Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

ETH *zürich*

5. Information Hiding and Encapsulation

5.1 Information Hiding

5.2 Encapsulation

Information Hiding

- Definition

  Information hiding is a technique for reducing the dependencies between modules:

  - *The intended client is provided with all the information needed to use the module **correctly**, and **with nothing more***

  - *The client uses only the (publicly) available information*

- Information hiding deals with programs, that is, with static aspects

- Contracts are part of the exported interfaces

ETH *zürich*

Objectives

- Establish strict interfaces

- Hide implementation details

- Reduce dependencies between modules

  - Classes can be studied and understood in isolation

  - Classes interact only in simple, well-defined ways

```
class Set {
  …
  // contract or documentation
  public void insert( Object o )
      { … }
}
```

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

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

ETH *zürich*

The Client Interface of a Class

- Class name
- Type parameters and their bounds
- Super-class
- Super-interfaces
- Signatures of exported methods and fields
- Client interface of direct superclass

```java
class SymbolTable
        extends Dictionary<String,String>
        implements Map<String,String> {
  public int size;

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

  public String lookup( String key )
          throws IllegalArgumentException {
    return atKey( key );
  }
}
```

ETH *zürich*

Other Interfaces

- Subclass interface
  - Efficient access to superclass fields
  - Access to auxiliary superclass methods

- Friend interface
  - Mutual access to implementations of cooperating classes
  - Hiding auxiliary classes

- And others

```java
package coop.util;
public class DList {
  protected Node first, last;
  private int modCount;
  protected void modified( )
    { modCount++; }
  …
}
```

```java
package coop.util;
/* default */ class Node {
  /* default */ Object elem;
  /* default */ Node next, prev;
  … }
```

ETH zürich

Expressing Information Hiding

- ## Java: Access modifiers
    - **public**   client interface
    - **protected**   subclass + friend interface
    - Default access   friend interface
    - **private**   implementation

- ## Eiffel: Clients clause in feature declarations
    - **feature** { ANY }   client interface
    - **feature** { T }   friend interface for class T
    - **feature** { NONE } implementation (only "**this**"-object)
    - All exports include subclasses

Safe Changes

- Consistent renaming of hidden elements
- Modification of hidden implementation as long as exported functionality is preserved
- Access modifiers and clients clauses specify what classes might be affected by a change

```
package coop.util;

public class DList {

  protected Node first, last;

  private int version;
  protected void modified( )
    { version++; }
  …
}
```

Exchanging Implementations

- **Observable behavior must be preserved**

- **Exported fields limit modifications severely**
  - Use getter and setter methods instead

- **Modifications are critical**
  - Fragile baseclass problem
  - Object structures

```
class Coordinate {
  private double x,y;

  …
  public double distOrigin( )
    { return Math.sqrt( x*x + y*y ); }
}
```

```
class Coordinate {
  private double radius, angle;

  …
  public double distOrigin( )
    { return radius; }
}
```

ETH *zürich*

Method Selection in Java (JLS1)

- At compile time:

  1. Determine static declaration

  2. Check accessibility

  3. Determine invocation mode (virtual / nonvirtual)

- At run time:

  4. Compute receiver reference

  5. Locate method to invoke (based on dynamic type of receiver object)

```
class T {
    public void m( ) { ... }
}
```

```
class S extends T {
    public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

Rules for Overriding: Access

- Access Rule:
  The access modifier of an overriding method must provide at least as much access as the overridden method

Default access

protected

public

```
class Super {
  …
  protected void m( ) { … }
}
```

```
class Sub extends Super {
  public      void m( ) { … }
}
```

In class Super or Sub:
public void test(Super v) {
  v.m( );
}

Rules for Overriding: Hiding

- **Override Rule**:
  A method Sub.m overrides the superclass method Super.m only if Super.m is accessible from Sub

- If Super.m is not accessible from Sub, it is hidden by Sub.m

- Private methods cannot be overridden

```
class Super {
  …
  private void m( )
    { System.out.println("Super"); }
  public void test( Super v )
    { v.m( ); }
}
```

```
class Sub extends Super {
  public void m( )
    { System.out.println("Sub"); }
}
```

```
Super v = new Sub( );
v.test( v );
```

Problems with Default Access Methods

- S.m does not override T.m (T.m is not accessible in S)

- T.m and S.m are different methods with same signature

- Static declaration for invocation is T.m

- At run time, S.m is selected and invoked

```
package PT;
public class T {
    void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
    public void m( ) { ... }
}
```

```
In package PT:
T v = new PS.S( );
v.m( );
```

Corrected Method Selection (JLS2)

- Dynamically selected method must override statically determined method

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)

- At run time:
  4. Compute receiver reference
  5. Locate method to invoke that overrides statically determined method

Problems with Protected Methods

- S.m overrides T.m

- Static declaration is T.m, which is accessible for C

- At run time, S.m is selected, which is not accessible for C

- **protected** does not always "provide at least as much access" as **protected**
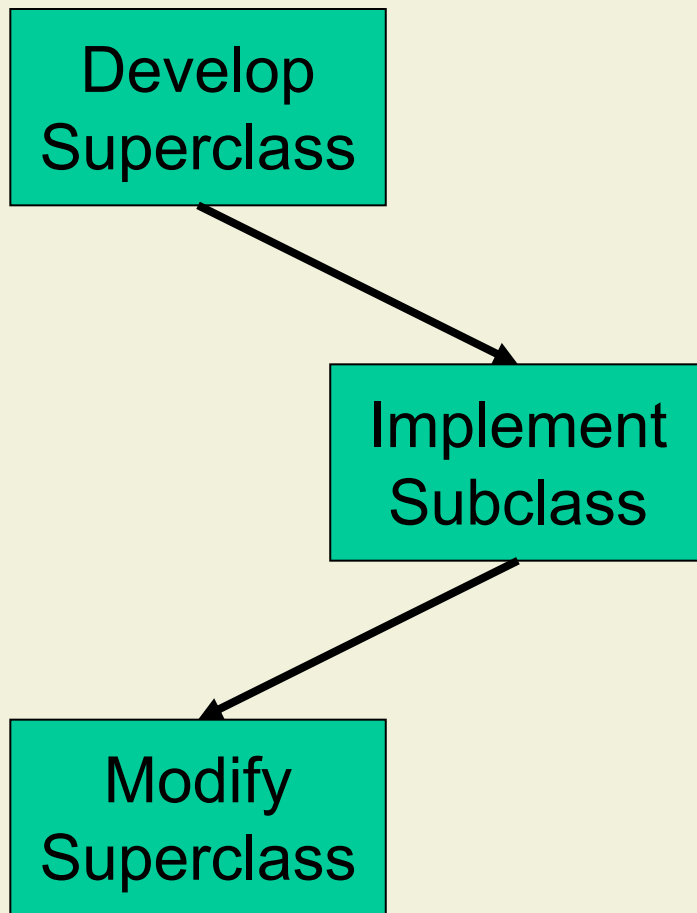
```
package PT;
public class T {
    protected void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
    protected void m( ) { ... }
}
```

```
package PT;
public class C {
  public void foo( ) {
    T v = new PS.S( );
    v.m( );                }
}
```

public would be safe

ETH *zürich*

Another Fragile Baseclass Problem

Develop Superclass

Implement Subclass

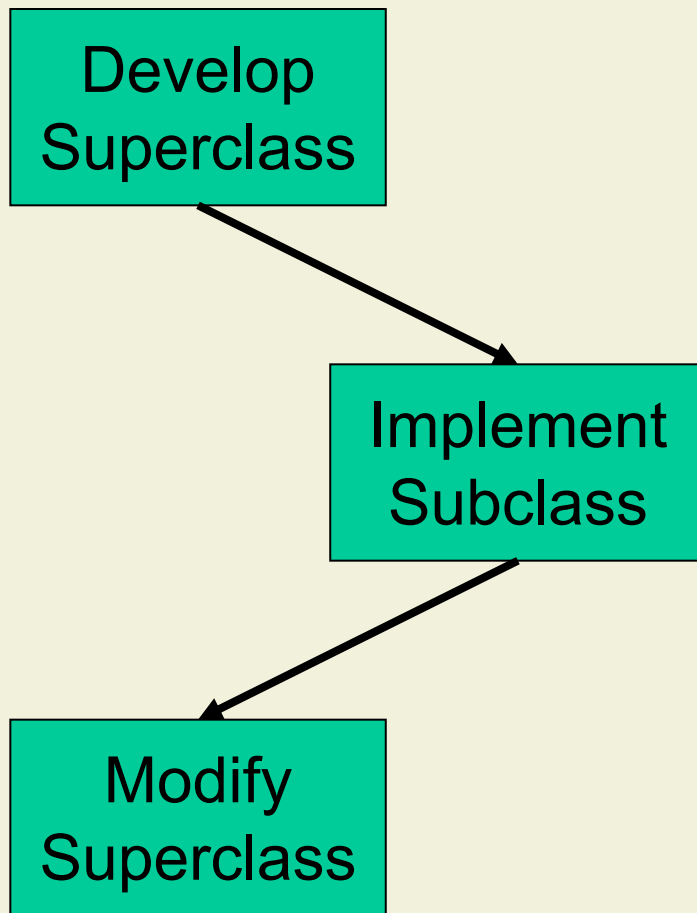Modify Superclass

```
class C {
  int x;
  public     void inc1( )
    { this.inc2( ); }
  private    void inc2( )
    { x++; }
}
```

```
class CS extends C {
  public     void inc2( )    { inc1( ); }
}
```

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

Another Fragile Baseclass Problem

Develop Superclass

Implement Subclass

Modify Superclass

```java
class C {
  int x;
  public      void inc1( )
    { this.inc2( ); }
  protected void inc2( )
    { x++; }
}
```

```java
class CS extends C {
  public      void inc2( )    { inc1( ); }
}
```

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

5. Information Hiding and Encapsulation

5.1 Information Hiding

5.2 Encapsulation

ETH *zürich*

Objective

- A well-behaved module operates according to its specification in any context, in which it can be reused

- Implementations rely on consistency of internal representations

- Reuse contexts should be prevented from violating consistency

```
class Coordinate {
  public double radius, angle;
  // invariant 0 <= radius &&
  //  0 <= angle && angle < 360
  …
  // ensures  0 <= result
  public double distOrigin( )
    { return radius; }
}
```

```
Coordinate c = new Coordinate( );
c.radius = -10;
Math.sqrt( c.distOrigin( ) );
```

Encapsulation

- Definition
  Encapsulation is a technique for structuring the state space of executed programs. Its objective is to guarantee data and structural consistency by establishing capsules with clearly defined interfaces.

- Encapsulation deals mainly with dynamic aspects

- Information hiding and encapsulation are often used synonymously in the literature;
 here, encapsulation is a more specific concept

Levels of Encapsulation

- **Capsules can be**
  - Individual objects
  - Object structures
  - A class (with all of its objects)
  - All classes of a subtype hierarchy
  - A package (with all of its classes and their objects)

- **Encapsulation requires a definition of the boundary of a capsule and the interfaces at the boundary**

Consistency of Objects

- Objects have (external) interfaces and an (internal) representation

- Consistency can include
  - Properties of one execution state
  - Relations between execution states

- The internal representation of an object is encapsulated if it can be manipulated only by using the object's interfaces

obj1
a1:
a2:
ha1:
ha2:
ha3:
m(p1,p2) {..}
m1( ) {..}
m2(p) {..}
h1(p,q) {..}
h2(r) {..}
h3( ) {..}
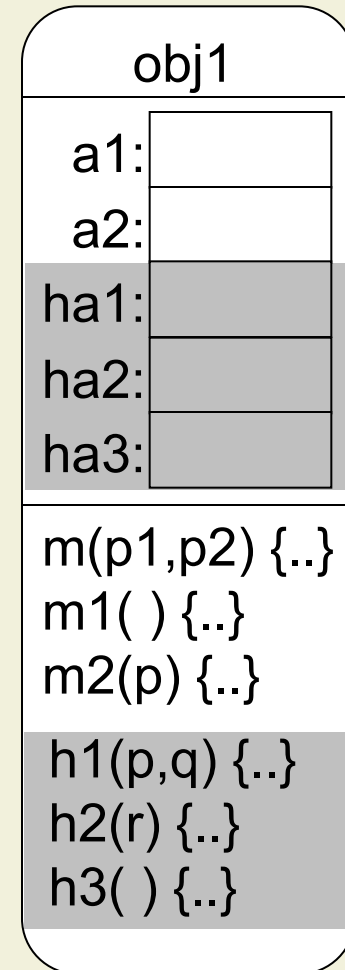
Example: Breaking Consistency

- Problem:
  Exported fields allow objects to manipulate the state of other objects

- Solution:
  Apply proper information hiding

Use **private**

```
class Coordinate {
  public double radius, angle;
  // invariant 0 <= radius &&
  //  0 <= angle && angle < 360
  …
  // ensures  0 <= result
  public double distOrigin( )
    { return radius; }
}
```

```
Coordinate c = new Coordinate( );
c.radius = -10;
Math.sqrt( c.distOrigin( ) );
```

Achieving Consistency of Objects

1. Apply information hiding:
    Hide internal representation wherever possible

2. Make consistency criteria explicit:
    Use contracts or informal documentation to
    express consistency criteria (e.g., invariants)

3. Check interfaces:
    Make sure that all exported operations of an
    object – including subclass methods – preserve all
    documented consistency criteria

Invariants

- **Invariants express consistency properties**

- **The invariant of object o has to hold in:**
  - Prestates of o's methods
  - Poststates of o's methods
- **Temporary violations possible**

```
class Redundant {
  private int a, b;
  // invariant a == b

  …
  public void set( int v ) {
    // prestate: invariant holds
    a = v;
    // invariant does not hold
    b = v;
    // poststate: invariant holds
  }
}
```

Checks for Invariants: Textbook Solution

- **Assume that all objects o are capsules**
  - Only methods executed on o can modify o's state
  - The invariant of object o refers only to the encapsulated fields of o

- **For each invariant, we have to show**
  - That all exported methods preserve the invariants of the receiver object
  - That all constructors establish the invariants of the new object

ETH zürich

Object Consistency in Java

- Declaring all fields **private** does not guarantee encapsulation on the level of individual objects

- Objects of same class can break the invariant

- Eiffel supports encapsulation on the object level

  - **feature** { NONE }

```
class Redundant {
  private int a, b;
  private Redundant next;
  // invariant a == b
  …
  public void set( int v ) { … }

  public void violate( ) {
    // all invariants hold
    next.a = next.b + 1;
    // invariant of next does not hold
  }
}
```

ETH *zürich*

Invariants for Java (Simple Solution)

■ Assumption: The invariants of object o may refer only to private fields of o

■ For each invariant, we have to show
- That all exported methods and constructors of class T preserve the invariants of all objects of T
- That all constructors in addition establish the invariants of the new object

References

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley: *The Java Language Specification. 2013* http://docs.oracle.com/javase/specs/

- Peter Müller and Arnd Poetzsch-Heffter: *Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur.* Java-Informations-Tage, 1998 (in German)

ETH *zürich*