Object-Orientation and Aspect-Orientation

Jörg Kienzle & Alfred Strohmeier

Overview

- Object-Orientation
  - Object
    - Identity, State, Behaviour, Operations, Attributes
    - Object Life Cycle
    - Object Interface and Implementation
  - Object Interactions
  - Object System
  - Class
  - Type
  - Generalization / Specialization (subtyping)
  - Polymorphism
- Aspect-Orientation
  - Shortcomings of Object-Orientation
  - Aspect-Oriented Terminology
  - Aspect-Oriented Programming Example: AspectJ

Foundations of Object-Orientation

- Abstraction
  - Extraction of essential properties while omitting inessential details.
- Information hiding (encapsulation)
  - Separation of the external view from the internal details.
  - Aspects that should not affect other parts of the system are made inaccessible.
- Modularity
  - Decomposition into a set of cohesive and loosely coupled units; i.e. purposeful structuring.
- Classification
  - Ability to group objects according to common properties.
  - Ability for an object to belong to more than one classification.

Object

- An object represents an individual, identifiable item, unit, or entity, either real or conceptual, with a well-defined role in the problem domain or in a system.
- In a computer-based system, an object may stand for itself, e.g. a window or a menu item, or it may represent, be a surrogate of, a real-world object, like a person or a car.
  - This distinction is not always clear-cut, see e.g. a bank account.
- When an object models a real-world entity, it is an abstraction of this entity. What is essential and what is accidental will depend on the application and system.
- A property is an inherent or distinctive characteristic, trait, quality, or feature of an object.

Object Examples

- The printer Neo, of type Phaser 4400N, made by Xerox, located in room McConnell 322...
- Mr. Rich, business man, 42 years old, living in Lausanne, Switzerland, married to Mrs. Dufour, ...
- The bank account of Mr. Rich with the Swiss Union Bank...

Graphical Representation in UML

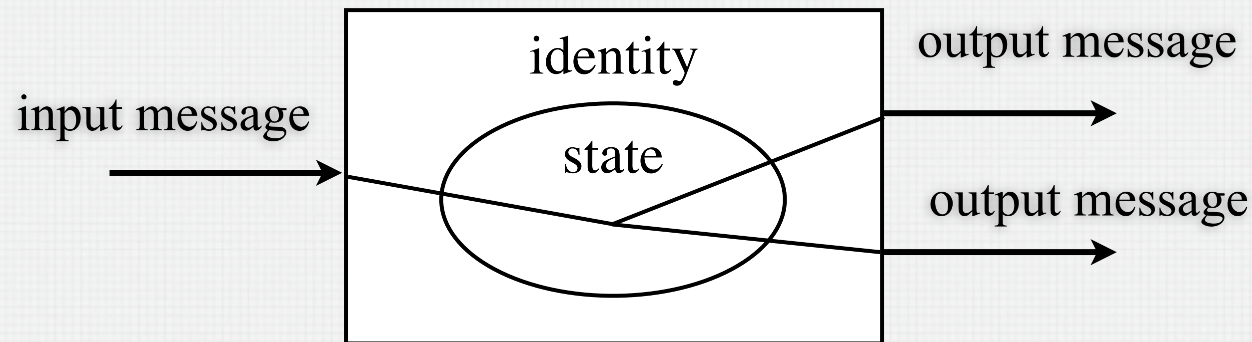
<u>object name [: Class name]</u>

<u>paul : Person</u>

lower case

<u>: Person</u>

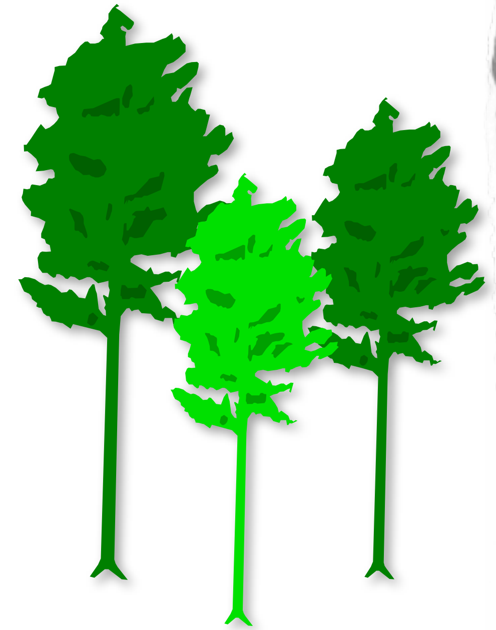
Properties of an Object



Object = Identity + State + Behaviour

Object Identity

- Identity is the property that distinguishes an object from all others
  - It is always possible to distinguish two objects, even if they have the same state
- The identity of an object cannot be changed
  - The name or a reference should not be confused with an object's identity
- In a computer system, the identity of an object may be implemented by its storage address, or a special attribute



State and Behaviour

- The state of an object is its memory. Since an object has a state, it takes up some amount of space, be it in the physical world or in computer memory.
- The behaviour is how an object acts on its own initiative and how it reacts to external stimuli, i.e. events or messages, in terms of its state changes and output messages.
  - The behaviour of an object usually depends on its history; this time-dependent behaviour is due to the existence of state within the object.
- State and behaviour are abstract concepts.

Describing State and Behaviour

- An object
  - is denoted by a name or a reference,
  - has attributes,
  - provides a set of operations.
- Data (state, attributes, structure) and operations (services, functions, subprograms) are gathered together in an object.

Operations on Objects

- An operation is an action that an object performs on its own initiative or upon request.
- The operations describe dynamic properties of the object; they are part of its behaviour.
- Object operations are ultimately responsible for providing the expected behaviour.
- The set of operations an object is able to perform is called its protocol or, in UML, it’s called its interface.

Kind of Operations

- Constructors
  - Create, build, and initialize an object
- Observers
  - Retrieve information about the state of an object
- Modifiers
  - Alter the state of an object
- Destructors
  - Destroy an object
- Iterators (for objects that encapsulate a collection of other objects)
  - Access all parts of a composite object, and apply some action to each of the parts

Object Attributes

- Attributes describe static properties of an object; they retrieve or hold information about the state of the object; the information may be a data value or a link to another object.
- A value is a characteristic that can be measured, or is defined by agreement, and that has no existence by its own, and therefore no identity.
- A value exists only when attached to an object, a property of which it describes.
- The attributes of an object remain the same, but their values may change.

Attribute and Operation Examples

- Neo has already printed 5614 b/w pages. The toner has to be replaced soon.
- Mr. Rich, business man, 42 years old, living in Lausanne, Switzerland, married to Mrs. Dufour, ...
- Mr. Rich has 36,880 CHF in his checking account. It is time to transfer part of it to his savings account.

Computer-based Implementation

- The state of an object is implemented by data fields or a data structure encapsulated in the object.
- The operations are implemented by methods (sometimes called subprograms, operation bodies, etc.).

Object Life Cycle

- An object has a life cycle:
  - It is created,
  - It lives and evolves,
  - It is destroyed.
- The object keeps its identity during its whole life cycle.
- During its life cycle, the state of the object may change, the values of its attributes may change, the effects of its operations may change, but the set of operations it provides remain the same.

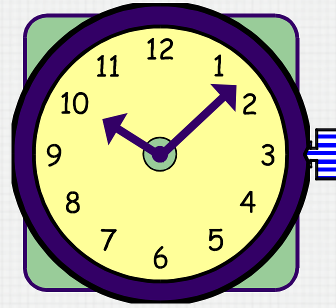
Objects as Machines

- The existence of state within an object means that the order in which operations are invoked is important.
- Each object is like a tiny, independent machine.
- The behaviour of an object can be modelled in terms of an equivalent finite state machine.

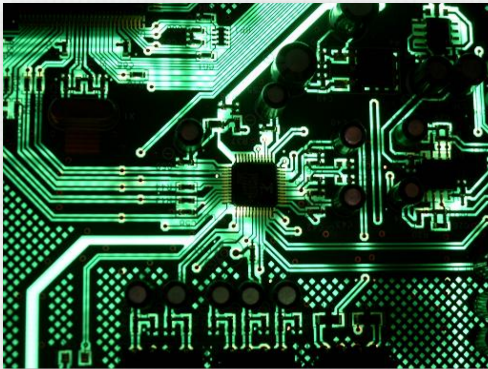
Interface and Implementation (1)

- The interface of an object provides its outside view. It comprises all methods applicable to the object and may include fields as well. It emphasizes the abstraction while hiding its internal structure and the secrets of its internal working.
- Abstraction allows us to write complex software without having to know how parts of it actually work.

Interface and Implementation (2)



Interface



Possible Implementations

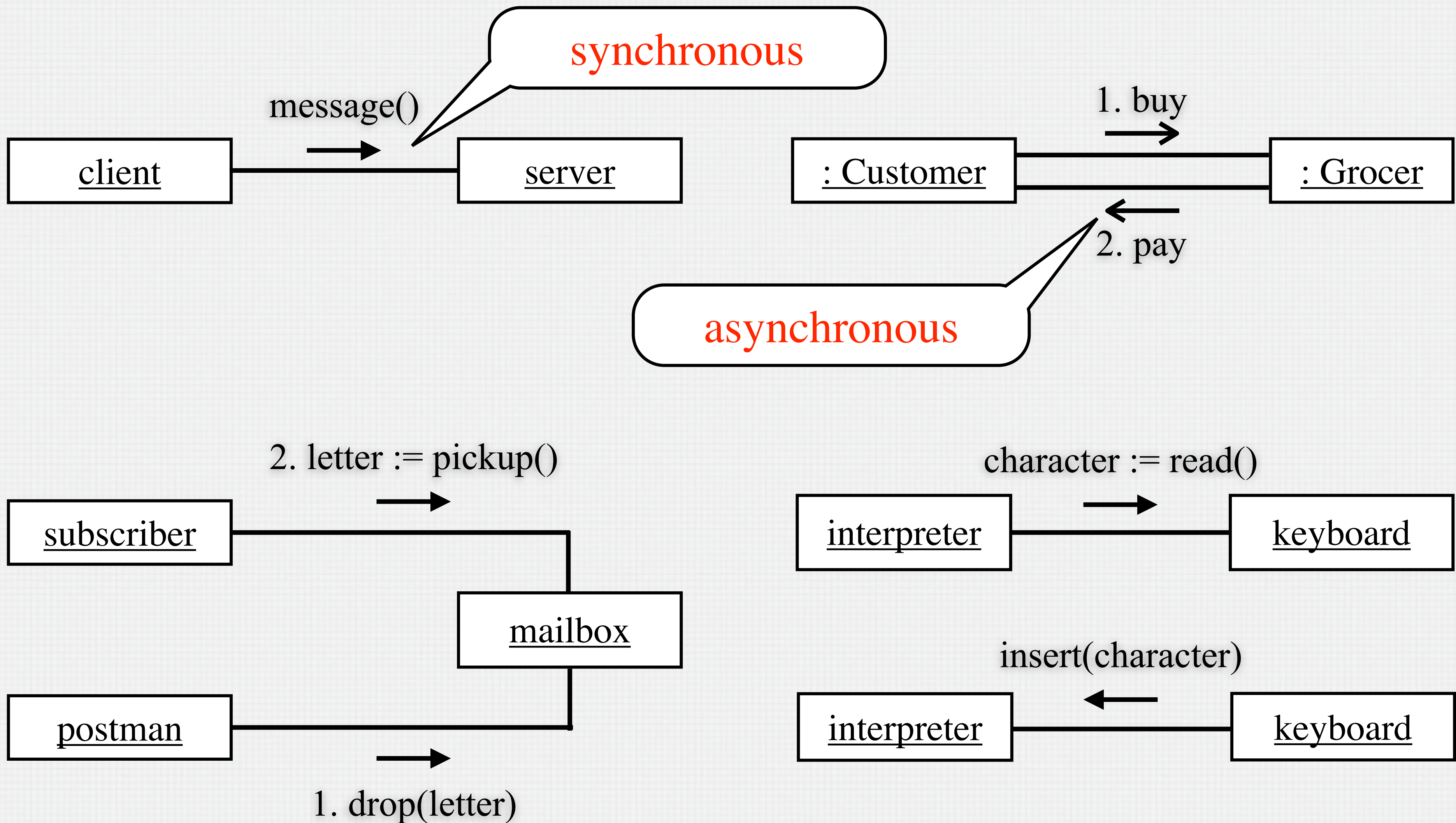
Object Interactions

- An object (client) may ask another object (server) to provide a service by sending a message to it. The message specifies:
  - A destination: a reference to the server object
  - A selector: the name of the service, operation or method to be performed
  - Parameters: additional information needed for specifying the request or for performing the service, including returning results.

Object Interaction Examples

- Mr. Rich withdraws one million dollars from his account with the Swiss Union Bank
- A car driver may:
  - Speed up,
  - Consult the speedometer,
  - Turn right by 30 degrees, etc.

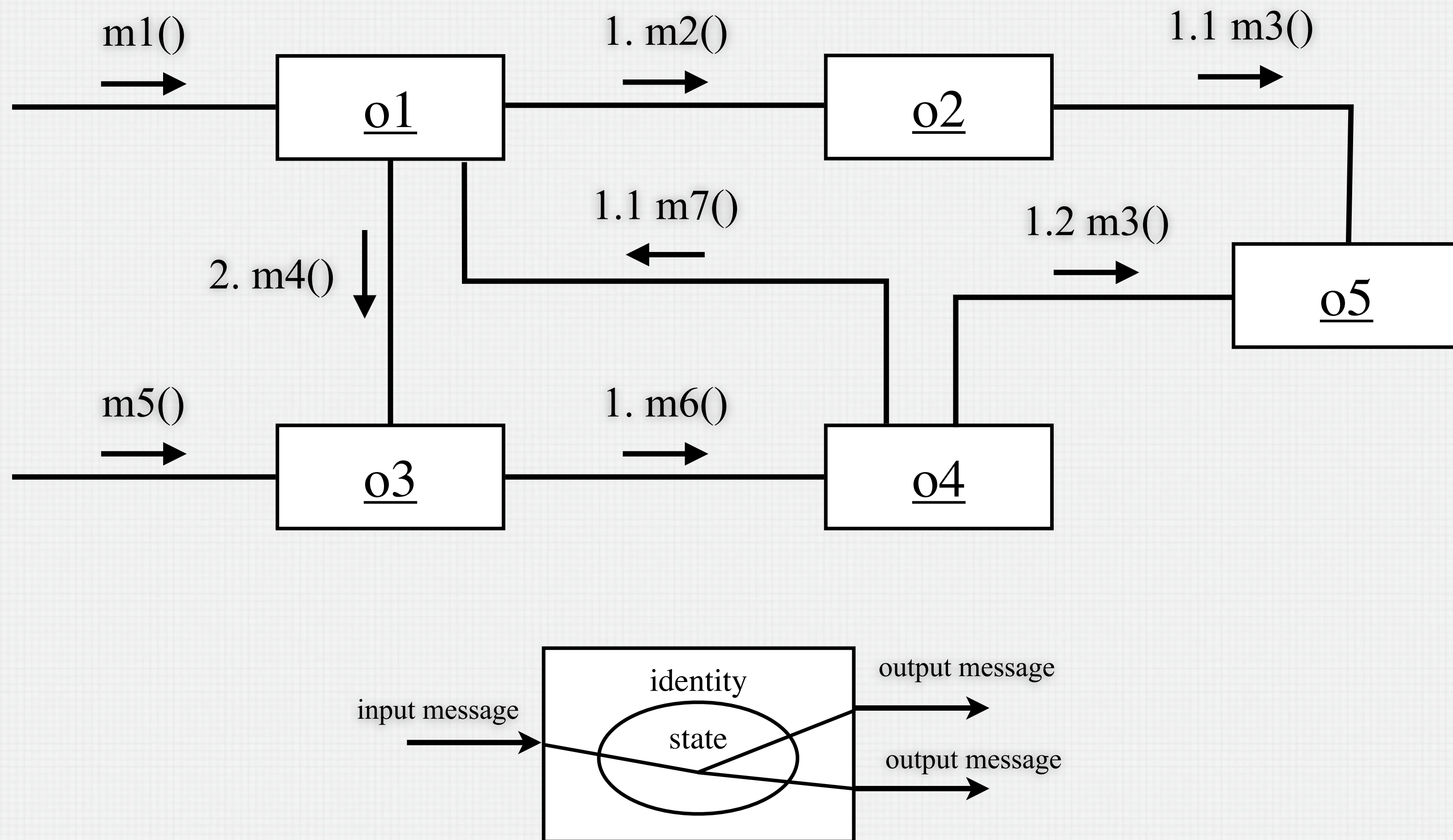
UML Communication Diagram



Object System (1)

- An object system or an object-oriented model is composed of:
  - a set of objects,
  - interactions between these objects.
- The dynamics of the object system is determined by its behaviour at run-time: the operations performed by the objects, the ordering of these operations, the interactions between objects, etc.
- The structure of communication between objects is flat, i.e. a network.

Object System (2)



Class (1)

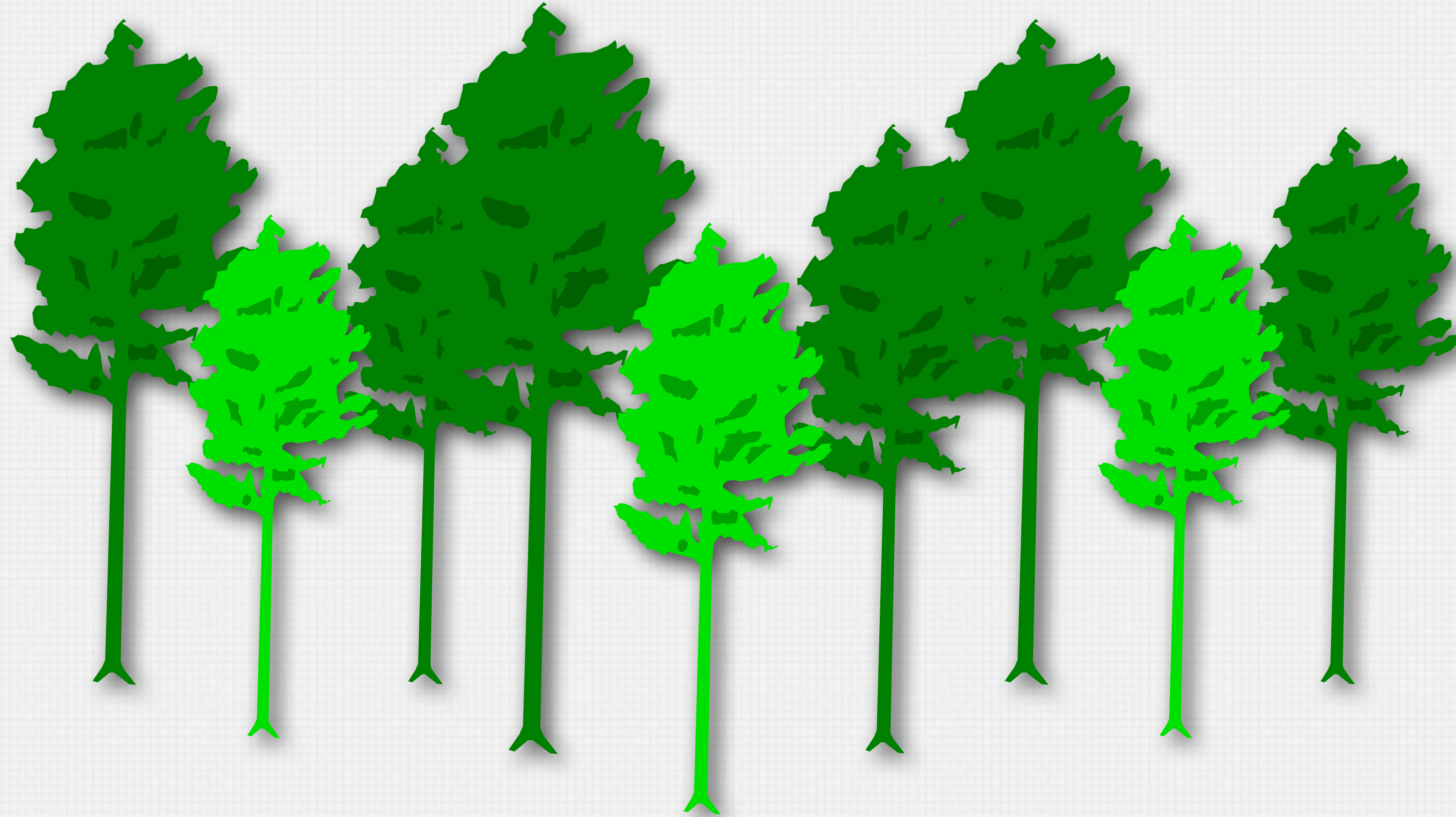
- A class groups objects in such a way that:
  - the similarities can be promoted,
  - and the differences ignored.
- Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of an object, as it were.
- A class can be made of all the objects having the same internal structure, or a similar internal structure, and the same behaviour, or a similar behaviour.
- A class can be made of all the objects having the same attributes and providing the same operations, or having similar attributes and providing a similar set of operations.

Class (2)

- The concept of a class has an economic interest
  - During analysis and design, instead of describing each object individually, it suffice to describe their classes.
  - During implementation, the implementation of a class can be shared by all its objects.
- A class is a template from which objects can be instantiated, i.e. created. We also say that an object is an instance of a class.
- Notice that the identity and the state belong to each individual object.

Class Examples (1)

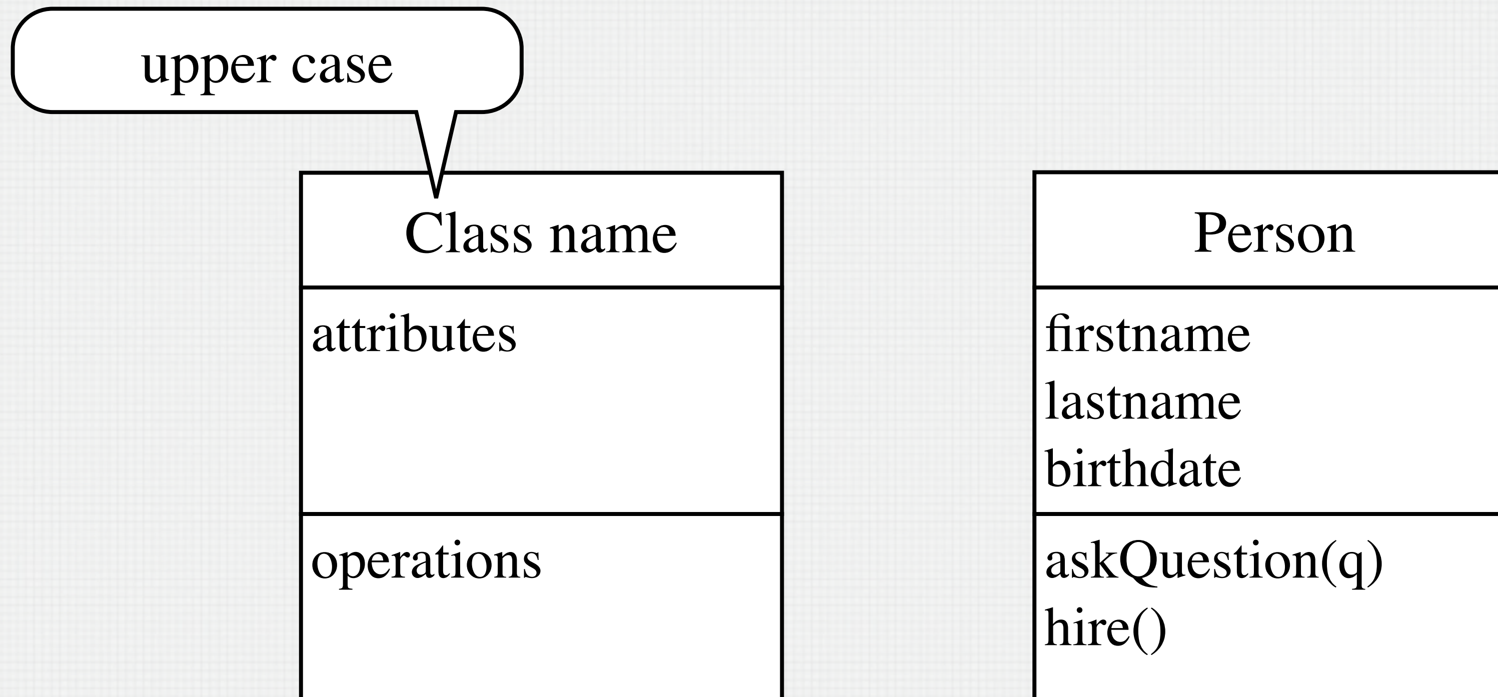
- Despite differences between individual objects, all are trees



Class Examples (2)

- The printer Neo, of type Phaser 4400N, made by Xerox, located in room McConnell 322 …
- Mr. Rich, business man, 42 years old, living in Zug, Switzerland, married with Mrs. Dufour, ...
- The bank account of Mr. Rich with the Swiss Union Bank...

UML Representation of a Class



Intension and Extension (1)

- There are two distinct possible views of a class:
  - Intension of a class
    - The set of properties shared by all objects defines the meaning of the grouping. The class is a template from which objects can be created (instantiated).
  - Extension of a class
    - The set of all objects belonging to a class denotes a population. The class is a collection of objects (instances).

Intension and Extension (2)

Person
firstname lastname birthdate
askQuestion(q) hire()

julie : Person

laura : Person

isabelle : Person

mira : Person

enya : Person

fox : Person

Class Instances

- The instances of a class can be shown in a table, the columns correspond to attribute values

Class notation, showing its intension	Class table, showing a possible extension

Class notation (Professor):

Professor
name subject

Class table (Professor):

name	subject
Martin Robillard	Software Evolution
Doina Precup	Machine Learning
Luc Devroye	Algorithms
Bruce Reed	Percolation

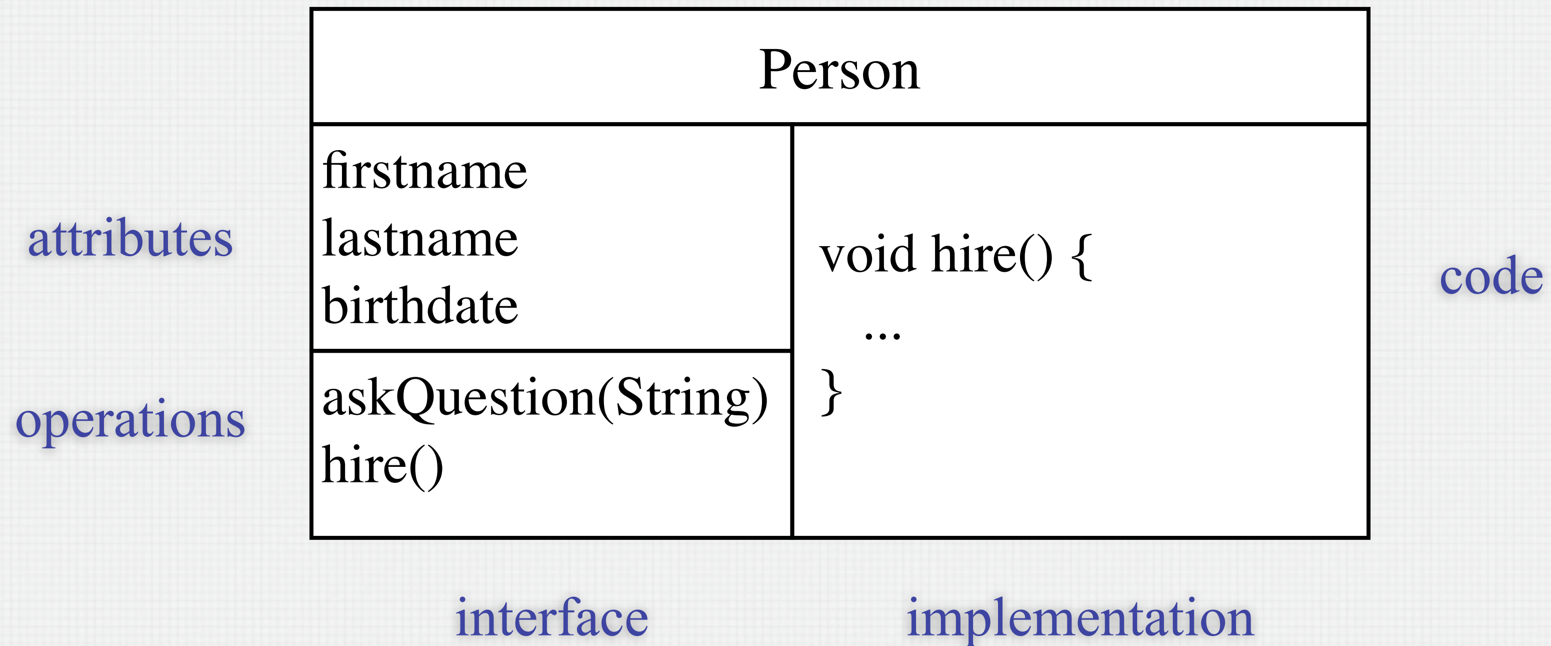
Class Interface

- The interface of a class is the same as the interface of its instances.
- The interface of a class captures its outside view, encompassing the abstraction of the behaviour common to all instances of the class, while hiding their internal structure and the secrets of their internal working.
- The interface of a class comprises all operations applicable to its instances; it may also include object attributes, and other entities needed to complete the abstraction.

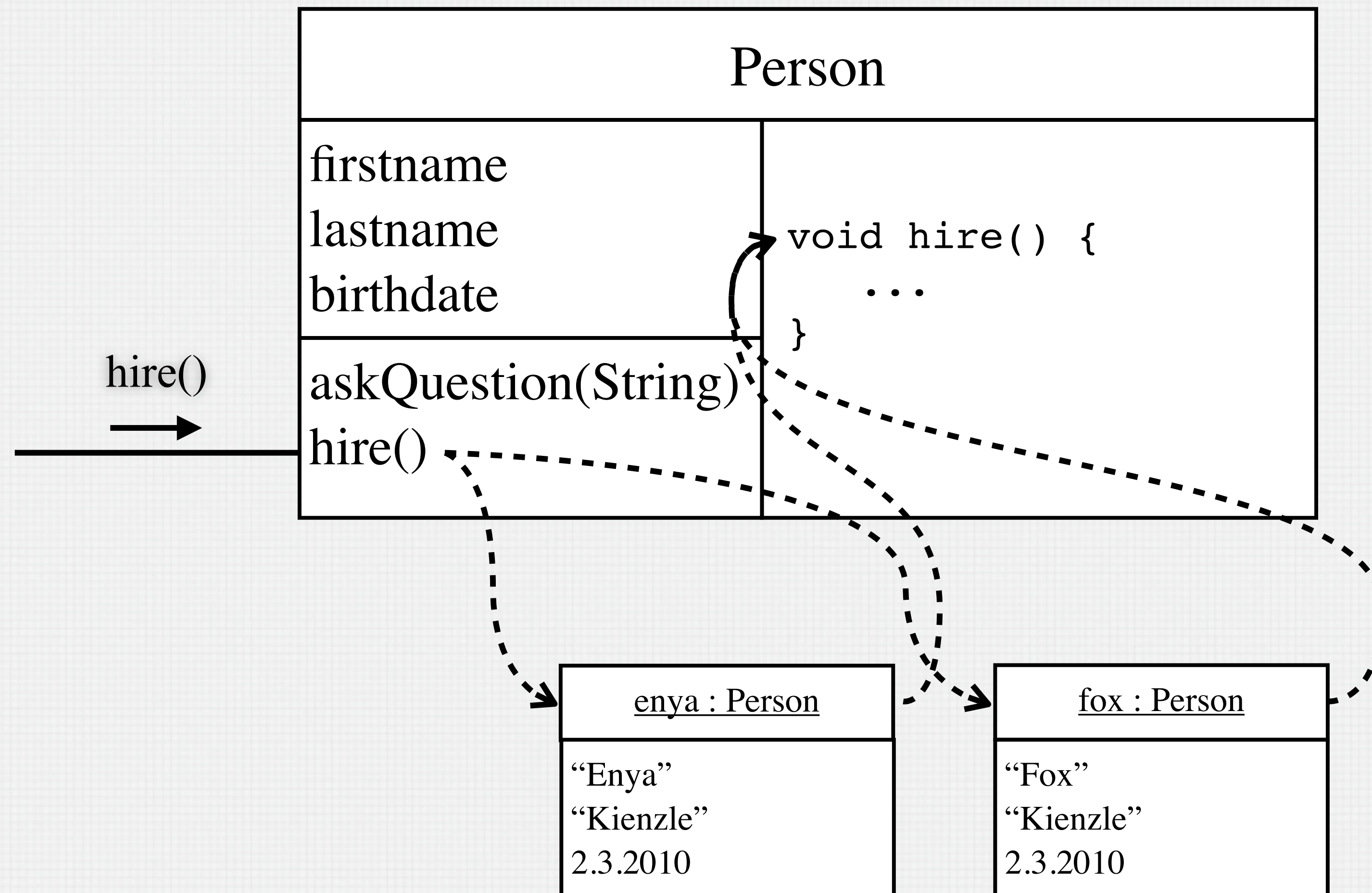
Class Implementation

- The implementation of a class is its inside view, which encompasses the secrets of its behaviour.
- The implementation of a class comprises the mechanisms used to store the state of an instance, as well as the mechanisms used to achieve the behaviour of an instance.
- The implementation of a class primarily consists of the definition of the internal data structure of its instances and of the implementation of all of the operations defined in the interface of the class.
- Each instance has its identity and carries its state, conforming to the data structure defined by its class. When asked to perform an operation, the implementation provided by the class is used.

Class Interface and Implementation



Typical Execution Environment



Interface of a Bank Account

```
class Account is
   operation Create (Money initial)
      precondition: initial > 0.0
      postcondition: balance = initial
   operation Deposit (Money amount)
      precondition: amount > 0.0
      postcondition: balance = old balance + amount
   operation Withdraw (Money amount)
      precondition: balance >= amount
      postcondition: balance = old balance - amount
private
   attribute Money balance
invariant
   balance > 0.0
end class Account
```

Classes As Objects

- A class can be considered itself as an object.
- A class has sometimes a state; the corresponding attributes are called “class variables” in contrast to “instance variables.”
  - For example, a class can keep track of the number of times it is instantiated using a class variable
- A class may also provide “book-keeping” operations for handling its instances, e.g. for creating and destroying an instance.

Type

- A type is a precise characterization of structural and behavioral properties which a collection of entities all share.
- Notice that following this definition, events, methods, subprograms and modules, e.g., may have a type.
- If the entities are objects, then a type and a class are very similar.
- The concept of a type places a different emphasis upon the meaning of abstraction.
- Typing is the enforcement of the rule that entities of different types may not be interchanged, or at the most, may be interchanged only in very restricted ways.

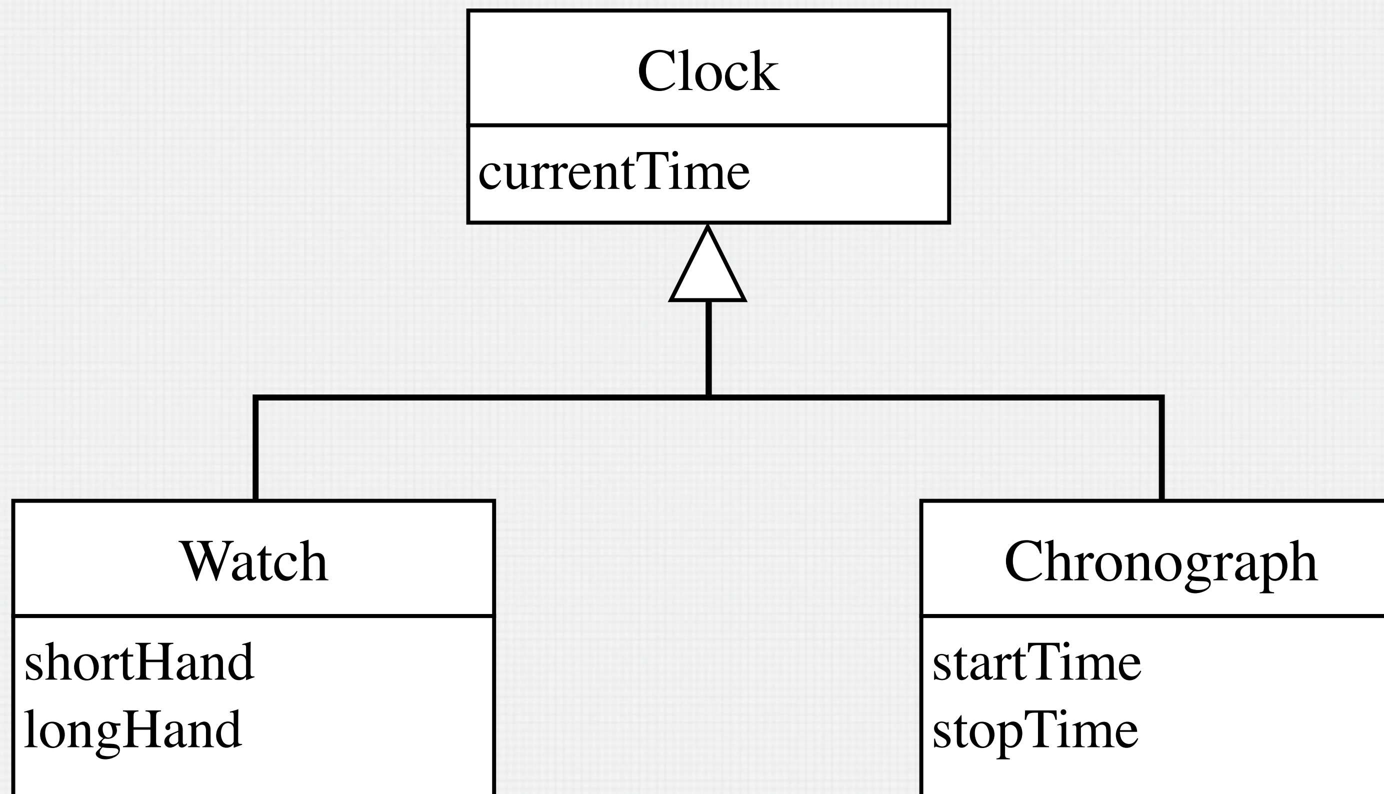
Generalization (1)

- There may be a partial ordering between classes:
  - All objects in class S have all the properties of class T
  - S may have additional properties
- The class S is then said to be a subclass of class T, which is its parent class or superclass
- The relationship between the two classes is called generalization-specialization, subtyping, or inheritance

Generalization Examples

- The printer Neo, of type Phaser 4400N, made by Xerox, located in room McConnell 322...
  - Printer has the subclasses: laser printer, ink-jet printer, daisy printer, etc.
- Mr. Rich, business man, 42 years old, living in Lausanne, Switzerland, married with Mrs. Dufour, ...
  - Person has the subclasses man and woman.
- The bank account of Mr. Rich with the Swiss Union Bank...
  - Bank account has the subclasses: checking account, savings account, fixed term deposit, etc.

Generalization-Specialization in UML

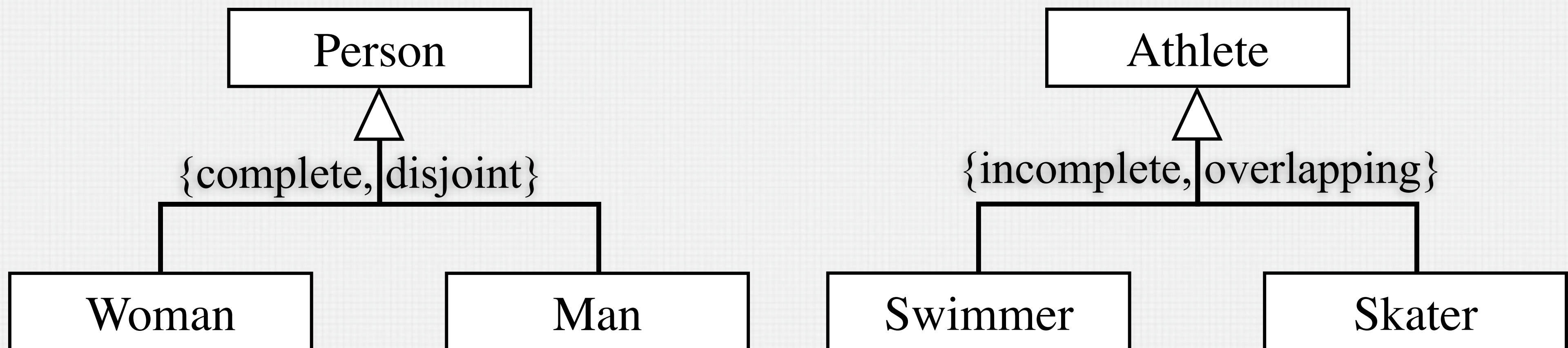


Generalization Comments

- As defined, generalization-specialization is a concept too general to be operational. Here are some more concrete possible definitions:
  - Principle of substitutability: S is a subclass of T, if and only if any instance of T can be substituted by an instance of S, without any visible effect
  - The objects of the subclass have all the attributes and operations of the superclass (and perhaps others)
- Substitutability is important for reasoning.
- Inheritance in object-oriented programming languages does not always enforce this property. Generalization therefore corresponds to a restricted use of inheritance.

Generalization Partitioning

- The subclasses may partition the superclass: an object belongs to exactly one of the subclasses.
- The subclasses may also overlap, and some superclass objects may not belong to any of the subclasses.



UML default is incomplete, disjoint

Generalization vs. Constraints

- A parallelogram is a quadrilateral having parallel opposite sides of equal length. A parallelogram with a right angle is a rectangle. A parallelogram having the four sides of equal length is a rhombus. A square is a rhombus with a right angle, or a rectangle having all sides of equal length.

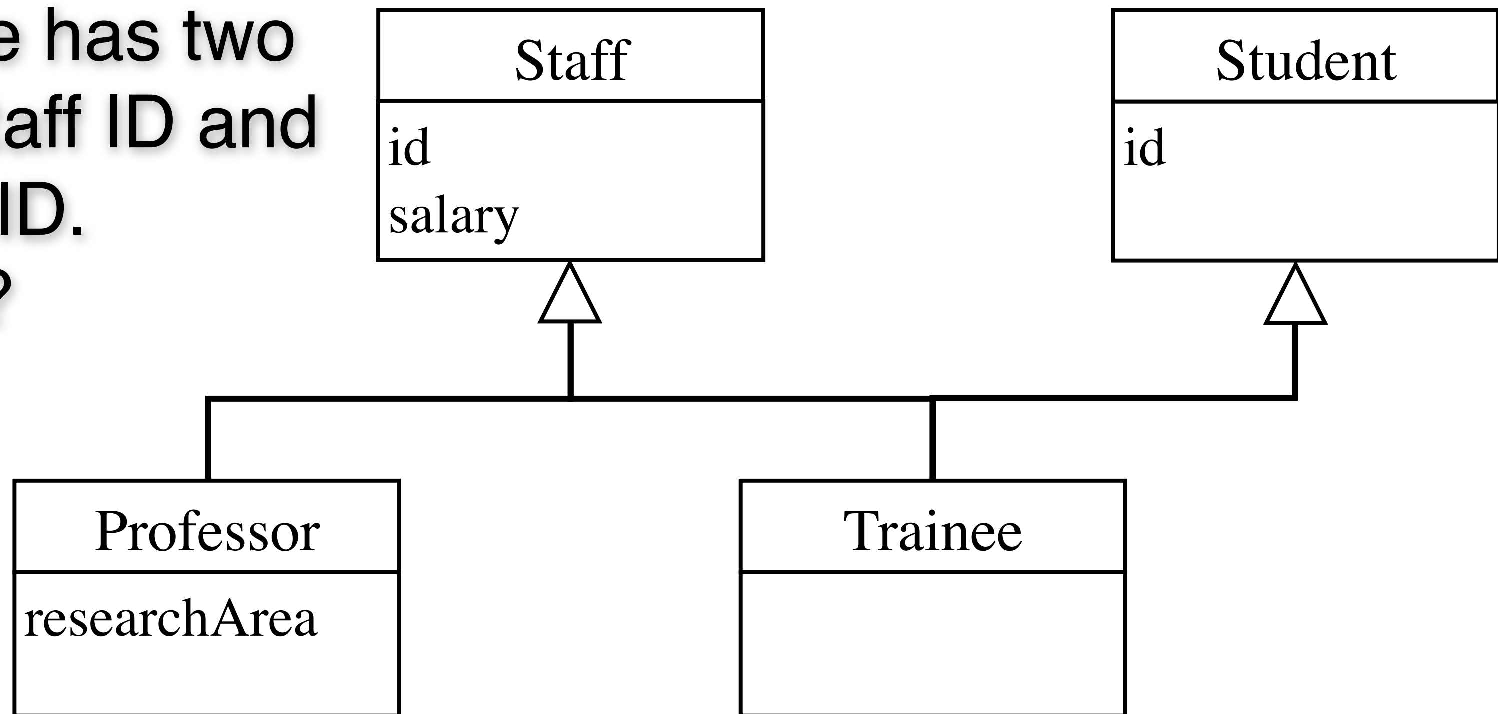
- These statements express constraints!
  - The subclasses don’t really have additional properties (features), but rather satisfy additional constraints.
    - Becomes clear when thinking about operations: e.g., a rectangle can be stretched, but not a square (without making it a rectangle).
  - Don’t use generalization / specialization to model constraints like these!

Multiple Specialization (1)

- Multiple specialization allows a subclass to be defined as a specialization of several immediate superclasses.
- The subclass inherits the attributes, operations and associations of all its superclasses.
- Multiple specialization becomes a problem when two or several superclasses have a common ancestor class (diamond-shaped inheritance).

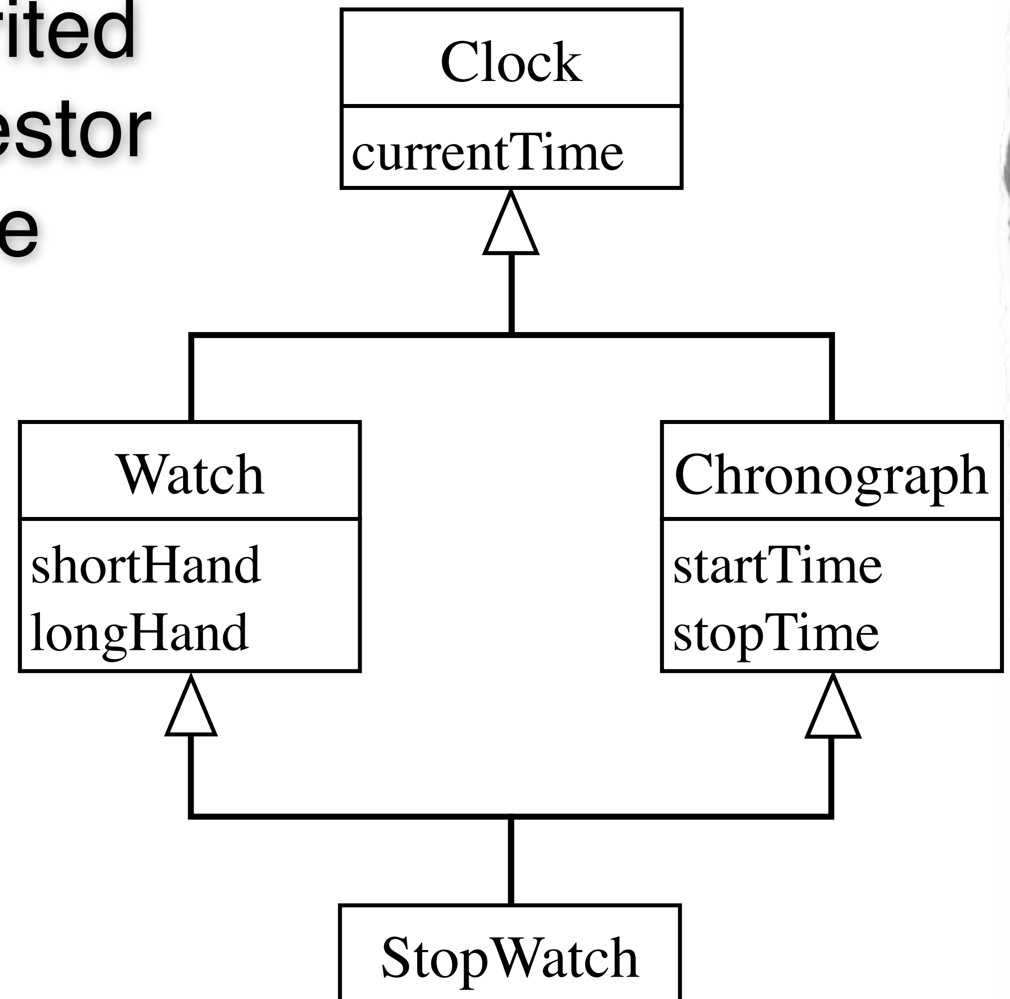
Multiple Specialization (2)

- A trainee is both a staff member and a student.
- As a staff member, a trainee gets a salary, and as a student, s/he gets a grade.
- A trainee has two IDs, a staff ID and student ID. Correct?



Multiple Specialization (3)

- Are the attributes inherited from the common ancestor duplicated, i.e. does the stopwatch have two time attributes?



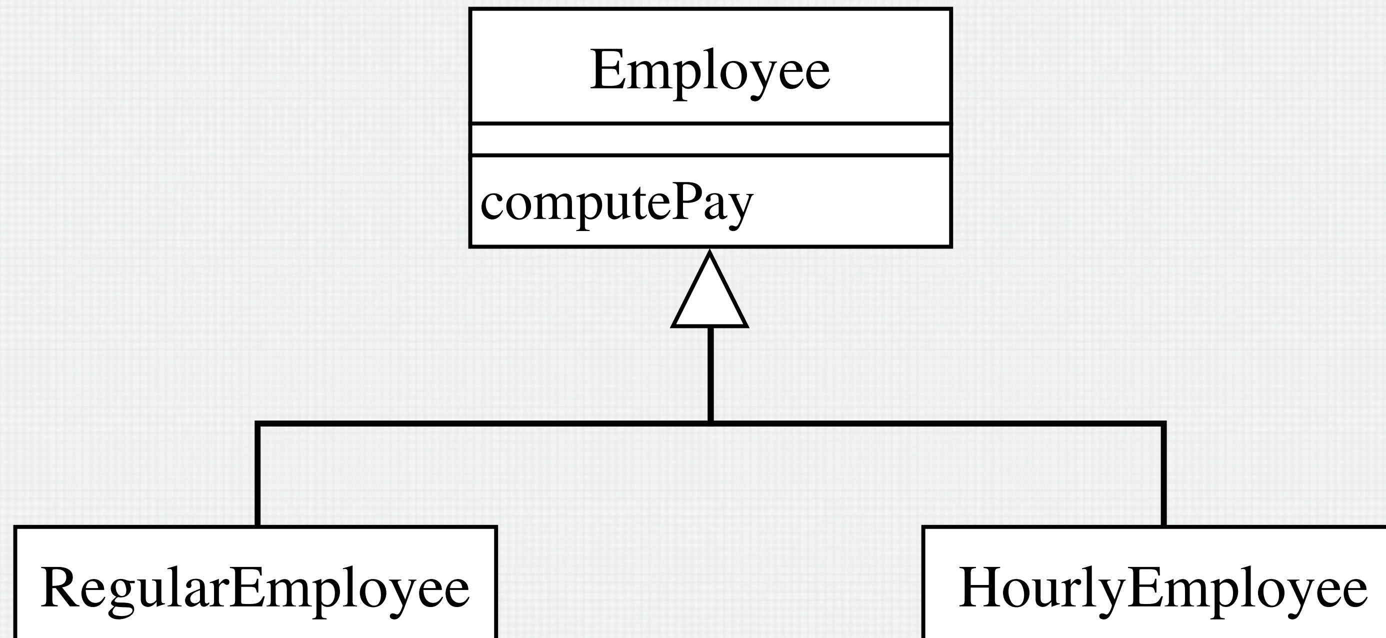
Polymorphism (1)

- Polymorphism is the ability of several classes of objects to respond to the same message in a similar way.
- The message sender does not need to know the specific class of the receiver - only that the semantics of the message will remain the same across many similar classes.
  - Again, OO languages typically do not enforce preservation of semantics for overridden methods.

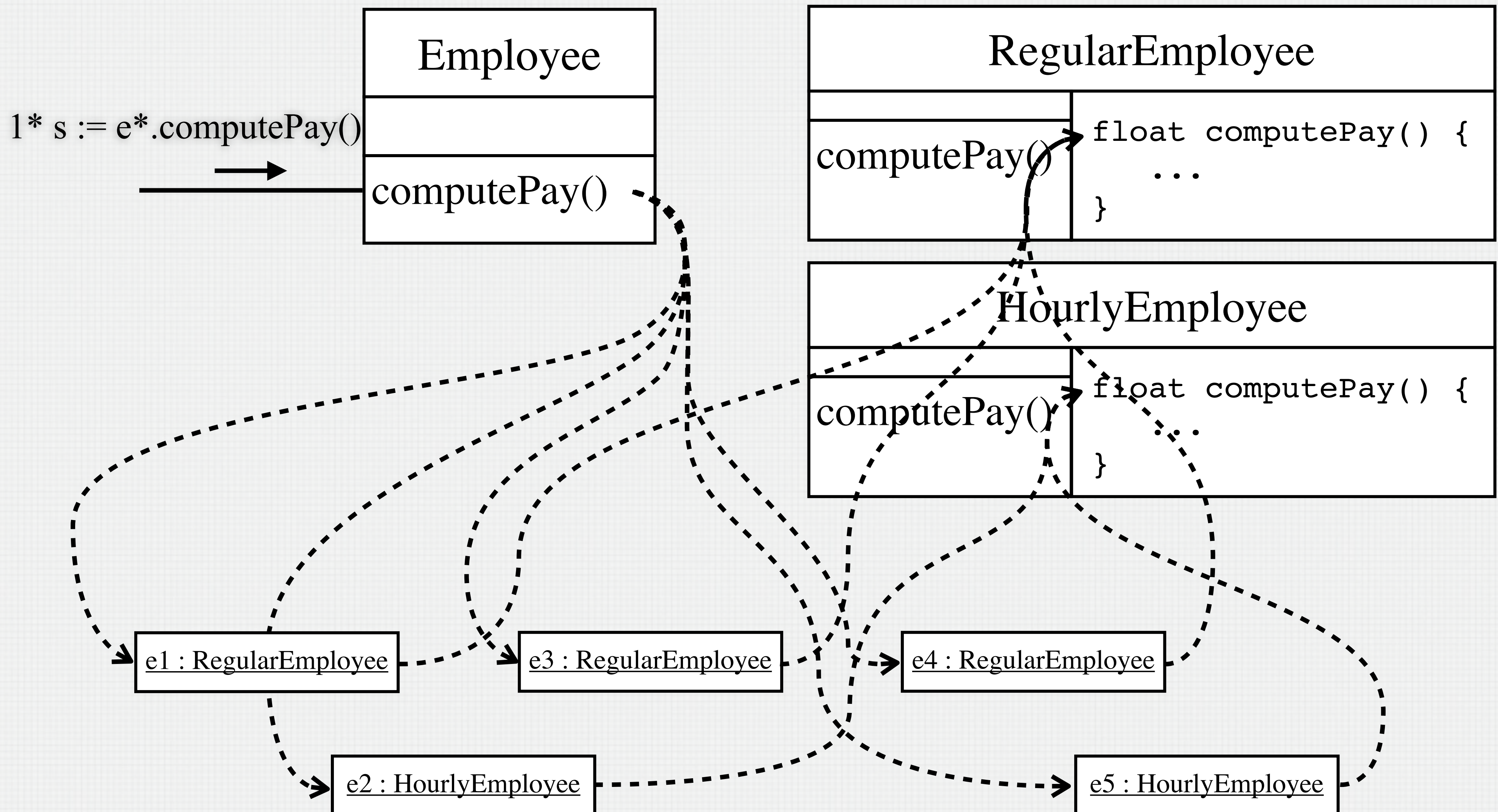
Polymorphism Example (1)

- A payroll system typically will process all employees one after another.
- Suppose there are two kinds of employee: regular or hourly. Each has its own way of computing its pay.
- The payroll system simply sends the `computePay` message to each employee in turn, and the employee takes care of computing its own pay according to the implementation of the operation.
- If a new kind of employees comes along, such as a contract worker, it would have its own way of computing its pay. This new employee could be mixed in with all other employees, and the payroll system will not have to be modified to account for this new employee type.

Polymorphism Example (2)



Polymorphism (3)



O-O Summary (1)

- Object-orientation is based on old principles
  - Abstraction
  - Information hiding and encapsulation
  - Modularity
  - Classification
- Object-orientation is based on a few concepts
  - Object
    - Groups together state and behaviour
  - Class
  - Inheritance
  - Polymorphism

Traditional (non-OO) Systems

- Conventional approaches distinguish between operations and data, and generally emphasize one or the other in their decomposition of the problem.
- Structured Analysis & Design
  - Focuses on operations (functions) first, deriving the data structures as a secondary activity. The value is in the functionality. The data are prepared in a form suitable for processing.
- Information Engineering
  - Places a higher priority on data, and drives the development from the perspective of the data to be managed. The data are the main value. Algorithms are trivial as long as all data are available.
- Global data structures shared among modules

OO Summary (2)

- An object system or object-oriented model is composed of:
  - a set of objects,
  - interactions between these objects.
- An object combines both operations and data.
- The object implementations are hidden behind stable interfaces.
- Any change to a data structure only affects the object that encapsulates it.

Why Object-Orientation?

- Software development is a complex task.
- There is a gap between the problem domain and its computerized support system.
- Humans naturally apply an object-oriented view to the world. Objects are more natural than functions or data.
- An object-oriented model bridges the gap between a problem domain and its software solution.

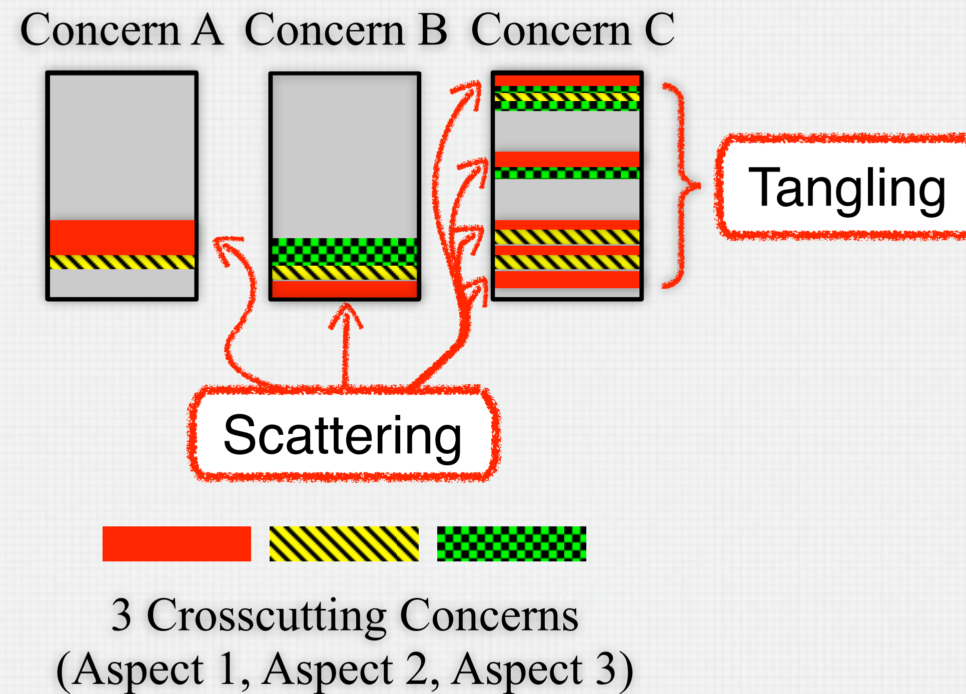
Object-Orientation and SE

- Object-Orientation stems from object-oriented programming, but can be applied within the whole software development life cycle
  - Requirements Elicitation and Specification
  - Design
  - Implementation
  - Testing
- Object-Orientation is a way of thinking about problems using models organized by real-world entities
- Object-Orientation is an engineering method used to create a representation of the problem domain and map it into a software solution

Experience has shown that OO alone is not enough!

- Object-Orientated Software Development
  - Decompose problem into a set of abstractions / objects
  - Objects encapsulate state and behavior
  - Are assigned responsibilities
- "Tyranny of the dominant decomposition" [T+99]
- Result:
  - Similar / identical code-fragments, all implementing some common functionality, are often scattered through the code
  - Within a class, and even within a method, code implementing different concerns is tangled

Object-Oriented Decomposition



Object-Oriented Bank Application

```
class Bank {
  void transfer(Account a, Account b) {
    a.withdraw(100);
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  void withdraw(int amount) {
    balance -= amount;
  }
  void deposit(int amount) {
     balance += amount;
  }
}
```

O-O Bank with Security

```
class Bank {
  void transfer(Account a, Account b) {
    a.authorize();
    a.withdraw(100);
    b.authorize();
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  boolean authorized = false;
  int PIN = ...;
  public void withdraw(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance -= amount;
    authorized = false;
  }
  public void deposit(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance += amount;
    authorized = false;
  }
  public void authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorized = true;
  }
}
```

O-O Bank with Security

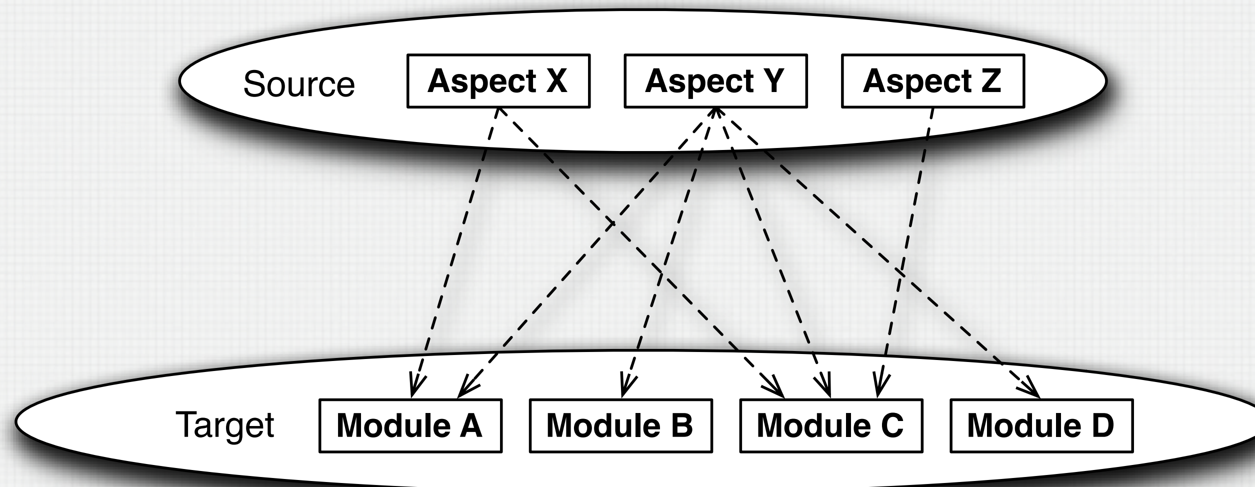
```
class Bank {
  void transfer(Account a, Account b) {
    a.authorize();
    a.withdraw(100);
    b.authorize();
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  boolean authorized = false;
  int PIN = ...;
  public void withdraw(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance -= amount;
    authorized = false;
  }
  public void deposit(int amount) {
    if (!authorized)
      throw new SecurityException();
    else
      balance += amount;
    authorized = false;
  }
  public void authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorize = true;
  }
}
```

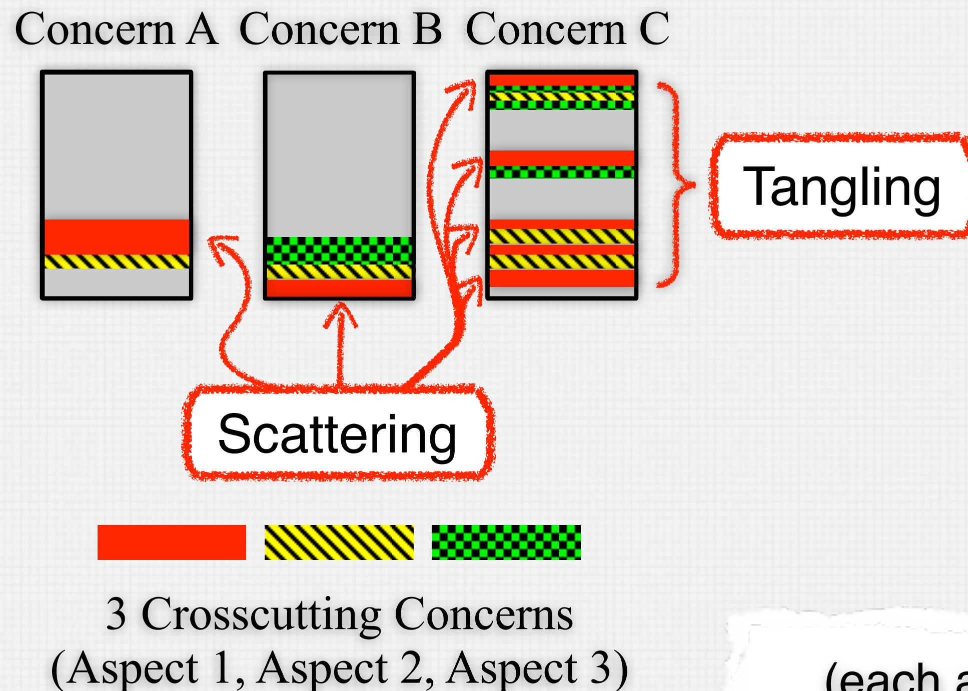
The security concern crosscuts the modules created by the object-oriented decomposition

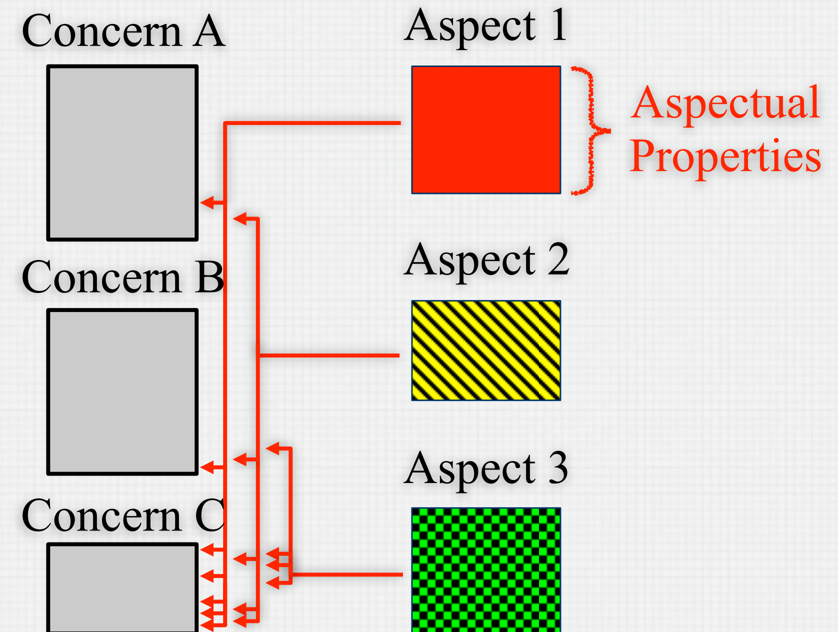
Aspect-Orientation

- Aspect-oriented software development (AOSD) techniques aim to provide systematic means for the identification, separation, representation and composition of *crosscutting* concerns



Essence of Aspect-Orientation

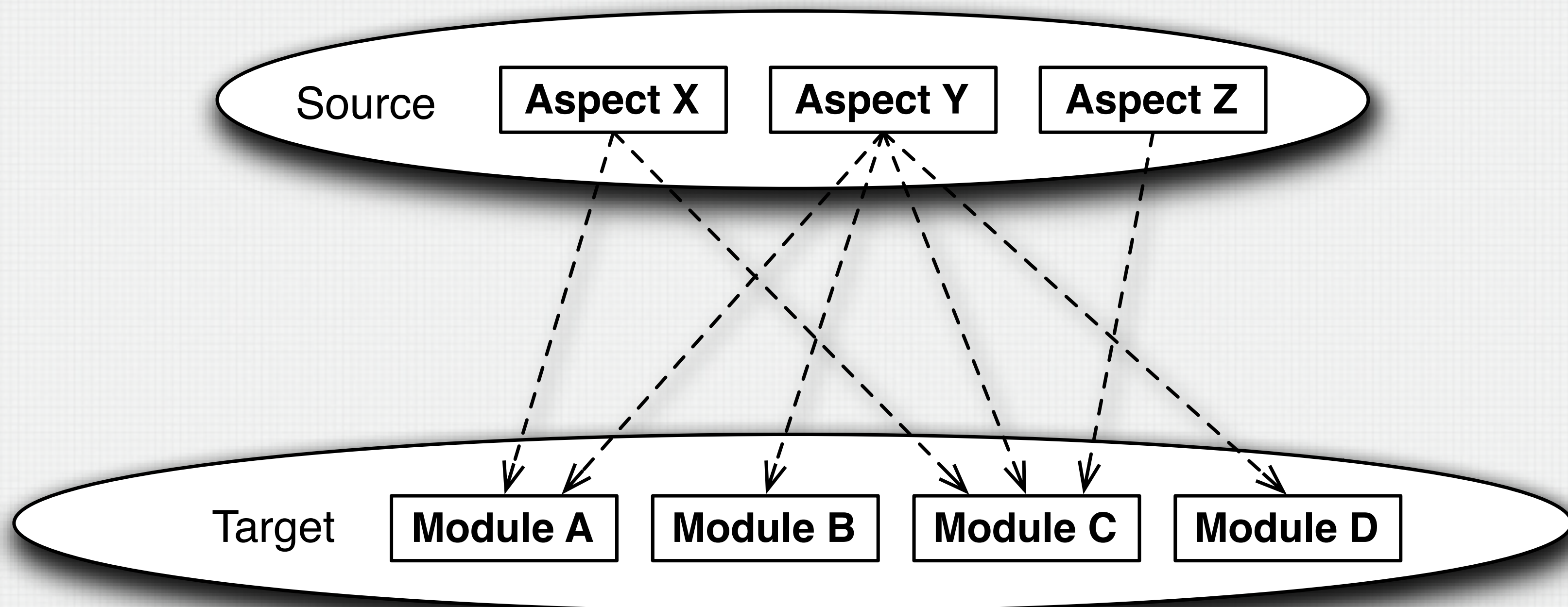




(each aspect contains a composition rule that defines how it is combined with other concerns)

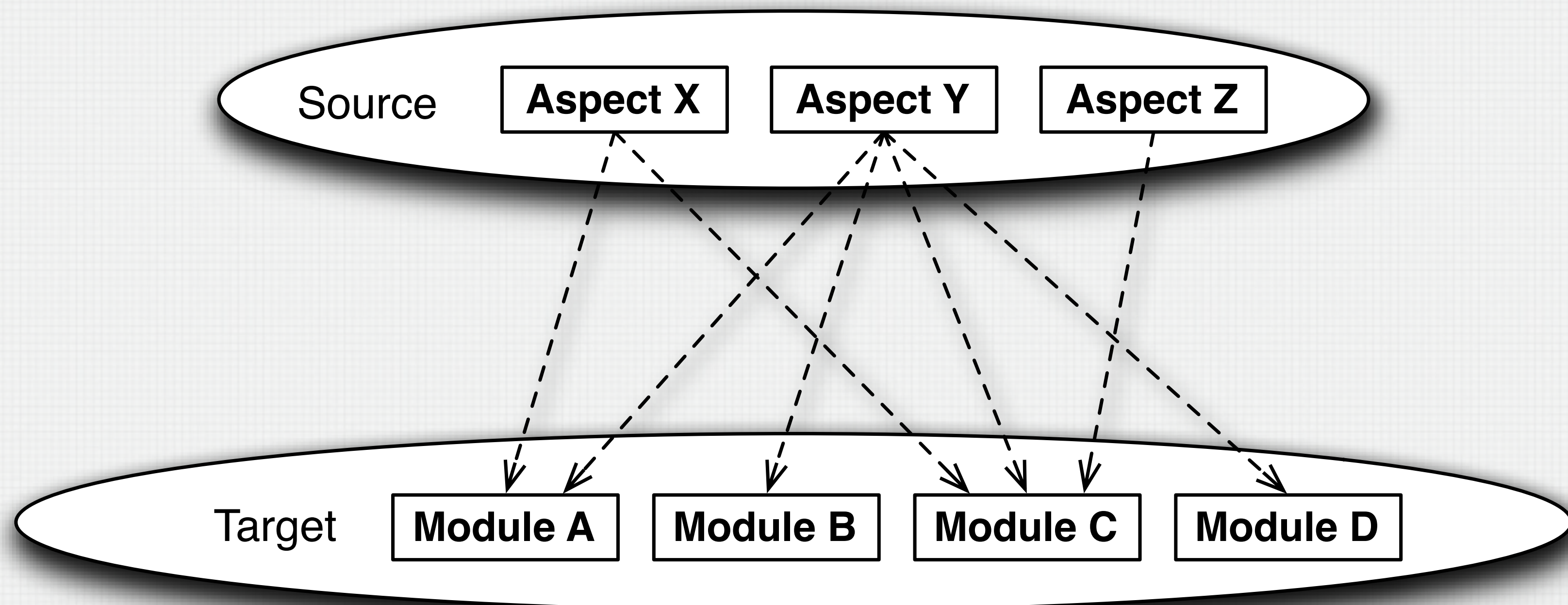
AO Terminology: Weaving

- Mapping (transformation) from a source representation to a target representation



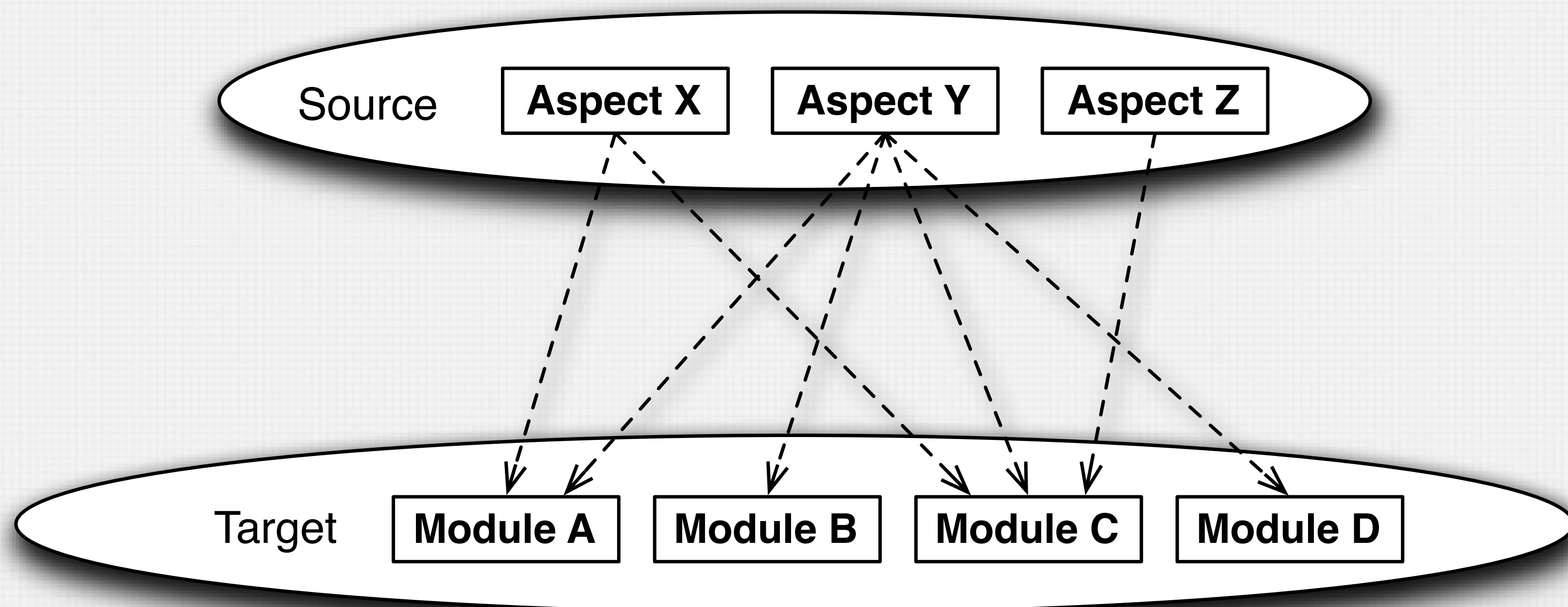
AO Terminology: Scattering

- A source module is scattered in a target representation if part of it ends up in many target modules



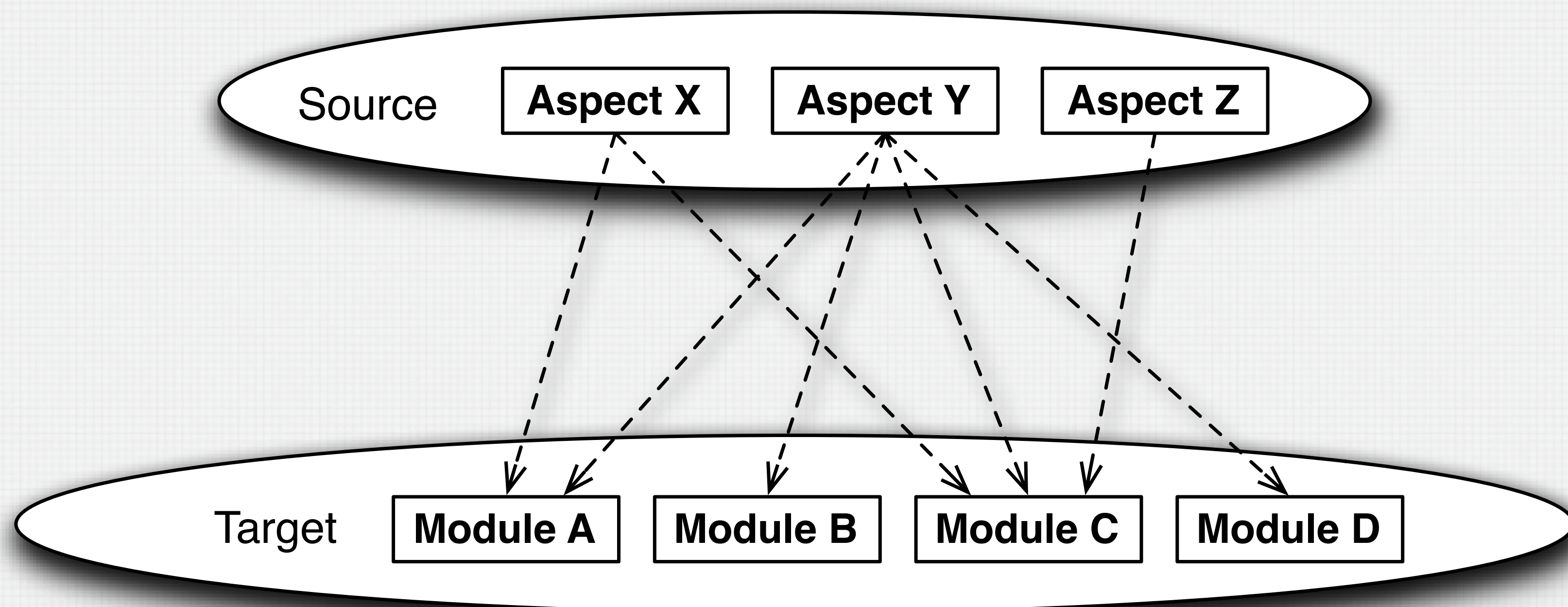
AO Terminology: Tangling

- A target module is tangled if it is composed of parts of several source modules



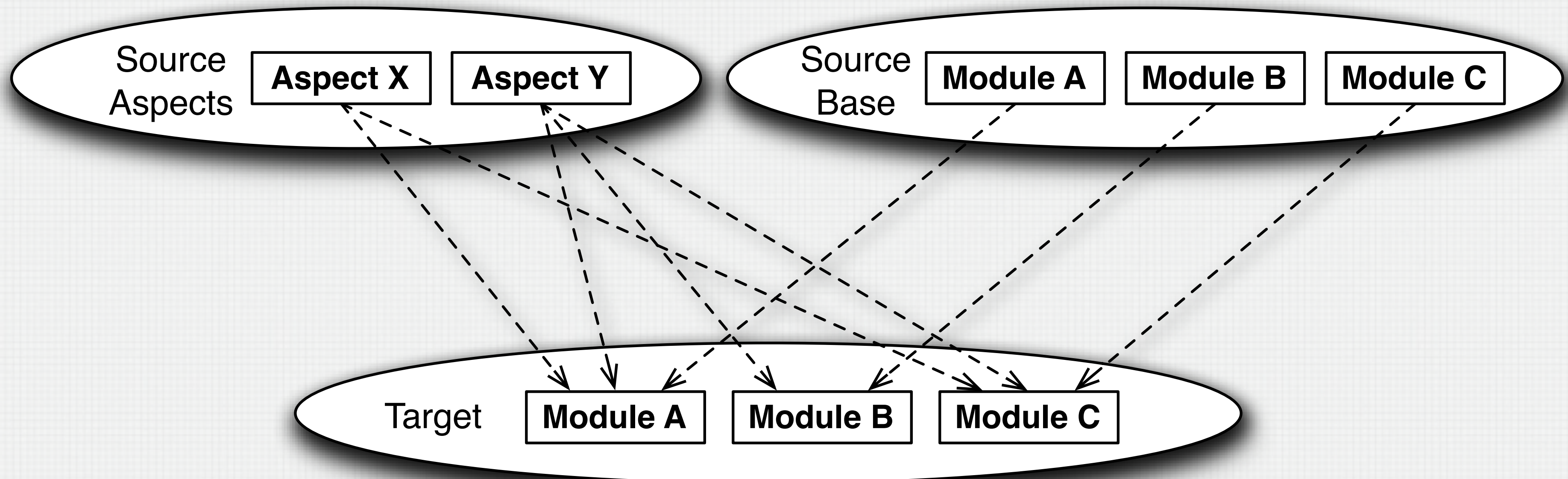
Crosscutting

- X crosscuts Y iff X is scattered in the target representation, and there exists a module in the target within which X and Y are tangled



Asymmetric AO

- Well-identified base elements do not (and are not allowed to) crosscut



Aspect-Orientation History

- Initial Paper in 1997 by Kiczales et al. at ECOOP on aspect-oriented programming (AOP)
- First International Conference on Aspect-Oriented Software Development (AOSD) in 2002
- Today Researchers work on Aspects at all phases of software development
  - “Early Aspects” (Aspects in Requirements)
  - Aspect-Oriented Modeling
  - Aspect-Oriented Programming
  - SPLAT, FOAL, AOM, ADI Workshops
- Industry Adoption
  - Siemens / Motorolla / IBM

Aspect-Oriented Programming

- Provide *new modularization features* at the programming language level *that allow to modularize crosscutting concerns*
- Modules implementing crosscutting functionality are called *aspects*
  - Aspects encapsulate *crosscutting state and behavior*
- Aspects are *woven* together to create final executed code
- Weaving happens at so-called *joinpoints*
- Benefits:
  - Simpler structure, improve readability, customizability and reuse
- Current main-stream aspect languages:
  - AspectJ (www.eclipse.org/aspectj), AspectC#, AspectC++, AspectC

AspectJ

- Aspect-Oriented extension of Java
- Pointcuts
  - Make it possible to name a set of joinpoints, e.g. method calls, setting or getting field values, etc.
- Advice
  - Specify behavior at joinpoints
- Introduction
  - Add fields / methods to classes
- Aspects
  - Group together pointcuts, advice and introductions

AspectJ Joinpoints (1)

- Joinpoints are identified using pointcut designators
- Methods and constructors
  - call(Signature), execution(Signature), initialization(Signature)
  - Example:
    call(public * Account.get*(..))
- Exeption handling
  - handler(TypePattern)
  - Example: handler(TransactionException+)
- Field accesses
  - get(Signature), set(Signature)
  - Example: get(private * Account+.*)

AspectJ Joinpoints (2)

- Objects
  - this(TypePattern), target(TypePattern), args(TypePattern, …)
  - Example:
    call(public * Account.get*(..)) && this(AccountManager)
- Lexical extent
  - within(TypePattern), withincode(Signature)
  - Example:
    call(public * Account.get*(..)) &&
    withincode(public void AccountManager.transfer())
- Based on control flow
  - cflow(Pointcut), cflowbelow(Pointcut)
  - Example:
  - cflow(public void AccountManager.transfer()) && call(public * Account.get*(..))

AspectJ Joinpoints (3)

- Conditional
  - if(Expression)
  - Example:
    if(debugEnabled) &&
    call (public * Account.*(..))
- Combination
  - !, &&, || and ()

AspectJ Pointcuts

- Pointcuts group together a set of joinpoints, and can pass on values from the execution context
- Examples:

```
pointcut PublicCallsToAccount(Account a) :
call(public * Account.*(..)) && target(a);

pointcut SettingIntegerFields(int
newValue) :
set(* int Account.*) && args(newValue);
```

AspectJ Advice

- Add behavior *before*, *after* or *around* a joinpoint
- Example:

```
around PublicCallsToAccount(Account a) {
  if (a.blocked) {
    throw new AccountBlockedException();
  } else {
    proceed();
  }
}
```

AspectJ Introduction

- Even the account class does not provide the "block" functionality, we can add it through introduction
- Example:

```
private boolean Account.blocked = false;
public void Account.block() {
  blocked = true;
}
public void Account.unblock() {
  blocked = false;
}
```

AspectJ Aspects

- Aspects group everything relevant for implementing a particular concern

```
public aspect BlockableAccounts {
  pointcut PublicCallsToAccount (Account a) :
    call(public * Account.*(..)) && target(a);
  private boolean Account.blocked = false;
  public void Account.block() {
   blocked = true;
  }
  around PublicCallsToAccount(Account a) {
    if (a.blocked) {
      throw new AccountBlockedException();
    } else {
      proceed();
    }
  }
}
```

Aspect-Oriented Bank with Security

```
class Bank {
  void transfer(Account a, Account b) {
    a.withdraw(100);
    b.deposit(100);
  }
}
```

```
class Account {
  int balance;
  public void withdraw(int amount) {
    balance -= amount;
  }
  public void deposit(int amount) {
     balance += amount;
  }
}
```

```
aspect AccountSecurity {
  boolean Account.authorized = false;
  int Account.PIN = ...;
  void Account.authorize() {
    p = GUI.askForPIN();
    if (p == PIN) authorize = true;
  }
  before (call public * Account+.*(..)
    && target(currentAccount)) {
    if (!currentAccount.authorized)
      throw new SecurityException();
  }
  after (call public * Account+.*(..)
    && target(currentAccount)) {
    currentAccount.authorized = false;
}
```

The security concern is nicely modularized within the *AccountSecurity* aspect

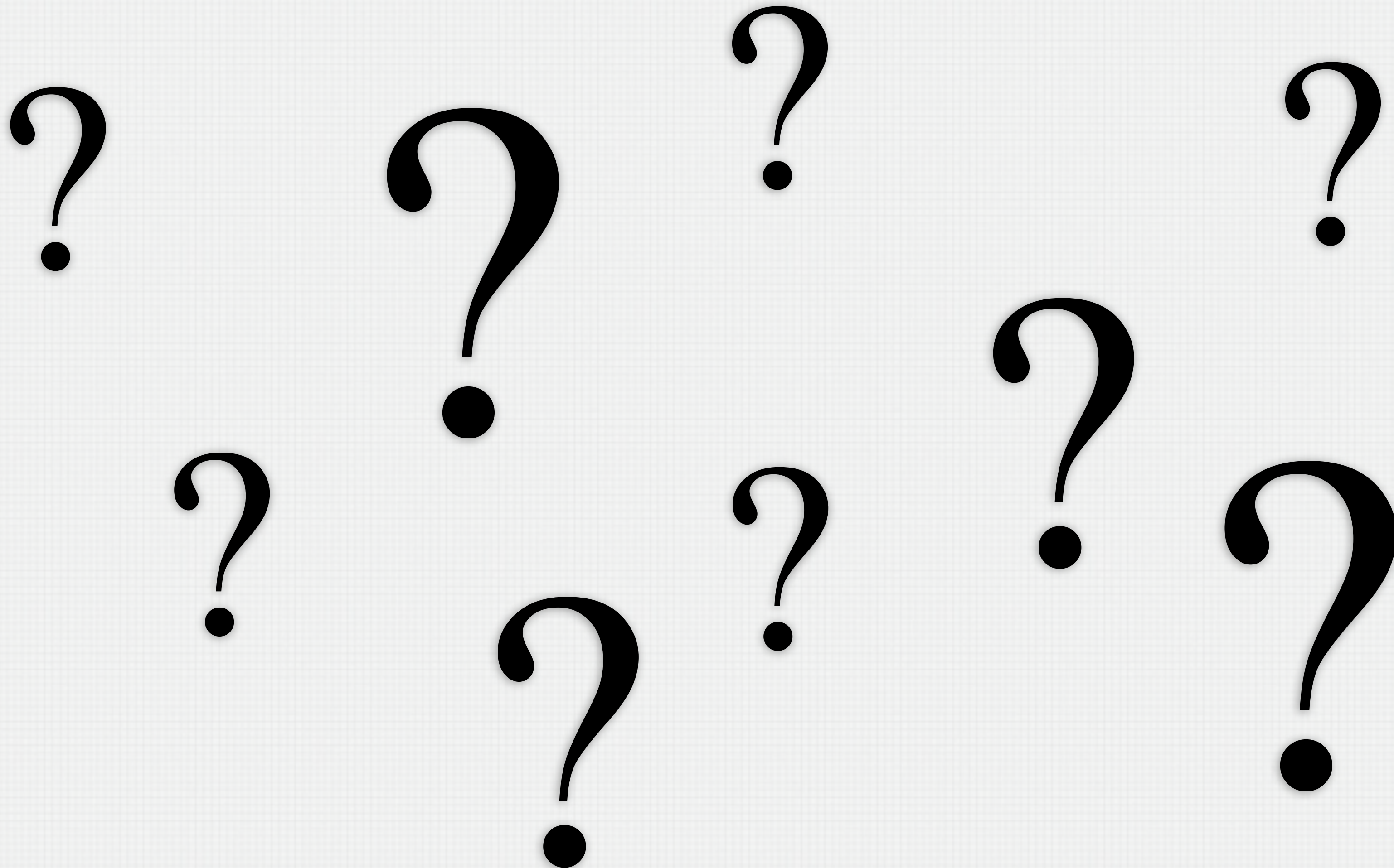
Advanced AspectJ

- Abstract aspects and pointcuts
- Implementing interfaces / inheritance
  - **declare parents** : `Account` **implements** `Blockable;`
  - **declare parents** : `Point` **extends** `GeometricObject;`
- Compile-time checking, e.g. for verifying programming conventions
  - **declare error** : `Pointcut : String;`
  - **declare warning** : `Pointcut : String;`
- Reflective access to run-time information through the static `thisJoinPoint` variable

AspectJ: How to Get Started

- http://www.eclipse.org/aspectj/
  or http://www.aspectbench.org
- AspectJ plug-ins exist for
  - Eclipse
  - Emacs
  - Sun Forte
  - JBuilder
- Version 1.0: Compile-time weaving
  Version 1.1: Byte-code weaving
  Current version: AspectJ 1.7.3 supports Java 1.7

Questions?



Homework

- Savings Account Interface
- Extension or Intension
- Professors and Students
- ETR 407
- Reflection on Aspects

Interface Question

- Find interface attributes for a savings account.

- Find interface operations for a savings account.

Extension or Intension Question

- For each statement, say if it is about the "intension" or the "extension" of a class, attribute or operation.
  - A professor has a name and teaches a subject.
  - Jörg is a professor.
  - He teaches software engineering.
  - Yesterday, he asked John, a junior student, to explain the difference between the extension and the intension of a class.
  - Since John is a clever student, he was able to answer the question.

Professors and Students Question

- Professors can ask students questions.
  1. What are the consequences for the interfaces of the classes Professor and Student? Sketch these interfaces using your favourite object-oriented programming language.
  2. Show on an example how professor Jörg can ask the student John a question.

ETR Question

Find classes, objects and attributes in the following description:

The 407 Express Toll Route is a highway that runs east-west just north of Toronto, and was one of the largest road construction projects in the history of Canada. The road uses a highly modern Electronic Toll Collection (ETC) system constructed by Raytheon.

The ETR technology allows motorists to pass through toll routes without stopping or even opening a window. To make this happen, each highway entry and exit point is equipped with a gantry.

The most cost-efficient way to pay for highway use is to open an account with the 407 ETR system. Registered vehicles require a small electronic tag, called a transponder, to be attached to the windshield behind the rear-view mirror. Transponders are leased for a small monthly fee. The registration includes the owner's personal data, and vehicle details.

Reflection on Aspects

- Among the following concerns relevant in the context of an on-line store, determine which ones are likely to be aspects
  - Placing an order
  - Security
  - Keeping inventory
  - Sending receipts by email
  - Finding the cheapest way to ship the goods to the customer
  - Error handling
  - Web interface
  - Fault tolerance
  - Notifying a customer that the goods have shipped