OO concepts

UML representation

- Objects, Classes, Messages/Methods
- Inheritance, Polymorphism, Dynamic Binding
- Abstract Classes, Abstract Coupling
Lecture notes at:

http://www.softwareresearch.net/teaching/ws-201516/vo-software-engineering/
Objects in UML

- **Object notation**

An object diagram provides a run time snapshot of the system, representing objects and the connections between them.
Object diagram
Class relationships (I)

An association can be refined by other relations

Often one models first only the fact that two classes are related and refines later this general notation element
Class relationships (II)

- Each association can be named with a text label (like in the ER-model)
- Role names can be specified at association ends
- Multiplicity can be marked at association ends
- A class can have an association with itself, expressing a relationship between objects of the same class
Class relationships (III)

Multiplicity specification:

1  exactly one
*  any (0 or more)
0..*  any (0 or more)
1..*  1 or more
0..1  0 or 1
2..5  range of values
1..5, 9  range of values or nine
Class relationships (IV)

Example:
Inheritance
Polymorphism
Dynamic Binding
Inheritance (I)

- A class defines the type of an object

- If one models for example a class `Customer` and a class `CorporateCustomer`, one expects that each object of type `CorporateCustomer` to be also of type `Customer`. The type `CorporateCustomer` is a *subtype* of `Customer`.
Inheritance (II)

- A superclass generalizes a subclass
- A subclass specializes a superclass
- A subclass inherits methods and attributes of its superclass
Inheritance(III)

- A subclass has the following possibilities to specialize its behavior:
  - Defining new operations and attributes
  - Modifying existing operations (overwriting methods of the superclass)

**Flatten view:**

![Flatten view diagram]
Inheritance (IV)

- UML Notation

- Customer
  - +checkRegularCustomer():boolean

- PrivateCustomer
  - +checkRegularCustomer():boolean

- CorporateCustomer
  - +checkRegularCustomer():boolean
  - +report(doc:ActivityReport):void
Inheritance (V)

„delta“ view

Flat view
(not in standard UML!)
Inheritance and access rights

- Private members of a superclass are **not accessible** in subclasses
- Protected members of a superclass are accessible **only** in subclasses
- Public members are accessible **everywhere**
- Access rights can be specified globally for a superclass (C++):
  
  ```
  class R : private A{ /* ... */ }
  class S : protected A{ /* ... */ }
  class T : public A{ /* ... */ }
  ```
Inheritance in Java

- Java supports single inheritance, where each class has at most one superclass

- The keyword is `extends`

Example:

```java
public class CorporateCustomer extends Customer{
    ...
}
```
Inheritance in C++

class Base {
    protected: int i;
};

class Derived_1 : private Base {
    int f(Base* b) { return b->i; }
    int g(Derived_1* d) { return d->i; }
};

class Derived_2 : public Base {
    int f(Base* b) { return b->i; }
    int g(Derived_1* d) { return d->i; }
    int f1() { return i; }
};
Inheritance
Polymorphism
Dynamic Binding
Polymorphism (I)

- An object type can be poly (=multiple) morph (=form). This can be depicted in the same way as plug-compatibility:

Objects compatible with the plug  "Plug"-Standard
Inheritance example revisited

Customer

+checkRegularCustomer():boolean

PrivateCustomer

+checkRegularCustomer():boolean

CorporateCustomer

+checkRegularCustomer():boolean
+report(doc:ActivityReport):void
Polymorphism (II)

- Objects of type `CorporateCustomer` (subclass) keep at least the same contract as objects of type `Customer` (superclass).
- Therefore it is meaningful to consider that an object of class $A_i$, which is a subclass of class $A$, **is not only of type** $A_i$ but also of the types given by all $A_i$‘s superclasses (starting with $A$).
- **An object has not only one type. It has multiple types**, and the number of types is given by the position of the class from which the object is generated in the class hierarchy.
Polymorphism – Example (I)

Customer customer = new Customer();
PrivateCustomer privateCustomer = new PrivateCustomer();
CorporateCustomer corporateCustomer = new CorporateCustomer();
Polymorphism – Example (II)

customer = privateCustomer;  // OK

customer
  checkRegularCustomer()

privateCustomer
  checkRegularCustomer()

CorporateCustomer
  checkRegularCustomer()
  report(ActivityReport)
Polymorphism – Example (III)

customer = corporateCustomer;  // OK

customer
    checkRegularCustomer()

PrivateCustomer
    checkRegularCustomer()

CorporateCustomer
    checkRegularCustomer()
    report(ActivityReport)
Polymorphism – Example (IV)

privateCustomer = customer; // wrong
Polymorphism – Example (V)

```java
// wrong

corporateCustomer = customer;
checkRegularCustomer();
report(ActivityReport);
```

```
Customer

PrivateCustomer

checkRegularCustomer()

checkRegularCustomer()

checkRegularCustomer()

corporateCustomer

customer

privateCustomer
```
Polymorphism – Example (VI)

- The reason for failure is that an object which is an instance of class `Customer` does not understand all method calls that an object which is an instance of class `CorporateCustomer` understands.

(1) `corporateCustomer = customer;`
(2) `corporateCustomer.report(monthlyReport);`

(1) Type mismatch: cannot convert from `CorporateCustomer` to `Customer`
(2) The method `report(activityReport)` is undefined for the type `Customer`. 
Polymorphism – Example (VII)

Person

+getBirthDate():String
+getHealthHistory():String

PrivateCustomer

+checkRegularCustomer():boolean

Hotel

+Customer[]
+addCustomer(Customer):void

+DevelopmentPlan():Plan

Investor

Hotel

+PrivateCustomer[]
+CorporateCustomer[]
+addPrivateCustomer(PrivateCustomer):void
+addCorporateCustomer(CorporateCustomer):void

Customer

+checkRegularCustomer():boolean

CorporateCustomer

+checkRegularCustomer():boolean
+report(doc:ActivityReport):void

Company
Static and dynamic type

- **Static type**
  - Accurately given by the declaration in the program text
  - Example: `customer` is of static type `Customer`

- **Dynamic type**
  - The type of the referenced object at runtime
  - Example: after the assignment `customer=corporateCustomer`, the dynamic type of `customer` is `CorporateCustomer`

- A variable with a static type can have several dynamic types during its lifetime, depending of the width and depth of the class hierarchy
Dynamic binding (I)

Dynamic binding: The compiler **does not specify which method is called at runtime**. The method is determined at runtime based on

- The method name
- The variable‘s dynamic type

```java
Customer c;
if (i > 0) then
    c = new CorporateCustomer();
else
    c = new PrivateCustomer();
...
c.checkRegularCustomer();
```
Dynamic binding (II)

When \(i > 0\) is true, the variable \(c\) references an object generated from the class `CorporateCustomer` (and thus has the dynamic type `CorporateCustomer`). Hence, the call to `checkRegularCustomer()` is linked to the method as implemented in `CorporateCustomer`.

- In Java, all methods are dynamically bound, except for the ones explicitly marked by using the keyword `static`.

- In C++, by contrast, methods must be explicitly marked as dynamically bound by using the keyword `virtual`.
Dynamic binding can be used for the plug-in concept

For example, the yellow object may implement \texttt{m1()} differently than the red object
Dynamic binding exercise

```java
public class BaseTest {
    protected int protMember;

    BaseTest(int i) {
        protMember = i;
    }

    public void printM() {
        System.out.println("Value in base class is " + protMember);
    }
}

public class DerivedA extends BaseTest {
    DerivedA(int i) {
        super(i + 1);
    }

    public void printM() {
        System.out.println("Value in derivedA class is " + protMember);
    }
}

public class DerivedB extends BaseTest {
    DerivedB(int i) {
        super(i + 2);
    }
}

public class Worker {
    BaseTest bt;

    public void work() {
        bt = new DerivedB(0);
        bt.printM();
        bt = new DerivedA(0);
        bt.printM();
    }

    public static void main(String[] args) {
        Worker wk = new Worker();
        wk.work();
    }
}
```
The diamond problem

Animal myPet = new BestPet();
myPet.talk();

This problem did not occur in Java prior to version 8
Is-A and Has-A

Typical error: Is-A instead of Has-A
Type test and type guard in Java

- **Type test**: Inquiry of the dynamic type
- **Type guard**: Runtime checking of type casting

Example:

```java
if(customer instanceof CorporateCustomer){  // test
    CorporateCustomer corpCust = (CorporateCustomer)customer;  // guard
    ...
}

if(customer instanceof CorporateCustomer)
    ((CorporateCustomer)customer).report(monthlyReport);
```
Understanding Interactions Between Objects
Object Game

Play a hotel room reservation scenario