**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

**Interfaces in UML**

- Component diagram

**Component implementations evolve...**

- New versions
- Specialized versions
- Subcontractor implementations...

- How can the provider’s implementation evolve without breaking any possibly existing client code, i.e., keep the contract?
  - Syntactically substitutable: The types still match, i.e., the compiler / linker will not complain when recompiling/relinking with client code
  - Semantically substitutable: The new component version still behaves at run time in the same / a compatible way (how to define that?) for old client codes
**Terminology for Direction of Varying Method Parameter Types in Subcontractors**

- **Covariance:**
  New parameter types, return value types, or exception types of replacing methods are proper subtypes (more specific) of the corresponding original types in the replaced method.

- **Contravariance:**
  New parameter types, return value types, or exception types of replacing methods are proper supertypes (more general) of the corresponding original types in the replaced method.

- **Invariance:**
  New parameter types etc. are of exactly the same type as in the replaced method.

Where is covariance or contravariance allowed at subclassing?

---

**OOP: Syntactic substitutability rules for a subclass as a subcontractor / new version**

- **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** (formal parameters):
  - Covariance and invariance is possible – but not contravariance
  - Provider of subclass can substitute a supertype (e.g., superclass W) for parameter type -> more general type accepted, overfulfills contract

- **For output parameters** (return values, thrown exceptions):
  - Covariance and invariance is possible – but not contravariance
  - Provider of subclass can substitute a subtype (e.g., subclass Z) for a result type -> more specific type returned, overfulfills contract

- **Workaround may require downcasts with dynamic type checking**

---

**Covariance example**

```java
interface View {
    ...  
    void setModel ( Model m ); // is this a good idea ?
}
```

However, a TextView object needs a TextModel object as its model, and a GraphicsView needs a GraphicsModel. But covariant change for input parameters would not be safe:

```java
interface TextView extends View {
    ...
    void setModel ( TextModel m ); // ???
}
```

Demanding a TextModel as input parameter type would break the contract set by the base class View.

---

**Contracts – beyond type conformance**

- **Semantic substitutability = conformant types + … ?**
- **Hoare triplets:** (precondition) operation (postcondition) [Hoare'69]

  - **Preconditions** of an operation:
    - True on invocation
    - Caller's / provider's requirements

  - **Postconditions** of an operation:
    - True on return
    - Caller'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

---

**A closer look: Syntactic substitutability**

- **Given a declaration of a variable or parameter of type X:**
  
  X x; or foo ( …, X x, … )

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

  - Y y; ...
  - X x; ...

  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 must be same or a supertype of Y (e.g., a superclass)

  Y must be same or a subtype of X (e.g., a subclass)

- **Also called Liskov substitutability** (attributed to B. Liskov, MIT)

---

**Contracts – beyond type conformance**

- **Semantic substitutability = conformant types + … ?**
- **Hoare triplets:** (precondition) operation (postcondition) [Hoare'69]

  - **Preconditions** of an operation:
    - True on invocation
    - Caller's / provider's requirements

  - **Postconditions** of an operation:
    - True on return
    - Caller'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

---

**Contracts – beyond type conformance**

- **Semantic substitutability = conformant types + … ?**
- **Hoare triplets:** (precondition) operation (postcondition) [Hoare'69]

  - **Preconditions** of an operation:
    - True on invocation
    - Caller's / provider's requirements

  - **Postconditions** of an operation:
    - True on return
    - Caller'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

---

**Contracts – beyond type conformance**

- **Semantic substitutability = conformant types + … ?**
- **Hoare triplets:** (precondition) operation (postcondition) [Hoare'69]

  - **Preconditions** of an operation:
    - True on invocation
    - Caller's / provider's requirements

  - **Postconditions** of an operation:
    - True on return
    - Caller'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

---

**Contracts – beyond type conformance**

- **Semantic substitutability = conformant types + … ?**
- **Hoare triplets:** (precondition) operation (postcondition) [Hoare'69]

  - **Preconditions** of an operation:
    - True on invocation
    - Caller's / provider's requirements

  - **Postconditions** of an operation:
    - True on return
    - Caller'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

```cpp
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, len, txt: array of char -->]
  // pre len > this.length()
  // post this.length() = len + 1
  // and (all i: 0 <= i < pos: this.read(i) = ch)
  // and (all i: pos <= this.length(): this.read(i) = txt[i])
}

class B {
  int char c;
  public:
  virtual void foo() {...}
  virtual int bar() {...}
}

class C : B {
  float w;
  virtual char* too() {...} // override
  virtual int bar() {...} // override
}

Translation of a virtual method call in C++:

```cpp
someMethod(B q) { ... // may pass a B or C,...
  q.m();
}
```

Example with pre- and postconditions (continued)

```cpp
class GreatTextModel implements TextModel {
  ... // as in TextModel on previous slide
  void write(int pos, char ch); // insert ch at pos
  // [ len, len, txt: array of char -->]
  // pre len := this.length()
  // post this.length() = max(len, pos) + 1
  // and (all i: 0 <= i < len: this.read(i) = ch)
  // and (all i: pos <= len: this.read(i) = txt[i])
}
```

Provider may overfulfill the contract:

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

class C : B {
  // subclass of B
  float w;
  virtual char* too() {...} // override
  virtual int bar() {...} // override
}
```

Syntactic FBCP example: C++

```cpp
class B {
  // base class
  int char c;
  public:
  virtual void too() {...}
  virtual void m() {...}
}

class C : B {
  // subclass of B
  float w;
  virtual char* too() {...} // override
  virtual int bar() {...} // override
}
```

Syntactic FBCP example (cont.)

```cpp
someMethod(B q) { ... // may pass a B or C,...
  q.m();
}
```

Fragile Base Class Problem

- Superclasses (e.g., system library classes) may evolve
  - Advantage: buggy classes 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?

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
class B {
  // base class
  int char c;
  public:
  virtual void too() {...}
  virtual void m() {...}
}

class C : B {
  // subclass of B
  float w;
  virtual char* too() {...} // override
  virtual int bar() {...} // override
}
```

Syntactic FBCP example (cont.)

```cpp
someMethod(B q) { ... // may pass a B or C,...
  q.m();
}
```
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
  - CuSy generated access layer to IR
  - EJB and Corba CCM Containers + Interfaces
  - State-Gatt 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:
  - Operations, pre- and postconditions
  - Syntactic substitutability, Covariance and Contravariance
- 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 /...