  - Advantage: bugfixes visible to all subclasses. But...
- Syntactic Fragile Base Class Problem
  - Binary compatibility of compiled classes with new binary releases of superclasses
  - "release-to-release binary compatibility"
  - Ideally, recompilation should not be necessary in case of purely syntactic changes of superclasses' interfaces, e.g.:
    - Methods may move upwards in the class hierarchy
    - Methods may be removed, replaced, added...
- Semantic Fragile Base Class Problem
  - How can a subclass remain valid (keep its semantic contract) if functionality inherited from the superclass evolves?
Semantic Fragile Base Class Problem

- Change of inherited functionality in base class may break subclass’s correctness (contract)

Class Bag{
  char [] bag;  ...
  void add ( char c ) { ... }
  void addAll ( char[] ac, int n ) {
    for (int i=...) self.add( ac[i] ); ...
    int cardinality() { return ... }
}

Class CountingBag: Bag{
  int counter;  ...
  void add ( char c ) {
    counter++;
    super.add( c ); }
  int cardinality() { return counter; }
}

Mixins and View-Based Composition

- Replace implementation inheritance by object composition
- A core component is extended by a view component
- Mixin: class fragment used for deriving a subclass
- Class vs. object level, static vs. dynamic

Variations on this topic:
- Mixin-Based Inheritance
- IBM SOM
- CoSy generated access layer to IR
- EJB and Corba CCM Containers + Interfaces
- State-Guttag transformation
- Subject-Oriented Programming
- Object Composition
- AOP and Invasive Software Composition

Summary

- Software components need well-defined interfaces and encapsulation
- Interfaces and Design by Contract
  - Syntactic substitutability, Covariance and Contravariance
  - Operations, pre- and postconditions
  - "Demand no more, provide no less"
- OOP is not the silver bullet for component-based software engineering
- Classes are an overloaded concept:
  - Type (-> super/subtype conformance), interface/encapsulation, implementation inheritance, interface inheritance, object instantiation
  - Implementation inheritance and dynamic method dispatch break encapsulation (is white-box reuse; but components are "black boxes")
  - Contravariance problem for input parameters
  - Fragile base class problem
- Possible solutions/workarounds (not perfect either): Tamed inheritance by Mixins / View-based Composition / Object Composition / SOP / AOP / ...

Further reading

- B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, Nov. 1994
- IBM SOP: www.research.ibm.com/sop/

Homework exercise

- Read Chapters 5, 6, 7 in the Szyperski book
- Summarize with your own words and examples the main obstacles to component-based software engineering that are imposed by OOP