Konzepte objektorientierter Programmierung

– Lecture 5 –

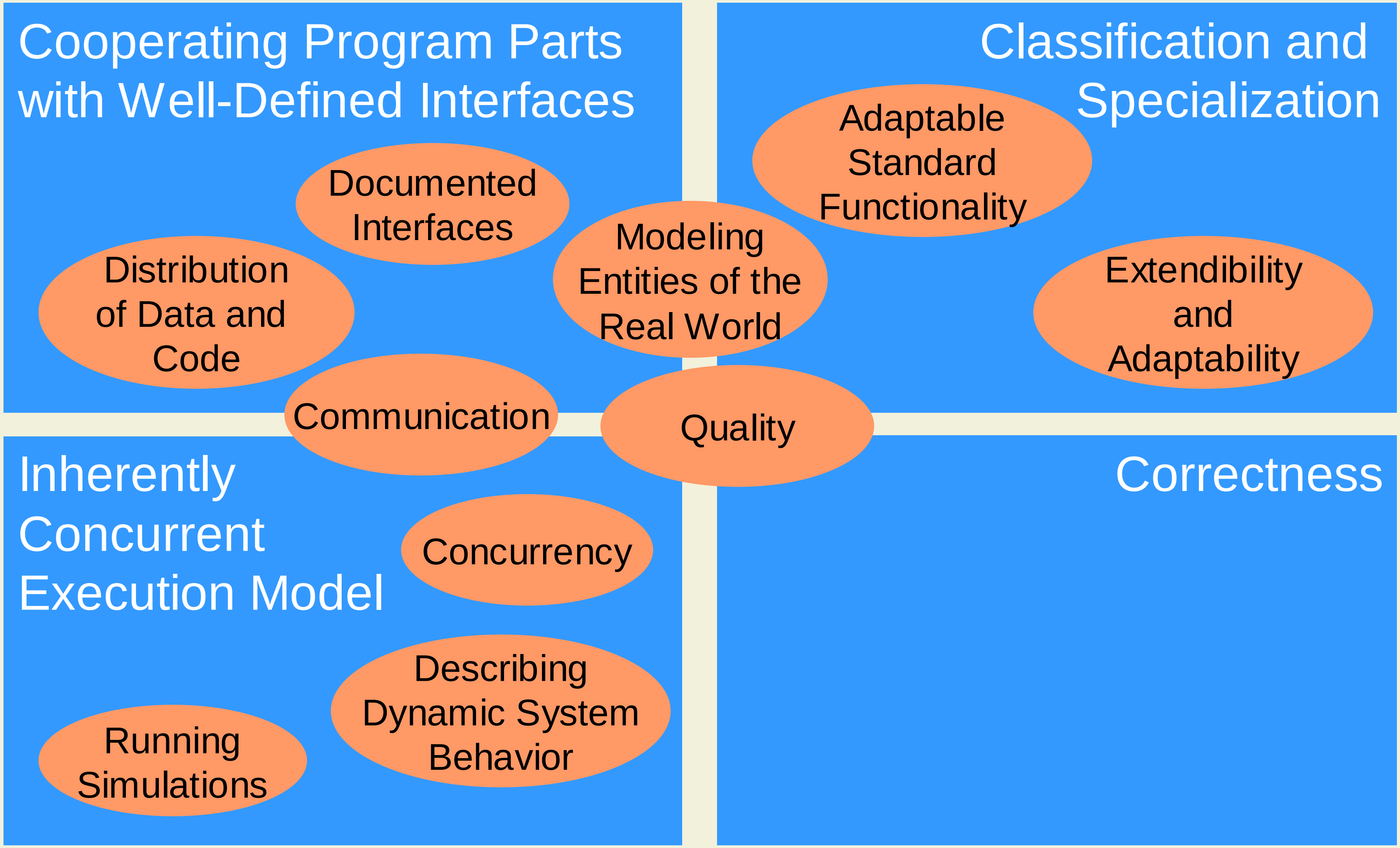
Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 06/07

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements



Cooperating Program Parts with Well-Defined Interfaces

Documented Interfaces

Distribution of Data and Code

Communication

Modeling Entities of the Real World

Classification and Specialization

Adaptable Standard Functionality

Extendibility and Adaptability

Quality

Inherently Concurrent Execution Model

Concurrency

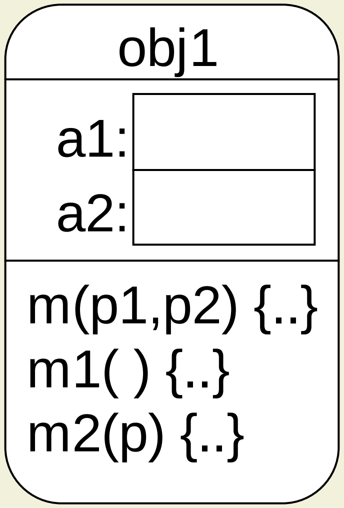
Describing Dynamic System Behavior

Running Simulations

Correctness

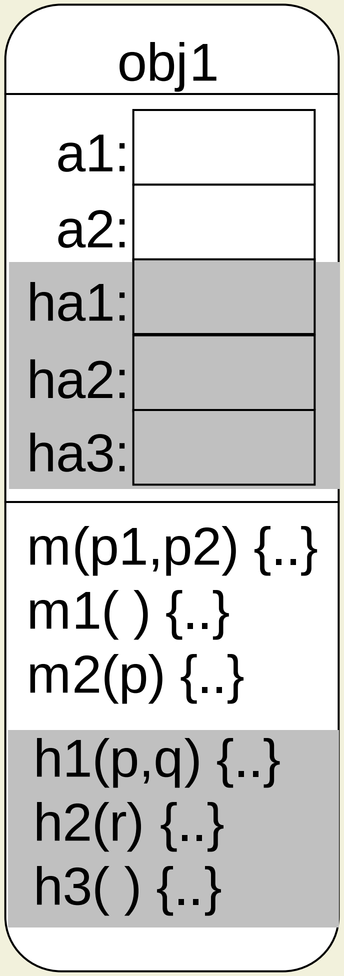
Interfaces and Encapsulation

- Objects have **well-defined interfaces**
  - Publicly accessible attributes
  - Publicly accessible methods
- **Implementation is hidden** behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for **describing behavior**



Interfaces and Encapsulation

- Objects have **well-defined interfaces**
  - Publicly accessible attributes
  - Publicly accessible methods
- **Implementation is hidden** behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for **describing behavior**



Agenda for Today

5. Information Hiding and Encapsulation

- 5.1 Modularization
- 5.2 Information Hiding
- 5.3 Encapsulation
- 5.4 Modularization Revisited

Objectives

- Concepts of modularization
- Consistency of objects

5. Information Hiding and Encapsulation

Objectives of Modularization

- Make **structure** of programs **explicit**
- Reduce dependencies between modules by **strict external interfaces**
- Enable **separate development** of modules
- Enable **reuse** of modules

Explicit Structure

- Closely related program elements are grouped in one module (high cohesion)
- Modules provide one or more interfaces to document how elements can be used by clients
- Dependencies between modules are made explicit

- In OO-programming, **classes** are the main units of modularization
- Some languages provide **additional module systems**

Strict External Interfaces

- Clients can only rely on properties documented in the external interface
- Internal implementation can be modified or exchanged without affecting clients

- **Information hiding** is used to enforce strict external interfaces

Separate Development

- Enables several developers to work in parallel
- Reduces compilation time
- Is a prerequisite for reuse

- OO-modules can be developed, compiled, tested, and deployed separately
  - Languages guarantee syntax and type correctness across module boundaries
  - Modules have to be re-compiled when the interface of an imported module changes

Reuse

- Modules must support adaptation and composition
- Context of reuse is in general not known during module development
- A well-behaved module operates according to its specification in any context, in which it can be reused

- Adaptation and composition are discussed in detail in lectures 3 and 4
- **Encapsulation** is an important technique to ensure well-behavior of modules

5. Information Hiding and 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

Objectives

- Establish strict interfaces
- Hide implementation details
- Reduce dependencies between modules
  - Classes can be studied and understood in isolation
  - Classes only interact 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 );
  }
}
```

The Client Interface of a Class

- Class name
- Names of implemented interfaces
- Signatures of exported methods
- Names and types of exported attributes
- Client interface of the direct superclass

```
class SymbolTable
        extends Dictionary
        implements Map {
  public int size;
  public void
     add( String key, String value )
   { put( key, value ); }
  public String
     lookup( String key )
     throws IllegalArgumentExc {
   return ( String ) atKey( key );
  }
}
```

What about Inheritance?

- Is the name of the superclass part of the client interface or an implementation detail?
- In Java, inheritance is done by subclassing
- Subtype information must be part of the client interface

```
class SymbolTable {
  Dictionary rep;

  public SymbolTable( )
   { … }
  public void
     add( String key, String value )
   { … }
  public String lookup( String key )
   { … }
}
```

```
Dictionary d = new SymbolTable();
d.put( “var”, “5” );
```

The Client Interface of a Class

- Class name
- **Name of direct superclass**
- Names of implemented interfaces
- Signatures of exported methods
- Names and types of exported attributes
- Client interface of the direct superclass

```
class SymbolTable
            extends Dictionary
            implements Map {
  public int size;
  public void
      add(String key, String value)
    { put( key, value ); }
  public String
      lookup( String key )
      throws IllegalArgumentExc {
    return ( String ) atKey( key );
  }
}
```

Other Interfaces

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

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

Other Interfaces

- Subclass interface
    - Efficient access to superclass attributes
    - Access to auxiliary superclass methods
- Friend interface
    - Mutual access to implementations of cooperating classes
    - Hiding auxiliary classes
- And others …

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

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

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

```
package koop.util;

public class DList {

  protected Node first, last;

  private int modCount;
  protected void incrModCount( )
    { modCount++; }
  …
}
```

Safe Changes

- Consistent renaming of hidden elements

```
package koop.util;

public class DList {

  protected Node first, last;

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

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 koop.util;

public class DList {

  protected Node first, last;

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

Exchanging Implementations

- Observable behavior must be preserved
- Exported attributes limit modifications severely
  - Use getter and setter methods instead
  - Uniform access in Eiffel
- Modifications are critical
  - Fragile baseclass problem
  - Object structures

```
class Coordinate {
  private float x,y;
  …
  public float distOrigin( )
    { return Math.sqrt( x*x + y*y ); }
}
```

```
class Coordinate {
  private float radius, angle;
  …
  public float distOrigin( )
    { return radius; }
}
```

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 target reference
  5. Locate method to invoke (based on dynamic type of target 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

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

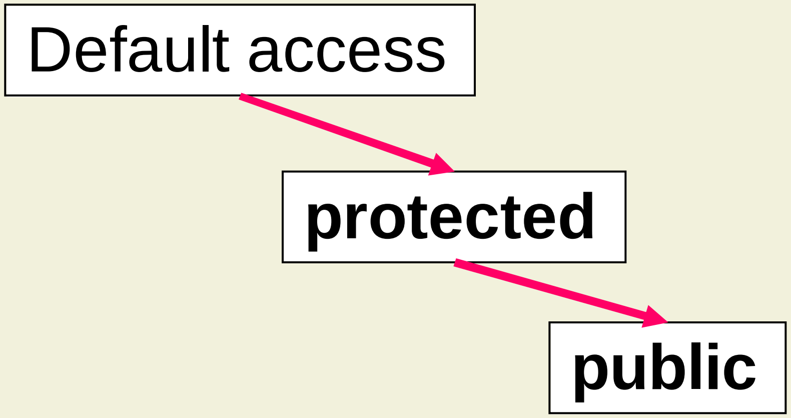
```
class Sub extends Super {
                   void m( ) { … }
}
```

In class Super:

```
public void test( Super v ) {
  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:

```
public void test( Super v ) {
  v.m( );
}
```

Rules for Overriding: Hiding

- **Override Rule**:
  A method Sub.m **overrides** the corresponding 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 runtime, **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
- Theoretically, **uniform treatment** of private and non-private methods possible

- At compile-time:
  1. Determine static declaration
  2. Check accessibility
  3. dropped
- At run-time:
  4. Compute target reference
  5. Locate method to invoke, which overrides statically determined method

Problems with Protected Methods

```
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( );                    }
}
```

Problems with Protected Methods

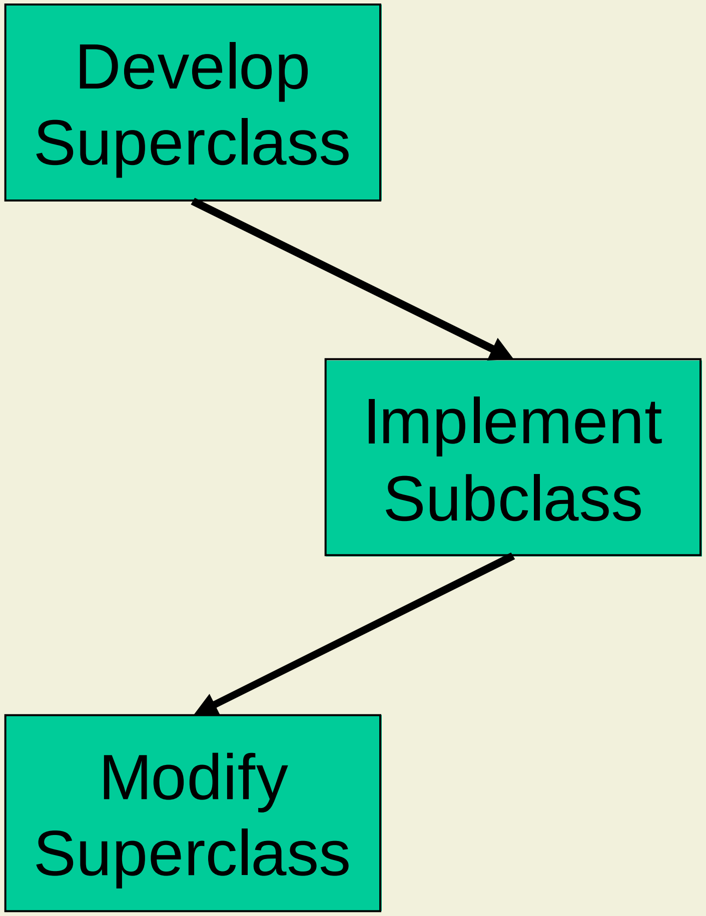
- S.m overrides T.m
- **Static declaration** is T.m, which is **accessible for C**
- **At runtime**, S.m is selected, which is **not accessible for C**
- **protected** does not always “**provide at least as much access**” as **protected**

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

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

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

Another Fragile Baseclass Problem



Develop Superclass

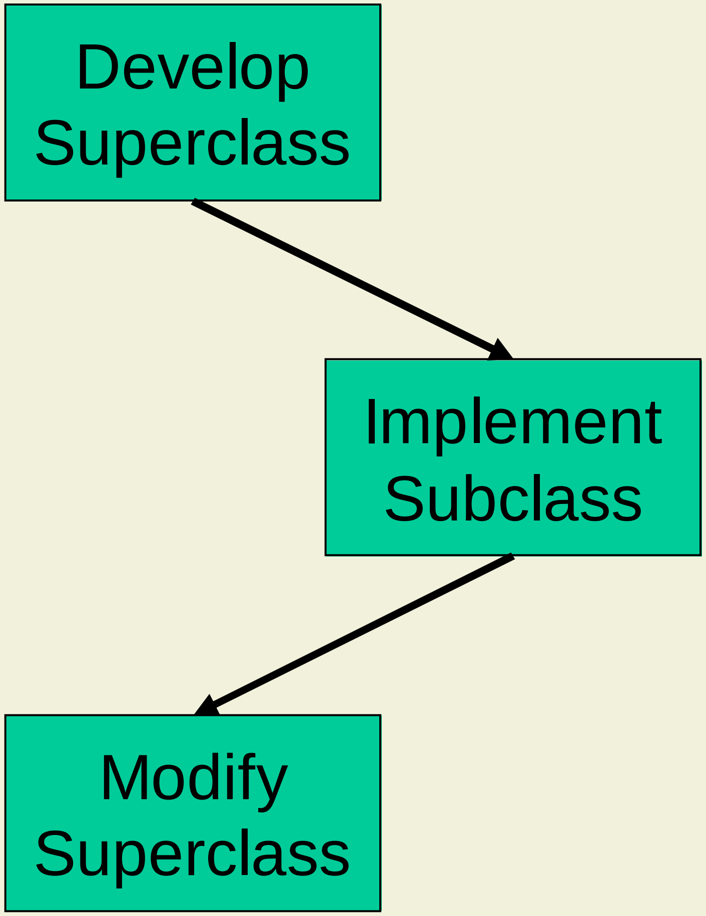
Implement Subclass

Modify Superclass

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

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

Another Fragile Baseclass Problem



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

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

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

5. Information Hiding and Encapsulation

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 float radius, angle;
  // invariant 0 <= radius &&
  //  0 <= angle && angle < 360
  …
  // ensures  0 <= result
  public float 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.

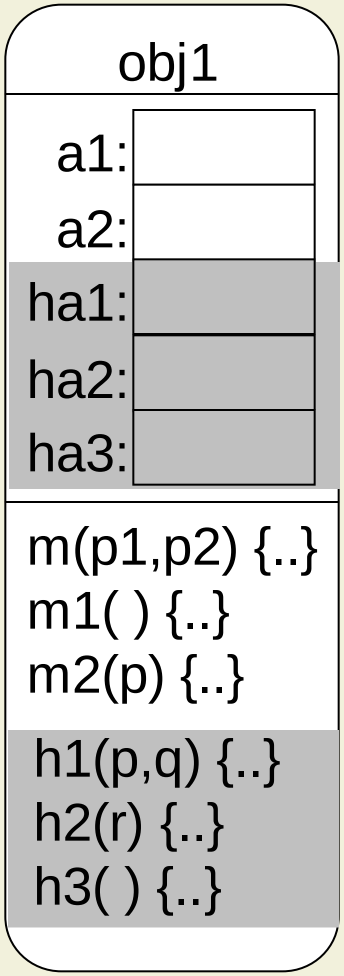
- Encapsulation deals mainly with dynamic aspects
- Information hiding and encapsulation are often used synonymously in the literature;
  here, encapsulation is a more specific concept

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)
  - Several packages
- 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 only be manipulated by using the object's interfaces



Example: Breaking Consistency (1)

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

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

Example: Breaking Consistency (1)

- Problem:
  Exported attributes allow objects to manipulate the state of other objects
- Solution:
  Apply proper information hiding

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

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

Example: Breaking Consistency (2)

- Problem:
  Subclasses can introduce (new or overriding) methods that break consistency
- Solution:
  Apply behavioral subtyping

```
class Coordinate {
  protected float radius, angle;
  // invariant 0 <= radius &&
  //         0 <= angle && angle < 360
  …
  public float getAngle( )
   { return angle; }
}
```

```
class BadCoordinate
                   extends Coordinate {
  public void violate( )
   { angle = -1; }
}
```

```
BadCoordinate c =
      new BadCoordinate( );
c.violate( );
Math.sqrt( c.getAngle( ) );
```

Achieving Consistency of Objects

1. Apply encapsulation:
   Hide internal representation wherever possible
2. Make consistency criteria explicit:
   Use contracts and 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 X has to hold whenever X can be accessed by other objects
  - Prestates of X's exported methods
  - Poststates of X's exported 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
  }
}
```

Proof Obligations for Invariants

- Assume that all objects X are capsules
  - Only methods executed on X can modify X’s state
  - The invariant of object X only refers to the encapsulated attributes of X

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

Object Consistency in Java

- Declaring all attributes **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( ) {
    // invariants hold
    next.a = next.b + 1;
    // invariant of next does not hold
  }
}
```

Invariants for Java (Simple Solution)

- Assumption: The invariants of object X may only refer to **private attributes** of X

- For each invariant, we have to show
  - That all exported methods **and constructors of class T** preserve the invariants **of all objects of T and T’s subclasses**
  - That all constructors **in addition** establish the invariants of the new object

Soundness

- Sound reasoning: Free from logical flaws
- Here, soundness means that the proof obligations suffice to show that all methods actually preserve the invariants of all objects
- Soundness Lemma: *If all methods and constructors fulfill the above proof obligations and the invariants of all objects hold in the prestates of all methods, then the invariants of all objects hold as well in the poststates of all methods*

Simplified Proof Sketch

- An object’s state can be modified directly by attribute updates or indirectly by method calls
- Case 1: Attribute update
  - Let m be a method of class T, and X any object of type S
  - If an invariant of X refers to X.a, a must be private (assumption) and declared in S or a superclass of S (since X has attribute a)
  - If m updates X.a, a must be declared in class T (since it is private)
  - Therefore, T must be S or a superclass of S, and we can apply the proof obligation for m
- Case 2: Method call
  - We can apply the induction hypothesis

5. Information Hiding and Encapsulation

5.1 Modularization

5.2 Information Hiding

5.3 Encapsulation

5.4 Modularization Revisited

Summary: Modularization in Java

Classes	
Explicit structure	–
Information hiding	✓
Separate development	✓
Reuse	✓

- Classes do not make dependencies explicit

Packages	
Explicit structure	–
Information hiding	✓
Separate development	✓
Reuse	–

- Packages do not make dependencies explicit
- Packages do not support encapsulation

- The issue *classes versus modules* is still a research topic