Konzepte objektorientierter Programmierung
– Lecture 3 –

Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 03/04

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Core and Basic Language Concepts

- Core Concepts
  - Object Model
  - Interfaces and Encapsulation
  - Classification and Polymorphism
- Basic Language Concepts
  - Description of Objects
  - Inheritance
  - Dynamic Method Binding
  - Contracts
  - Etc.

Subtyping

- **Substitution principle**
  Objects of subtypes can be used wherever objects of supertypes are expected

- **Subtype polymorphism**
  Program parts working with supertype objects work as well with subtype objects

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inheritance versus Subtyping

- **Subtyping** expresses **classification**
- **Inheritance** is a means of **code reuse**
- Inheritance is **usually coupled** with subtyping
  - Terminology: **Subclassing** = Subtyping + Inheritance

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Classification in Software Technology

- Syntactic classification
  - Subtype objects have **wider interfaces** than supertype objects
  - Subtype objects can understand at least the messages that supertype objects can understand

- Semantic classification
  - Subtype objects provide **at least the behavior** of supertype objects

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rules for Subtyping

- Subtype objects must **fulfill contracts** of supertypes, but
  - Subtypes can have **stronger invariants**
  - Overriding methods of subtypes can have
    weaker preconditions
    stronger postconditions
    than corresponding supertype methods

- Concept is called **Behavioral Subtyping**
- Consequence of substitution principle

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inheritance without Subtyping

- Using subclassing without establishing the “is-a” relation is problematic

```
class List {
  …
  void appendFront( Object o ) { … }
  void appendBack( Object o ) { … }
}
```

```
class Stack extends List {
  …
  // appendFront used as push
  void appendBack( Object o ) {
    System.out.println (“Should not
      be used!!”);
  }
}
```

```
void foo ( List l, Object o )
  { l.appendBack( o ); }      //  l could be a Stack object!
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

3. Reusable Components

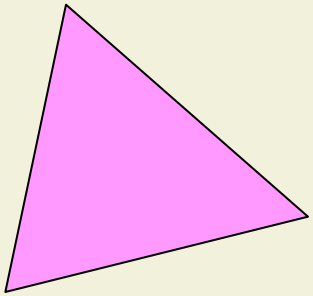
- 3.1 Units of Reuse
- 3.2 Forms of Reuse
- 3.3 Libraries and APIs

Objectives

- Main concepts of reuse
- Deeper understanding of inheritance

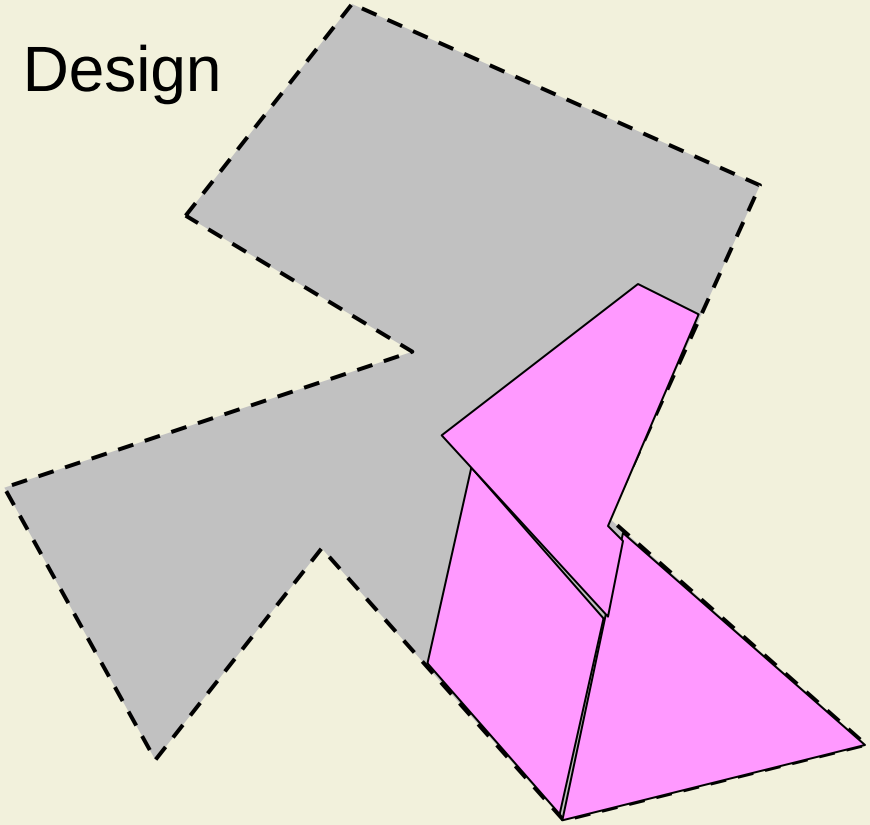
3. Reusable Components

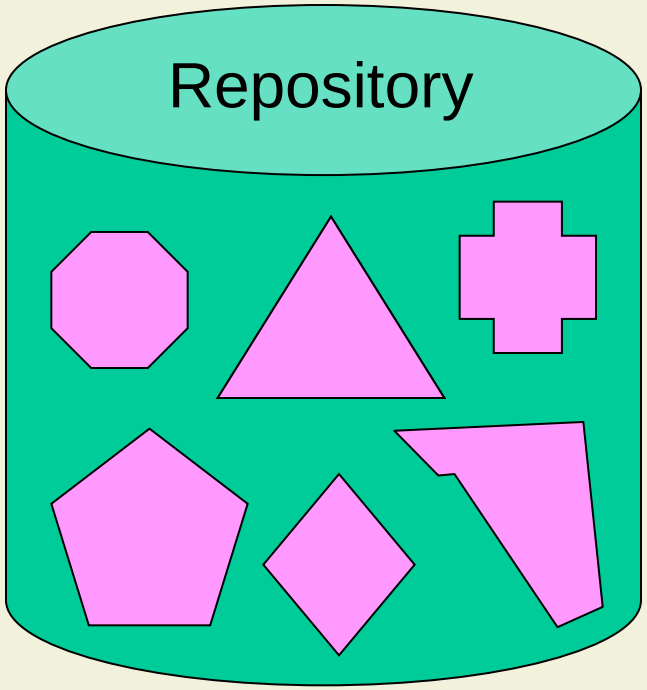
The Vision



Retrieve
Adapt
Reuse

Design

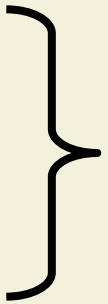




ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Levels of Reuse

- Program parts
  - Code
  - Examples: String, LinkedList
- Designs
  - Design patterns
  - Examples: Observer pattern, factory pattern
- Software architectures
  - Architectural patterns
  - Examples: Client-server, layered architecture



Components (reuse in the small)

Frameworks (reuse in the large)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Component

- Definition:
  An object-oriented component is a group of one or more cooperating classes and interfaces that implements a common abstraction. Components can be reused without further specialization.

- Examples
  - Simple classes such as String, BigInteger, etc.
  - Groups of classes such as DoublyLinkedList – Node – Iterator
  - But not: The Java Abstract Window Toolkit

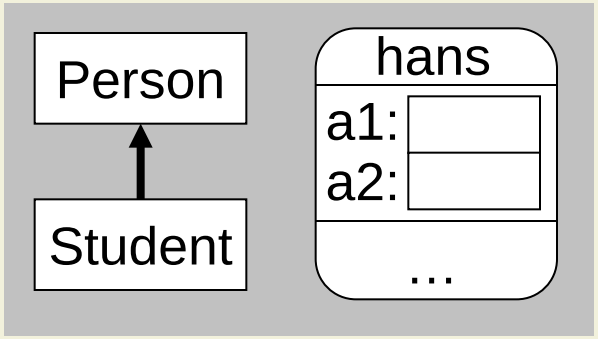
Characteristics of Components

- Components can be in source or binary format
- Components have dependencies
  - Superclasses
  - Types of attributes
  - Return and parameter types of methods
- Many programming languages provide modules to group cooperating classes
  - Modules make dependencies explicit
  - High cohesion within one module (common abstraction)

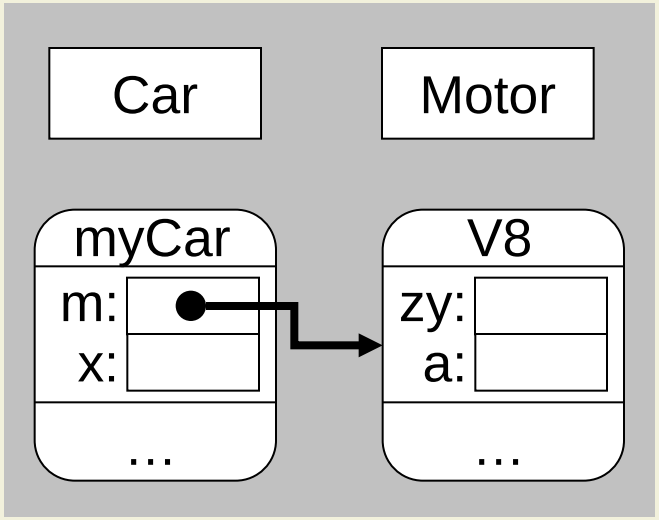
3. Reusable Components

Main Forms of Reuse “in the Small”

- Inheritance
  - Subclassing establishes **“is-a” relation**
  - Enables subtype **polymorphism**
  - Only **one object** at runtime



- Aggregation
  - Establishes **“has-a” relation**
  - **No subtyping** in general
  - **Two objects** at runtime



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inheritance or Aggregation?

- Frequent design issue:
  Reuse a class by inheritance or by aggregation?

- Rule of thumb: Model the real world
  - A student *is a* person – use inheritance
  - A car *has a* motor – use aggregation
  - However, the real world is not always that simple!
- Aggregation is often useful
  - To avoid subtyping
  - To simulate multiple inheritance (e.g., in Java)

Example: Implementation of SymbolTable

- Stores pairs of name / value
- Provides methods add and lookup
- Reuse library class Dictionary

```
class Dictionary {
  Pair[ ] elems;
  int next;

  Dictionary( )                              { … }
  void put( Object key, Object value )       { … }
  Object atKey( Object key )                 { … }
}
```

```
class Pair {
  Object first, second;
  Pair( Object f, Object s )  { … }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Alternative 1: Inheritance

```
class SymbolTable extends Dictionary {

  void add( String key, String value ) {
    put( key, value );
  }

  String lookup( String key ) {
    return ( String ) atKey( key );
  }

}
```

- Attributes and constructor are inherited
- Inherited methods work still
- New methods for naming conventions and type conversion

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Alternative 2: Aggregation

```
class SymbolTable {
  Dictionary rep;

  SymbolTable( ) {
    rep = new Dictionary( );
  }
  void add( String key, String value ) {
    rep.put( key, value );
  }
  String lookup( String key ) {
    return ( String ) rep.atKey( key );
  }
}
```

- Attribute and constructor needed
- Methods implemented by delegation

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Comparison

- Both alternatives are proper OO-implementations
  - SymbolTable is behavioral subtype of Dictionary
- Inheritance leads to shorter implementation
  - Quicker to develop
- Aggregation permits exchange of implementation
  - Not possible with subclassing
- Aggregation approach has smaller interface
  - Dictionary methods can be used in inheritance approach
- Maintainability
  - Aggregation: Longer, but smaller interface

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3. Reusable Components

3.1 Units of Reuse

3.2 Forms of Reuse

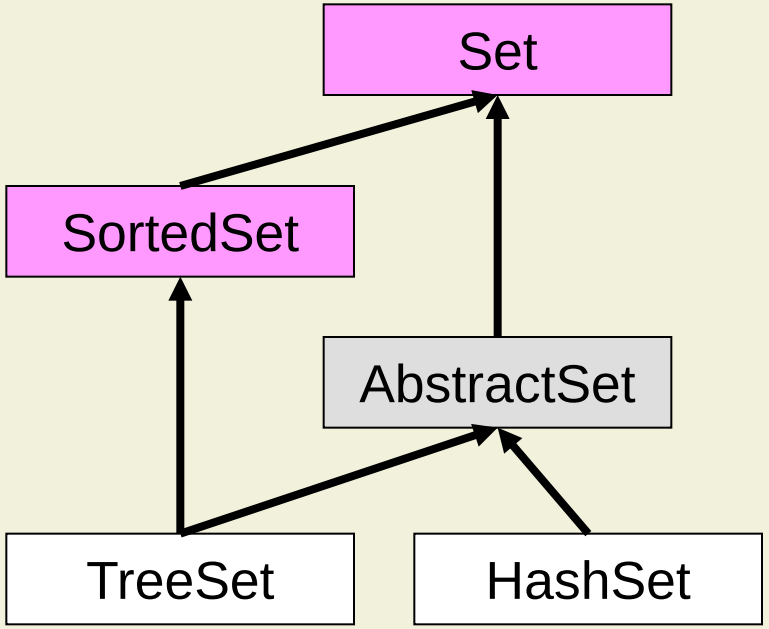
- Aggregation and Inheritance
- **A Deeper Look at Inheritance and Subtyping**
- The Fragile Baseclass Problem

3.3 Libraries and APIs

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Classification of Components

- Reusable components are classified
  - To structure libraries
  - To use inheritance
  - To support polymorphism
- Classification is often not trivial in practice
- Example: Sets and bounded sets



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Attempt 1: BoundedSet extends Set

- BoundedSet refines insert method
- Precondition of insert is strengthened
- Clients using Set might fail when using a BoundedSet

➔ **BoundedSet is no (behavioral) subtype of Set**

```
class Set {
  …
  // requires true
  // ensures isElem( o )
  void insert( Object o ) { … }
}
```

```
class BoundedSet extends Set {
  int size, maxSize;

  // requires size < maxSize
  // ensures isElem( o )
  void insert( Object o ) {
    if (size<maxSize) super.insert(o);
  }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Attempt 2: Set extends BoundedSet

- Set must respect BoundedSet's invariant
- Code duplication is necessary
- Hack: Assign very high number to maxSize
  - Set is more general
  - Might cause other problems (e.g., array implementation)
  - Loss of mathematical properties

➔ **Set is no (behavioral) subtype of BoundedSet**

```
class BoundedSet {
  int size, maxSize;
  // invariant size <= maxSize
  …
  // requires size < maxSize
  // ensures isElem( o )
  void insert( Object o )
  { if (size < maxSize) { code } }
}
```

```
class Set extends BoundedSet {
  // requires size < maxSize
  // ensures isElem( o )
  void insert( Object o ) { code }
}
```


ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Discussion

- The presented classes for Set and BoundedSet are not subtypes
  - Syntactic requirements are met
  - Semantic requirements are not met
  - No effective reuse in attempt 2

- Large parts of the implementation are identical
  - This code should be reused
  - Aggregation is another form of reuse

Solution 1: Aggregation

- BoundedSet **uses** Set
- Method calls are **delegated** to Set
- Code duplication is avoided
  - But code for delegation is needed, even if methods work identically for Set and BoundedSet
- **No subtype relation**
  - No polymorphism

```
class Set {
  …
  void  insert( Object o )    { … }
  int   size( )               { … }
}
```

```
class BoundedSet {
  Set rep;
  int maxSize;

  void insert( Object o ) {
    if (rep.size( ) < maxSize)
      rep.insert( o );
  }
  int size( ) { return rep.size( ); }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Variant of the Problem

- Aggregation seems okay for Set and BoundedSet
- Similar examples require subtyping
- Polygons and Rectanges
  - Polygon: Unbounded set of vertices
  - Rectangle: Bounded set of (exactly four) vertices
  - A rectangle is a polygon!

```
class Polygon {
  Vertex[ ] vertices;
  …
  void addVertex( Vertex v ) { … }
}
```

```
class Rectangle extends Polygon {
  // vertices contains 4 vertices
  …
  void addVertex( Vertex v ) {
    // do nothing
  }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Solution 2: Returning Appropriate Objects

```
class Polygon {
  Vertex[ ] vertices;
  …
  // requires true
  // ensures result.hasVertex( v )
  Polygon addVertex( Vertex v ) {
    … // add v to vertices
    return this;
  }
}
```

```
class Rectangle extends Polygon {
  // vertices contains 4 vertices
  …
  // requires true
  // ensures result.hasVertex( v )
  Polygon addVertex( Vertex v ) {
    return new Pentagon(
      vertices[ 0 ], vertices[ 1 ],
      vertices[ 2 ], vertices[ 3 ], v );
  }
}
```

```
void foo ( Polygon[ ] p, Vertex v ) {
  for( int i=0; i < p.length; i++ ) { p[ i ].addVertex( v ).display( ); }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

BoundedSet Revisited

```
class Set {
  …
  // requires true
  // ensures result.isElem( o )
  Set insert( Object o ) {
    … // insert new element
    return this;
  }
}
```

```
class BoundedSet extends Set {
  int size, maxSize;
  // requires true
  // ensures result.isElem( o )
  Set insert( Object o ) {
    if (size < maxSize)
      return super.insert(o);
    else {
      Set res = new Set( );
      res.insertAll( this );
      res.insert( o );
      return res;
    }
  }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Discussion

- BoundedSet.insert may return Set or BoundedSet object
  - No covariant result types
- No problem for polymorphic client code
- Error-prone for clients of BoundedSet
  - Result types must be invariant in Java

```
Set:
  Set insert( Object o )
```

```
BoundedSet:
  Set insert( Object o )
```

```
Set union( Set from ) {
  Set to = this;
  forall e ∈ from { to = to.insert( e ); }
  return to;
}
```

```
BoundedSet bs = …;
bs = ( BoundedSet )
                    bs.insert( “Risky” );
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Solution 3: Descendant Hiding in Eiffel

- Eiffel supports CAT-calls
  - Changed Availability or Type
- Subclasses can
  - Hide methods
  - Have covariant parameter types
- Possible technical solutions
  - Dynamic checks
  - No polymorphic CAT-calls
- Problems
  - Not compliant with classical subtyping theory
  - Solves Rectangle problem, but not BoundedSet problem

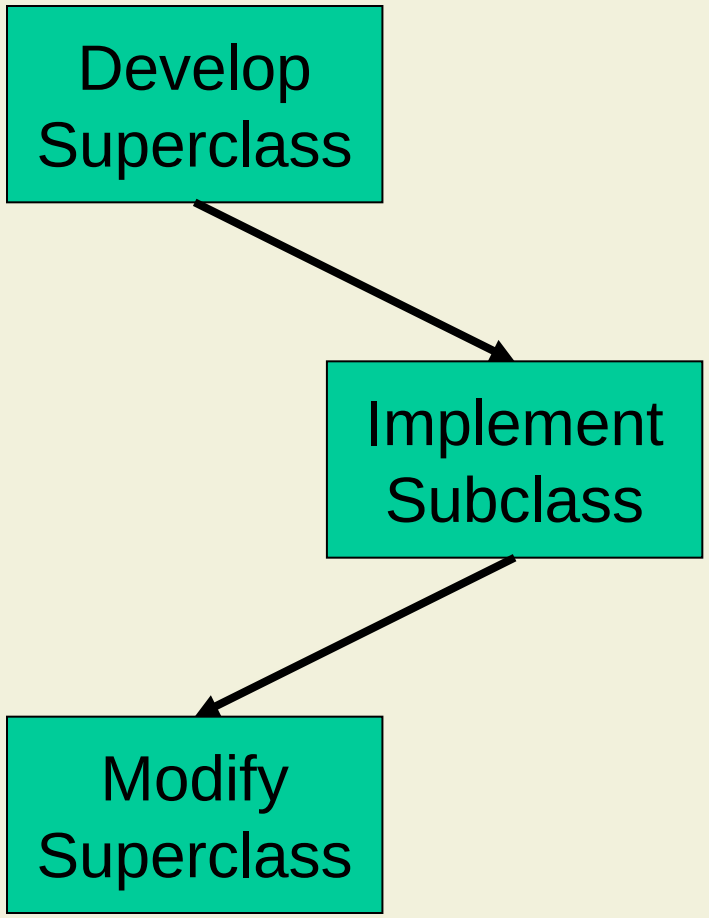
```
class Rectangle
            extends Polygon {
  …
  HIDE void addVertex
                ( Vertex v );
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3. Reusable Components

Fragile Baseclass Scenario

- Software, including reusable components, is not static
  - Maintenance
  - Bugfixing
  - Reengineering
- Subclasses can be affected by changes to superclasses
- How should we apply inheritance to make our code robust against revisions of superclasses?



Example 1: Selective Overriding

```
class Bag {
  …
  int getSize( ) {
    … // count elements
  }

  void insert( Object o )
    { … }

  void insertAll( Object[ ] arr )   {
    … // insert elements of arr
      // directly (not using insert)
  }
}
```

```
class CountingBag extends Bag {
  int size;

  int getSize( )
    { return size; }
  void insert( Object o )
    { super.insert( o ); size++; }
}
```

```
Object[ ]  oa = …  // 5 elements
CountingBag cb =
                    new CountingBag( );
cb.insertAll( oa );
System.out.println( cb.getSize( ) );
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example 1: Discussion

Using inheritance, rely on interface documentation, not on implementation

```
// requires true
// ensures ∀i. 0 <= i < arr.length:
//                  isElem( arr[ i ] )
void insertAll( Object[ ] arr )    {
  for( int i=0; i < arr.length; i++ )
    insert( arr[ i ] );
  }
}
```

```
class CountingBag extends Bag {
  int size;
  // invariant size==super.getSize( )
  …
  void insert( Object o )
    { super.insert( o ); size++; }

  void insertAll( Object[ ] arr ) {
     for( int i=0; i < arr.length; i++ )
       insert( arr[ i ] );
    }
}
```

Override all methods that could break invariants

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Example 2: Unjustified Assumptions

```
class Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return √ f;
  }
  // requires f >= 0
  // ensures result ^ 4 = f
  float fourthRt( float f ) {
   return  squareRt( squareRt( f ) );
  }
}
```

```
class MyMath extends Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return –√f;
  }
}
```

```
MyMath m = new MyMath( );
System.out.println
                ( m.fourthRt( 16 ) );
```

Revising or writing a class, rely on interface documentation, not on implementation

Example 3: Mutual Recursion

```
class C {
  int x;
  // requires true
  // ensures x = old( x ) + 1
  void inc1( ) {
    inc2( );
  }
  // requir
  // ensu
  void in
    x = x
  }
}
```

Be careful when introducing additional invocations on **this.**

```
class CS extends C {
  // requires true
  // ensures x = o    x ) + 1
  void inc2( ) {
    inc1( );
  }
}
```

Avoid inheriting from classes that are expected to be changed (often)

```
CS cs = new CS( );
cs.x = 5;
cs.inc2( );
System.out.println( cs.x );
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example 4: Additional Methods

```
class DiskMgr {

  void delete( ) {
    … // remove temporary files
  }

  void cleanUp( ) {
    delete( );
  }
}
```

```
class MyMgr extends DiskMgr {
  void delete( ) {
    … // erase whole and disk
  }
}
```

Be careful when introducing additional dynamically-bound methods.

Avoid inheriting from classes that are expected to be changed (often)

Rely on common properties of all subclass methods only.

```
MyMgr mm = new MyMgr( );
…
mm.cleanUp( );
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example 4: Solutions

- Java
  - Private, static, and final methods are bound statically
  - **private** and **static** work
  - **final** leads to runtime error
- C#
  - Only **virtual** methods are dynamically bound
  - Only methods tagged **override** override inherited virtual methods
  - Otherwise (tag **new**) the superclass method is hidden
  - Compiler warning for MyMgr

```
class DiskMgr {
  final void delete( ) {
    … // remove temporary files
  }

  void cleanUp( ) {
    delete( );
  }
}
```

```
class MyMgr extends DiskMgr {
  void delete( ) {
    … // erase whole hard disk
  }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Summary: Rules for Proper Subclassing

- Use subclassing only if there is an **"is-a" relation**
  - Syntactic and **behavioral** subtypes
- Do not rely on implementation details
  - Use **precise documentation**, **contracts** where possible
- Make sure that overriding **adapts all methods** that should be specialized
- Rely only on **common behavior of all implementations** of a dynamically-bound method
- Do not use subclassing if superclass implementation is expected to change often
- Apply proper version control

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3. Reusable Components

Libraries and APIs

- Definition of *Library*:
  A collection of components to be used in many different programs
  - Often delivered with software development environments
  - The components in a library may be general purpose or designed for some specific function
- Examples
  - java.util (Set, HashSet, Stack, Iterator, etc.)
  - EiffelBase Support cluster (PRIMES, RANDOM, etc.)
- Special cases of libraries
  - Standard libraries of the programming language
  - Application Program Interfaces (APIs)

Standard Libraries

- Components that are tightly coupled with language
  - Object as root of the subtype hierarchy
  - String as type for string constants
  - Throwable for typing **throws** and **catch** statements
- Standard libraries are part of the language
  - Usually components cannot be implemented in the language (not just a matter of reuse)
  - Syntax and semantics of language cannot be described without standard libraries
- Examples
  - java.lang (Object, Throwable, String, Thread, etc.)
  - EiffelBase Kernel (ARRAY, STRING, REAL, etc.)

Application Program Interfaces (APIs)

- Definition of *API*:
  The interface by which an application program accesses operating system and other services. An API provides a level of abstraction between the application and the kernel (or other privileged utilities).
- Examples
  - java.awt (Component, Window, Event, etc.)
  - .NET Framework Class Library System.IO (File, TextReader, etc.)
  - EiffelBase Dynamic External Shared Call cluster (DLL_32, etc.)