TDDD05 Component-Based Software
DF14900 Software Engineering CUGS

# OO Technology:
## Properties and Limitations for Component-Based Design

- **Interfaces**
- **Design by Contract**
- **Syntactic Substitutability**
- **Inheritance Considered Harmful**
- **Fragile Base Class Problems**
- **Mixins and View-Based Composition**

Christoph Kessler, IDA,
Linköpings universitet, 2010

## 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 **?**



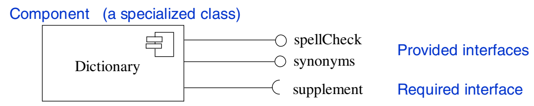
## Interfaces

- **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)

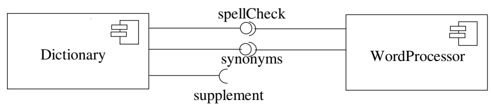


## Interfaces in UML

- Component diagram




- Wiring of components






## 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



## 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?
  (a) Syntactically substitutable:
  The types still match, i.e., the compiler / linker will not complain when recompiling/relinking with client code
  (b) 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?



## 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 = y; or 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 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)

ACM - Turing Award 2009!

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



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

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

- **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):
  **Contravariance and invariance is possible – but not covariance**
  - Provider of *sub*class can substitute a *super*type (e.g., superclass W) for a 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 *sub*class can substitute a *sub*type (e.g., subclass Z) for a result type -> more specific type returned, overfulfills contract
- Workaround may require downcasts with dynamic type checking



## Covariance example

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

- Clients that only care about View will get a generic Model as result type when asking for the view model.
- Clients that know that they are dealing with a TextView object will get a TextModel.
- This is the benign case.



## Contravariance example

```
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:

```
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

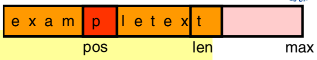
- Semantic substitutability = conformant types + … ?

**Hoare triplets**: {precondition} operation {postcondition} [Hoare'69]

- **Preconditions** of an operation:
  - True on invocation
  - 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

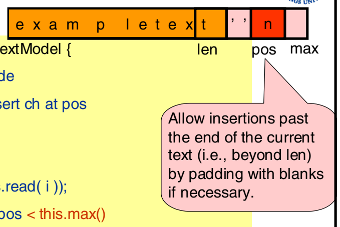
## Example with pre- and postconditions




```
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
   // [ 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
   //          and  (all i: 0<=i<pos:  this.read( i ) = txt[i] )
   //          and  this.read( pos ) = ch
   //          and  (all i:  pos<i<this.length():  this.read( i ) = txt[i-1] ) ]
```



## Provider may overfulfill the contract:




```
class GreatTextModel implements TextModel {
  ... // as in TextModel on previous slide
  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 < this.max()
   //  post  this.length() = max ( len, pos ) + 1
   //         and  (all i: 0<=i< min ( pos, len ):  this.read( i ) = txt[i] )
   //         and  this.read( pos ) = ch
   //         and  (all i:  pos < i <= len:  this.read( i ) = txt[i-1] ) ]
   //         and  (all i:  len < i < pos:  this.read(i) = ' ' ) ]
```



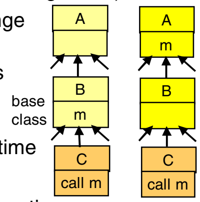
## Fragile Base Class Problem

- 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?



## 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



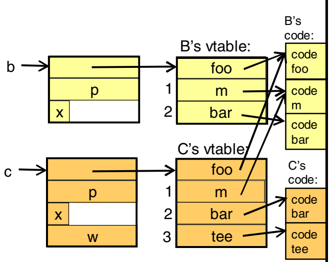

- **Solution 1:**  (IBM SOM)
  Initialize method dispatch tables at load time
- **Solution 2:**  (Java VM)
  Generally look up all virtual methods at run time, even if they could be bound statically e.g. after analysis



## Syntactic FBCP example:  C++

```
class B {    // base class
  int p;   char x;
 public:
  virtual void foo() {...}
  virtual void m ( void ) { ... }
  virtual int bar() { ... }
} b;

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




Translation of a virtual method call in C++:

Same offsets in vtable!

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

```
R1 := q;   // self pointer for q passed in R1
R2 := *q;  // vtable address for q's class
R2 := *(R2 + 1*4);  // index offset 1 by compiler
call R2
```

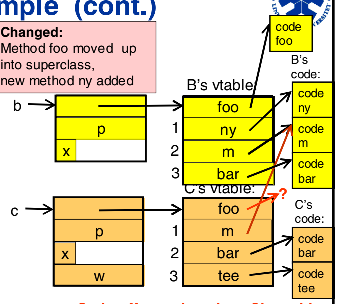


## Syntactic FBCP example  (cont.)

```
class B {    // refact. base class
  int p;   char x;
 public:
  virtual char ny () { ... }
  virtual void m ( void ) { ... }
  virtual int bar() { ... }
} b;

class C : B {  // subclass of B
  float w;      // not recompiled
 public:
  virtual char* tee() { ... }
  virtual int bar() {...} // override
} c;
```

**Changed:**
Method foo moved up into superclass,
new method ny added




Not-recompiled virtual method call:

Stale offset values into C's vtable if C is not recompiled!

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

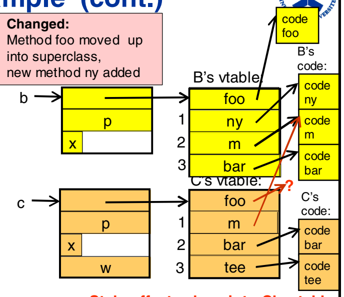
```
R1 := q;   // self pointer for q passed in R1
R2 := *q;  // vtable address for q's class
R2 := *(R2 + 1*4);  // index offset 1 by compiler
call R2
```

## Syntactic FBCP example (cont.)

```
class B : A { // refact. base class
   int p;   char x;
 public:
   virtual char ny () { ... }
   virtual void m ( void ) { ... }
   virtual int bar() { ... }
}  b;
```

**Changed:**
Method foo moved up into superclass,
new method ny added

```
class C : B {  // subclass of B
   float w;      // not recompiled
 public:
   virtual char* tee() { ... }
   virtual int bar() {...} // override
}  c;
```




**Stale offset values into C's vtable if C is not recompiled!**

Recompiled virtual method call:

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

```
R1 := q;   // self pointer for q passed in R1
R2 := *q;  // vtable address for q's class
R2 := *(R2 + 2*4);  // index offset 2 by compiler
call R2
```



## Semantic Fragile Base Class Problem

- Change of inherited functionality in base class may break subclass's correctness (contract)
- L. Mikhajlov, E. Sekerinski: A Study of The Fragile Base Class Problem. *ECOOP'98, Springer LNCS* **1445**: 355-382, 1998.

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

evolves

```
Class Bag' {
   char [] bag;  ...
   void add ( char c ) { ... }
   void addAll ( char[] ac, int n ) {
      /* new implementation
         without calling add() */ ... }
   int cardinality() { ... }
}
```

Correct change within Bag

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

Unchanged (but recompiled)

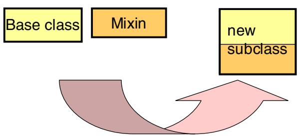
```
Class CountingBag : Bag' {
   int counter;  ...
   void add ( char c ) {
      counter++;  super.add( c ); }
   int cardinality() { return counter;}
}
```

... but breaks the contract of CountingBag



## 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




"extend" operation: this is metaprogramming!

- Variations on this topic:
  - Mixin-Based Inheritance
  - IBM SOM
  - CoSy generated access layer to IR
  - EJB and Corba CCM Containers + Interfaces
  - Stata-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

- Szyperski et al.: *Component Software – Beyond Object-Oriented Programming.* 2nd edition, 2002. Chapters 5, 6, 7.
- Stevens: *Using UML, Software Engineering with Objects and Components.* 2nd edition, Addison-Wesley, 2006.
- U. Assmann: *Invasive Software Composition.* Springer, 2003.
- B. Meyer: Applying Design by Contract. *IEEE Computer*, Oct. 1992, pp. 40-51.
- B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, Nov. 1994
- L. Mikhajlov, E. Sekerinski: A Study of The Fragile Base Class Problem. *ECOOP'98, Springer LNCS* **1445**: 355-382, 1998.
- W. Harrison, H. Ossher: Subject-Oriented Programming (A Critique of Pure Objects). *ACM OOPSLA'93* pp. 411-428.
- 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
- Write a toy example program in C++ that demonstrates the Syntactic Fragile Base Class Problem