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
  - Directly (explicitly) provided by a component
  - Static method dispatch

- Indirect interfaces
  - Object interfaces
  - Provided by the objects instantiated from the component
  - Dynamic method dispatch – potentially routed to a third component...

  Procedural interface may be modeled as object interface for a static object
  within the component (singleton)

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.
  Or: such a subclass fulfills the contract of the superclass.
  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:
  \[ X \ x; \text{ or } \ foo \ ( \ldots, X \ x, \ldots ) \ (\ldots) \]
  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:
  \[ Y \ y; \text{ or } \ call \ foo \ ( \ldots, y, \ldots ); \]
  - 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) \( \Rightarrow \) 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)
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.

Covariance example

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

Contravariance example

```
interface View {  
    ...  
    void setModel ( Model m );    // is this a good idea ?  
}  
interface TextView extends View  
    ...  
    void setModel ( TextModel m );   // ???  
}  
interface GraphicsView extends View  
    ...  
    void setModel ( GraphicsModel m );  // ???
```

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

```
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
}  
```

<table>
<thead>
<tr>
<th>exam p</th>
<th>loa s o d</th>
</tr>
</thead>
</table>

Demanding a TextModel as input parameter type would break the contract set by the base class View.
Syntactic FBCP example (cont.)

```cpp

Syntactic Fragile Base Class Problem

- 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 1: (IBM SOM)
  Initialize method dispatch tables at load time

- Solution 2: (Java VM)
  Generally look up all virtual methods at run time, even if they could be bound statically e.g. after analysis

Syntactic FBCP example: C++

```cpp

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-Guttage 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 / ...

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
- Write a toy example program in C++ that demonstrates the Syntactic Fragile Base Class Problem

Further reading

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