Advanced Object-Oriented Design
Lecture 1

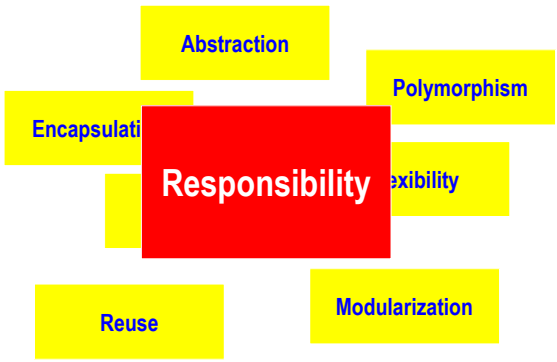
Introduction to objects

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Agenda

1. Introduction
2. Java 2 collections
3. Unit testing
4. Metrics
5. Refactoring I
6. Refactoring II
7. Refactoring III
8. Refactoring IV
9. Design patterns I
10. Design patterns II
11. Design patterns III
12. Aspect-oriented programming
13. Secure code

Different aspects of object-orientation



Object

Definition 1 (traditional)

Object represents a real-world entity; it holds data and defines behavior to operate on it

Definition 2 (object-oriented)

Object is an entity **responsible** for itself.

Example: Conference Lecturer

Problem (Shalloway&Trott 2004)

A lecturer gives a lecture at conference. After it is completed, the attendees are going to listen to some more speeches, but they do not know where they are held. The lecturer is in charge of informing them on the subsequent lectures' locations.

There are different kinds of attendees (students, professionals), who act differently.

Example: Conference Lecturer (cd.)

Solution 1 (functional decomposition)

1. Create a list of attendees for the lecture
2. For each attendee on the list:
   a) Find the next lecture the attendee is going to listen to
   b) Find location for that lecture
   c) Give advice on the way
   d) Pass the information to the attendee

Procedures needed

1. Make a list of attendees
2. Get the plan for each attendee
3. Find a way to the next location
4. Main routine to coordinate remaining procedures

Example: Conference Lecturer (cd.)

Solution 2 (object-oriented)

- Make the plan of a building publicly avaliable
- Allow the attendees to find their way themselves

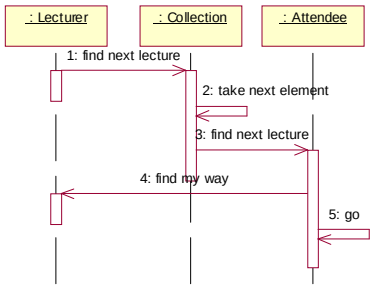
Algorithm for Lecturer

1. Create a collection of attendee instances
2. For every attendee
   a) Instruct the attendee to find the next lecture themselves

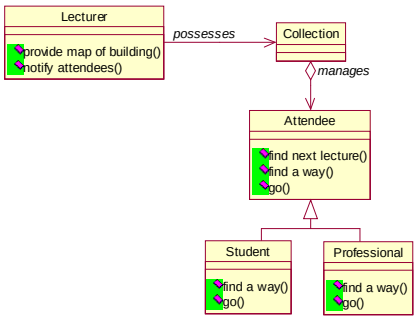
Algorithm for Attendee

1. Find next lecture's localization
2. Find a way there
3. Go

Example: Conference Lecturer (cd.)



Example: Conference Lecturer (cd.)



Abstraction

Abstraction

- The program's ability to ignore some aspects of the information that it is manipulating, and to focus on the essentials.
- Each object in the system serves as a model of an abstract "actor" that can perform work, report on and change its state, and "communicate" with other objects in the system, without revealing how these features are implemented.
- Processes, functions or methods may also be so abstracted, and when they are, a variety of techniques are required to extend an abstraction.

Polymorphism

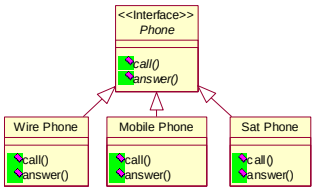
Polymorphism

- poly (gr. *many*) morph (gr. *form*) – coexistence of different behaviors executed in response to a single method call
- the ability to treat an specific object as a general, abstract entity

Polymorphism in Java

- Abstract classess and their derivatives
- Interfaces and their implementations
- Generic types (parametric polymorphism)

Interfaces

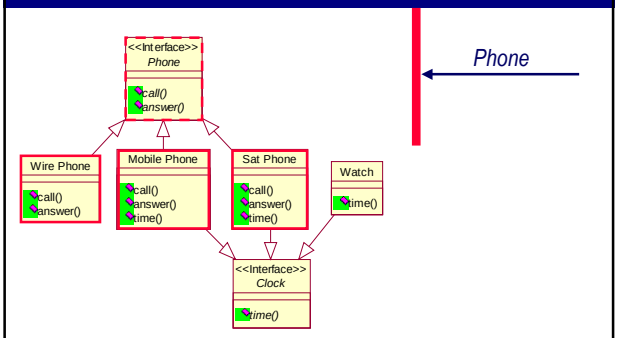


Interface: visible part of the class, defines the responsibility

Principle 1

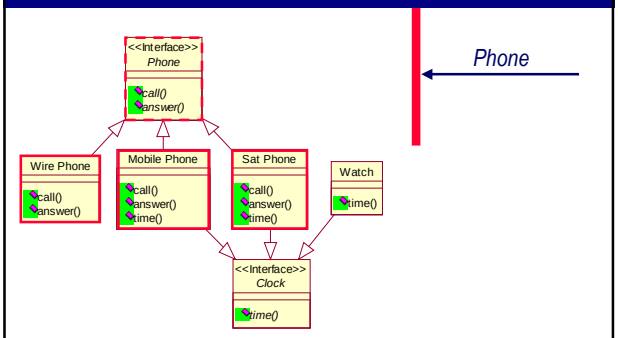
Programming to interfaces (not classes) supports abstraction

Multiple interfaces



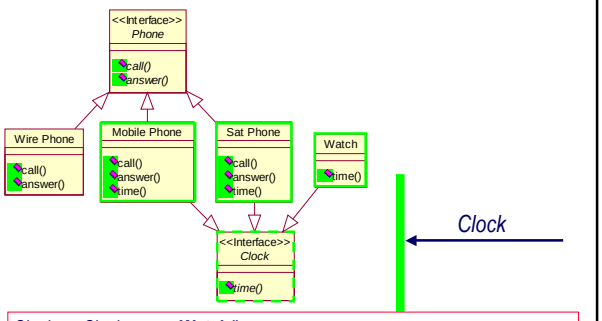

```
Phone myPhone = new WirePhone();
myPhone.call();
```

Multiple interfaces



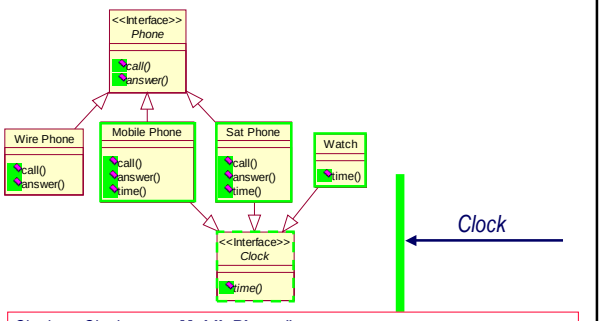

```
Phone myPhone = new MobilePhone();
myPhone.call();
```

Multiple interfaces



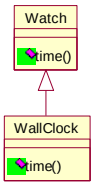

```
Clock myClock = new Watch();
myClock.call();
```

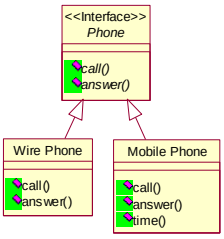
Multiple interfaces




```
Clock myClock = new MobilePhone();
myClock.call();
```

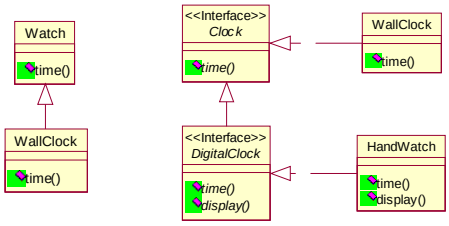
Inheritance vs. interfaces





- **WallClock** is a kind of a **Watch**. It inherits almost all characteristics of a **Watch**.
- **WallClock** inherits both type and implementation.
- **WirePhone** and **MobilePhone** are **Phones**. They both can ***call*** and ***answer.***
- ***WirePhone*** and **MobilePhone** share the type only.

Class inheritance vs. interface inheritance



Class inheritance: inherits type and implementation

Interface inheritance: inherits only type

Encapsulation

Definition

- Ensures that users of an object cannot change the internal state of the object in unexpected ways; only the object's own internal methods are allowed to access its state.
- Each object exposes an interface that specifies how other objects may interact with it.

Comments

Encapsulation is about hiding:

- (popular) Encapsulation is about hiding access to data
- (general) Encapsulation is about hiding design decision that are likely to change: type, implementation, behavior, data.

Principle 2

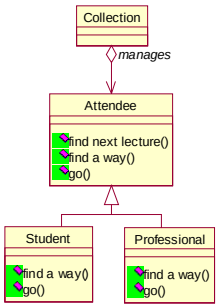
Localize variability within a system and encapsulate it.

Type encapsulation

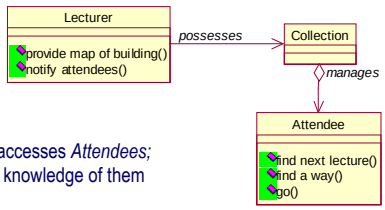
Example

Collection manages *Attendees*, ignoring the existence of a *Student* and a *Professional*

Attendee protects deriving types from its clients



Object encapsulation



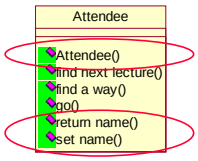
Example

only *Collection* accesses *Attendees;* *Lecturer* has no knowledge of them

Data encapsulation

Example

- *Attendees* cannot change their names
- creation of invalid objects is explicitly prohibited

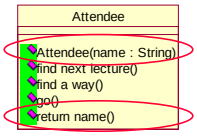



```
Attendee attendee = new Attendee();
Attendee.setName("Smith");
```

Data encapsulation

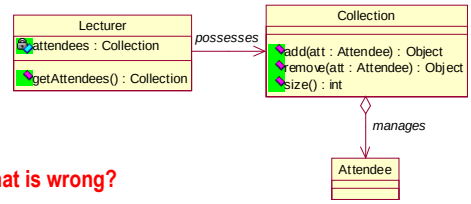
Example

- *Attendees* cannot change their names
- creation of invalid objects is explicitly prohibited




```
Attendee attendee = new Attendee("Smith");
```

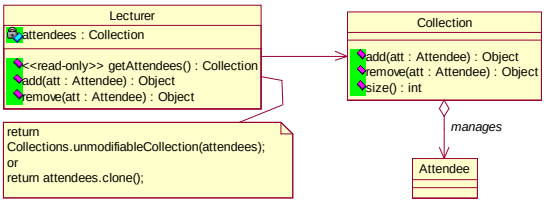
Encapsulation



What is wrong?

```
Collection attendees = lecturer.getAttendees();
attendees.add(new Attendee("George Bush"));    // freely modifiable
attendees.remove(new Attendee("Tony Blair"));  // freely modifiable
```

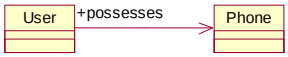
Encapsulation




```
Collection attendees = lecturer.getAttendees();
attendees.add(new Attendee("George Bush"));    // exception
```

```
lecturers.add(new Attendee("George Bush"));
```

Different types of relations: association



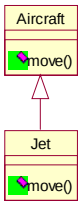
- *Phone* belongs to the *User*
- Both *Phone* and *User* can change their counter-parties
- *User* knows their *Phone,* while *Phone* ignores its *User*
- Phone and User exist independently

Different types of relations: composition

- *Book* is composed of *Pages.*
- *Page* is a part of *Book.*
- *Page* cannot exist independently from a *Book*
- *Book* is responsible for *Pages* (adds, removes etc.)

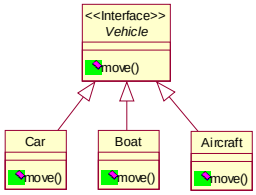
Different types of relations: inheritance

- **Type inheritance**
- **Behavior inheritance**



- *Jet* is a kind of an *Aircraft*
- *Jet* inherits all characteristics of an *Aircraft*, including type
- *Jet* is a substitite for an *Aircraft*
- *Jet* indirectly accesses an *Aircraft*
- the relation between *Jet* and *Aircraft* is indistractable

Different types of relations: realization



- *Vehicle* declares a *move()* operation
- *Car*, *Boat* and *Aircraft* define *move()* operation on their own

Inheritance vs. composition

Inheritance	Composition
Relation is fixed at compile-time and cannot be distracted	Relation is changeable at run-time
It prevents from NPE	It does not assure the related objects to exist
Passes to the descendent both type (interface) and implementation	The object at owner side knows only type (interface) of its party
Exhibits internals to descendent classes	

Principle 4

Prefer composition over inheritance

Cohesion

Cohesion

The cohesion of an object or class is the extent to which the elements (or characteristics) of the object or class are related to one another.

Cohesive means that a certain class performs a set of closely related actions, thus its responsibility is clearly defined. A lack of cohesion, on the other hand, means that a class is performing several unrelated tasks and should possibly be splitted.

Principle 5

A well-designed object is highly cohesive.

Coupling

Coupling

The coupling between two objects/classes is the manner and degree of interdependence between them. Class *A* is coupled to a class *B* if *A* needs to know *B* in some way:

- as its member
- as implementation
- as its subclass
- as a parameter, return type or a declared exception in a method signature

Principle 6

Promote loose coupling. Excessive coupling decreases maintainability and understandability.

Coupling (cont.)

Comment

Classes needs to be related to exchange messages, thus some coupling is unavoidable.

Good design means removing excessive and unnecessary coupling.

What is the acceptable coupling?

Reference objects

Identity: Identified by object's reference (or other explicit identifier)

Cardinality: There exist a single instance (unless it is cached), referenced simultenously by multiple clients

Mutability: Object is mutable, changes are immediately available to clients

Comparing: objects are equal iff their references (identifiers) are equal

Features

- Reference objects usually represent larger, unique entities with many atributtes and methods, difficult to create or synchronize, like people, books, accounts etc.
- They are often created by a dedicated factory, which manages the sole instance.

Value objects

Identity: Identified by an overall value (state) stored within object

Cardinality: There may exist multiple equal (with regard to content) objects

Mutability: Objects are immutable (fixed at creation time and never altered)

Comparing: objects are equal iff they implement same type and they hold same state (their corresponding attributes are equal)

Features

- Reference objects usually represent smaller, exchangeable entities with few (often one) atributte, inexpensive at creation, like timestamps, numbers, money values etc.
- They are usually created by a direct call to a constructor.

CRC card

Class:
Responsibility
Collaboration

CRC card

Class: Lecturer

Responsibility

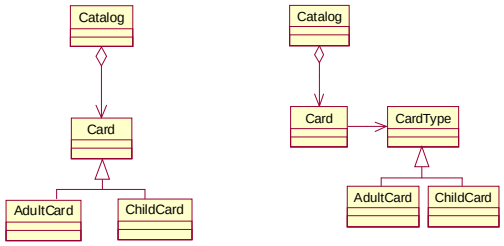
- Providing access to the building's plan
- Allowing *Attendees* to find their way

Collaboration

- Collection: indirectly notifying the *Attendees* to find their next lecture

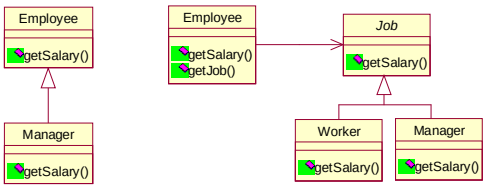
Example 1

Readers' catalog stores adults and children.



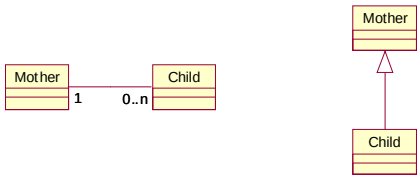
Example 2

Some of company's employees are managers.



Example 3

Every child knows his/her mother. Every mother knows her children



Readings

1. J. W. Cooper: *Java. Wzorce projektowe.* Helion 2001
2. B. Eckel: *Thinking in Java.* Helion 2001
3. J. Shalloway, J. Trott: *Projektowanie zorientowane obiektowo. Wzorce projektowe.* Helion 2001/2005
4. E. Gamma et al.: *Design patterns. Elements of reusable software.* Addison-Wesley 1995
5. M. Fowler: *Refactoring. Improving design of existing software.* Addison-Wesley 1999
6. J. Langer: *Java style. Patterns for implementation.* Prentice Hall 2000

Q&A