An introduction to UML

Frank Tip
IBM T.J. Watson Research Center

Overview

- introduction to UML
- use cases & use case diagrams
- packages & “package diagrams”
- statechart diagrams
- classes & class diagrams

UML: The Unified Modeling Language

- what it is: graphical language for expressing object-oriented designs
  - developed by Booch, Rumbaugh, Jacobson
- Booch, Rumbaugh, and Jacobson also introduced a design process: Rational Unified Process
  - but UML notation can be used with any design methodology, and is useful in its own right
- originally developed by Rational
- standardized by OMG (Object Management Group), currently version 1.4

Goals of the UML

- provide users with a ready-to-use, expressive visual modeling language
- provide extensibility and specialization mechanisms to extend the core concepts
- language-independent and process-independent
- encourage growth of the OO tools market
- support higher-level design concepts such as collaborations, frameworks, patterns, and components
- integrate best practices

Why build (UML) models?

- useful for visualizing a system's architecture
- specifying the structure and/or behavior of a system
- provide guideline for constructing an implementation
- documenting the important design decisions
- facilitate communication between developers
  - common language for expressing design elements
- reverse engineering: reconstruct a model from an existing implementation
  - often to re-implement it in another language
- especially important for large/complex systems, to get a handle on the complexity

Types of UML Diagrams

There are 9 types of UML diagrams, constructed from a fairly small set of common elements:

- class diagram
- object diagram
- component diagram
- deployment diagram

} structural diagrams

- use case diagram
- sequence diagram
- collaboration diagram
- statechart diagram
- activity diagram

} behavioral diagrams

Package Diagrams

- Note: this is not an official UML diagram name, but terminology proposed by Fowler & Scott
- a package diagram shows a set of packages with dependences between them
  - officially, just a special kind of class diagram
- useful for high-level design

Use Cases

- definition [Fowler & Scott]:
  - a scenario is a sequence of steps describing an interaction between a user and a system
  - a use case is a set of scenarios tied together by a common user goal.
- a set of sequences of actions (including variants) to yield an observable result to an actor
  - actor: a user, hardware device or other system that interacts with the system
- usually written in plain English, as a numbered sequence of steps
- useful for:
  - requirements gathering
  - creating designs: finding classes and relationships

Use Case Diagrams

- used for capturing requirements, the first step towards creating a design
  - helpful for finding classes, creating class diagrams
- use case: model of interactions between the system, the user (and possibly with other systems)
  - a use case represents an external view of the system
  - some people make a distinction between business and system use cases
- use case components:
  - name
  - informal description
  - more details later

Example: Use Case

Name: ATM withdrawal

1. enter ATM card
2. enter PIN number
3. system verifies that PIN is correct
4. system asks “show balance” or “withdrawal”
5. customers selects “withdrawal”
6. system asks amount
7. customers enters amount
8. system verifies amount <= available balance
9. system dispenses money
10. system asks if receipt is required
11. customers requests receipt
12. system prints receipt
13. systems returns ATM card

Example: Use Case (2)

Variant: PIN incorrect

At step 2, the system determines the PIN is incorrect

2b. Ask user to re-enter PIN

3b. Return to primary scenario at step 3

Variant: user requests account balance

At step 5, user selects “show balance”

5c. system displays account balance

6c. system asks if other services are required

7c. customers confirms

8c. Return to primary scenario at step 4

Use Cases, continued

- a use case shows what a system does, not how it does it
- keep descriptions short, clear, and precise
- separate main flow of events from alternate and exceptional flows
- state preconditions
  - e.g., “Loan officer is logged on”
- use cases may have extension points
  - e.g., “Place order” with extension point “set priority”

Actors

- an actor is a role that a human, device, or another system plays w.r.t. the system
  - actors do not need to be humans
- actors carry out use cases
  - look for actors, then their use cases
- actors can get information from a use case, or participate in it
- actors are connected to use cases by association only (models communication, which can be bidirectional)

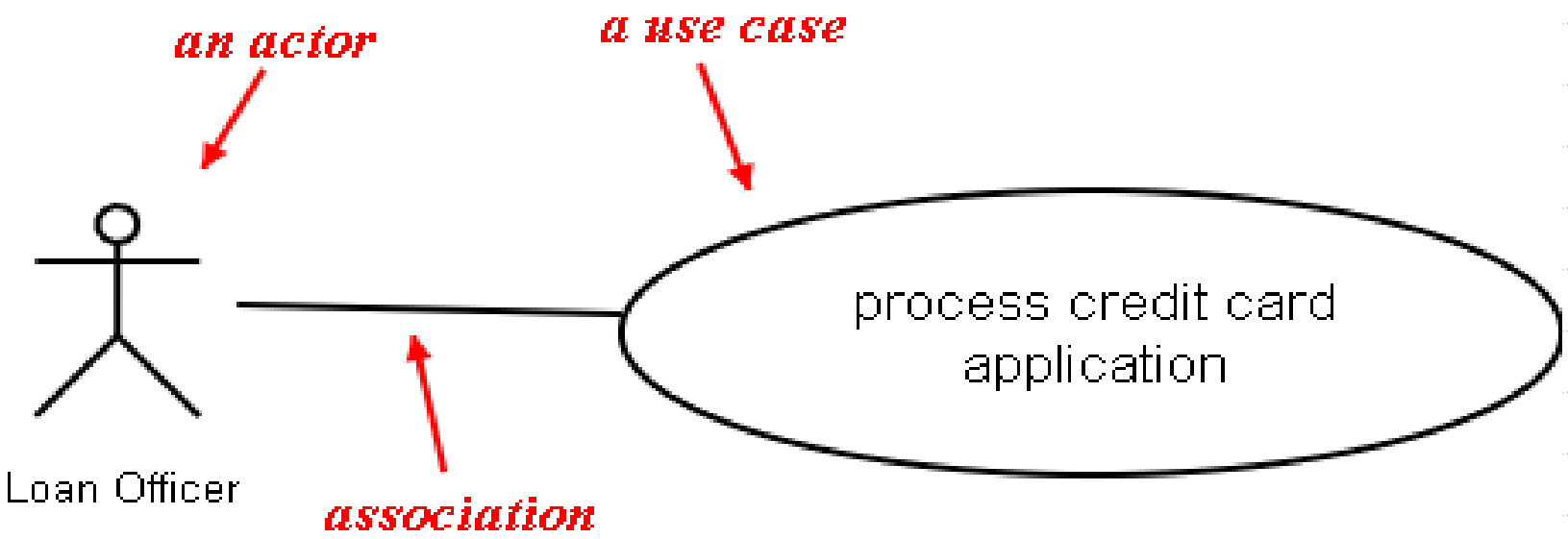
ATM Example

- Actors
  - the customer withdrawing money
  - the database system that contains account information
- Use cases
  - Enter PIN
  - Verify PIN
  - Select Service (withdrawal or balance)
  - Enter amount
  - Check account balance
  - Print receipt
  - Dispense money

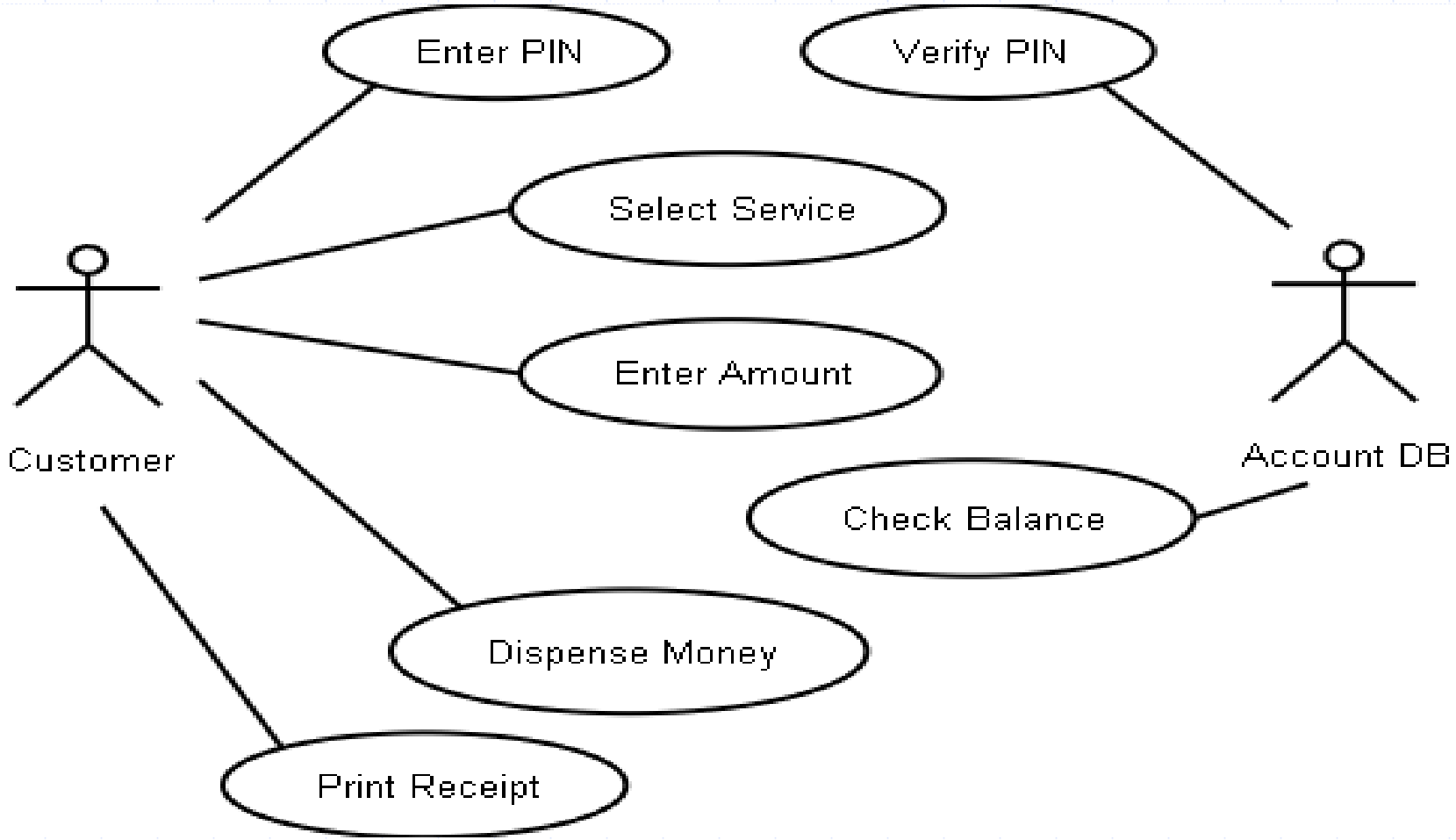
Use Case Diagrams

a use case diagram consist of:

- use cases: oval with text inside
- actors: stick figure
- dependencies, generalizations, associations



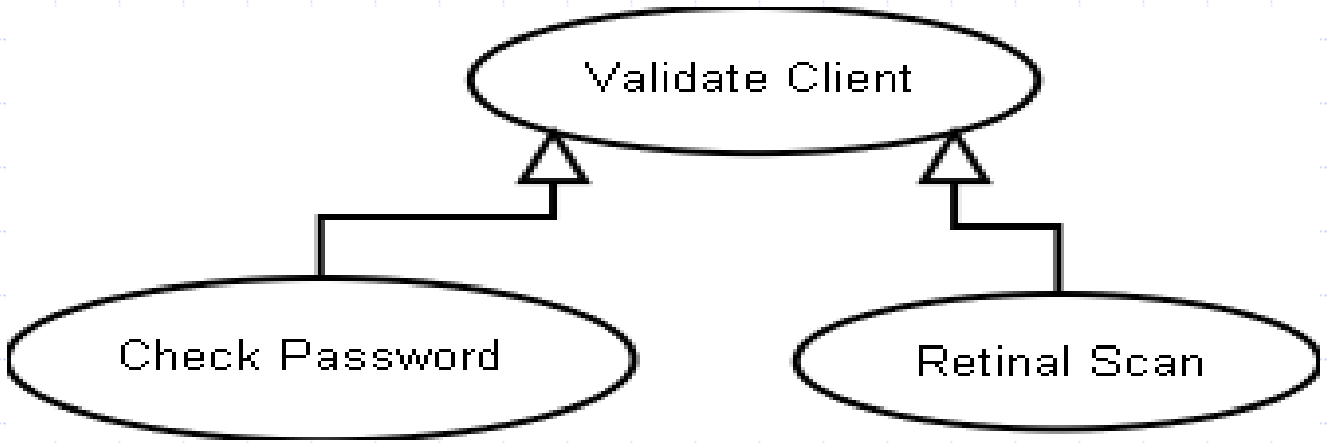
Use Case Diagram for ATM Example



Use Cases: Generalization

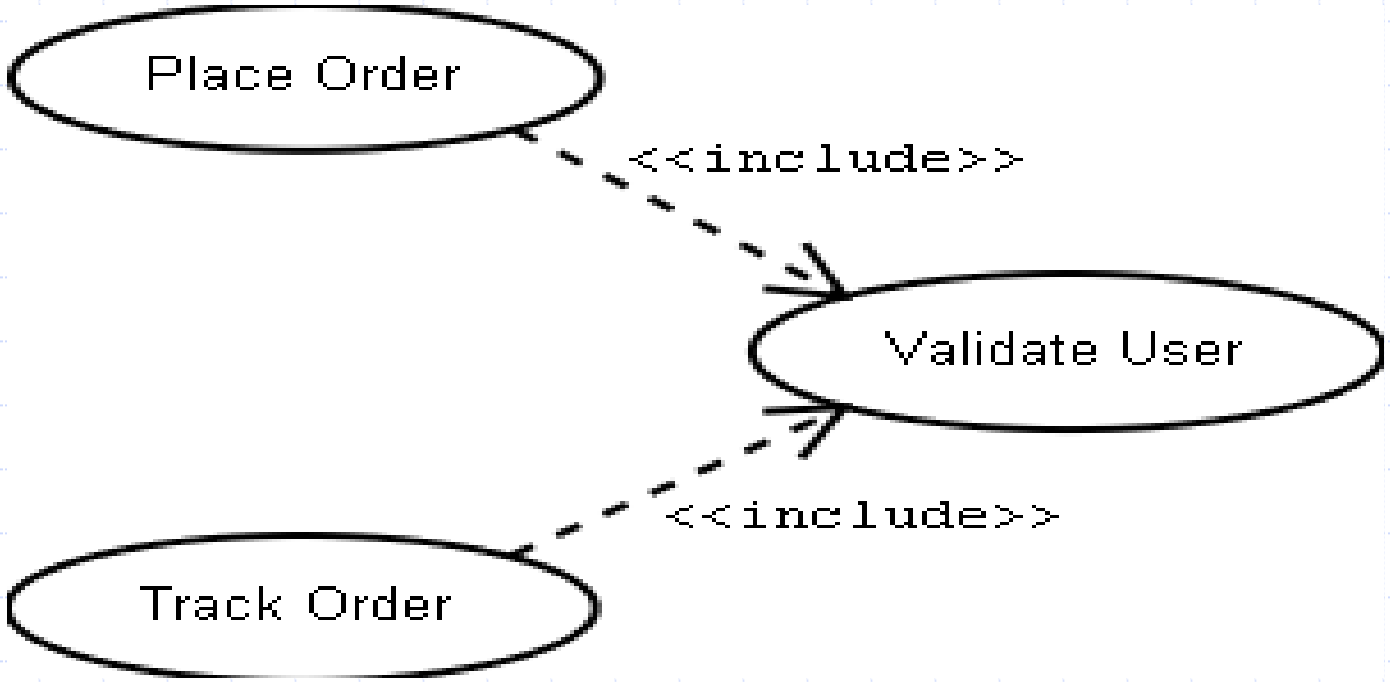
generalization:

- child use case inherits behavior and meaning from parent use case
- child may add or override parent behavior
- child may be substituted where parent occurs
- notation: arrow with open triangle



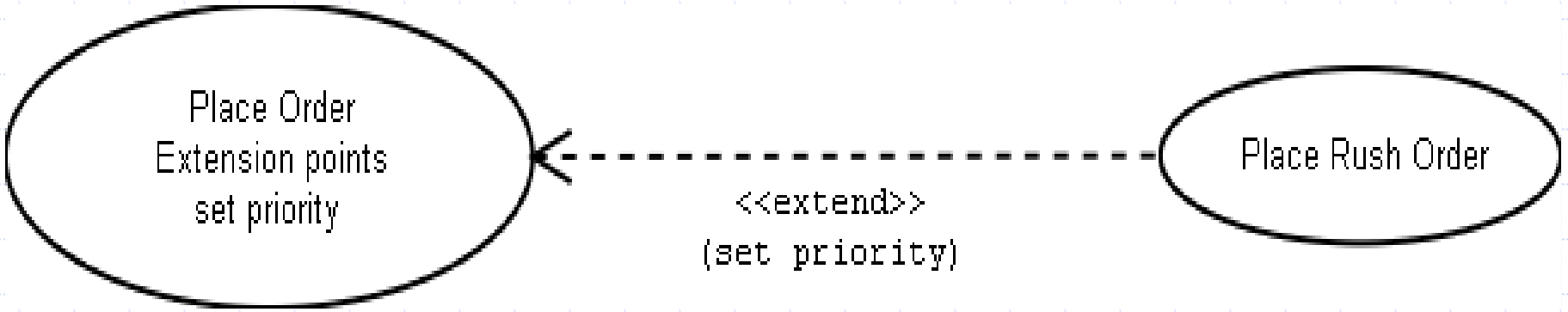
Use Cases: “Include” Relationship

- avoid duplication of the same flow of events by putting common behavior in a use case of its own
- use to avoid copy & paste in use case descriptions

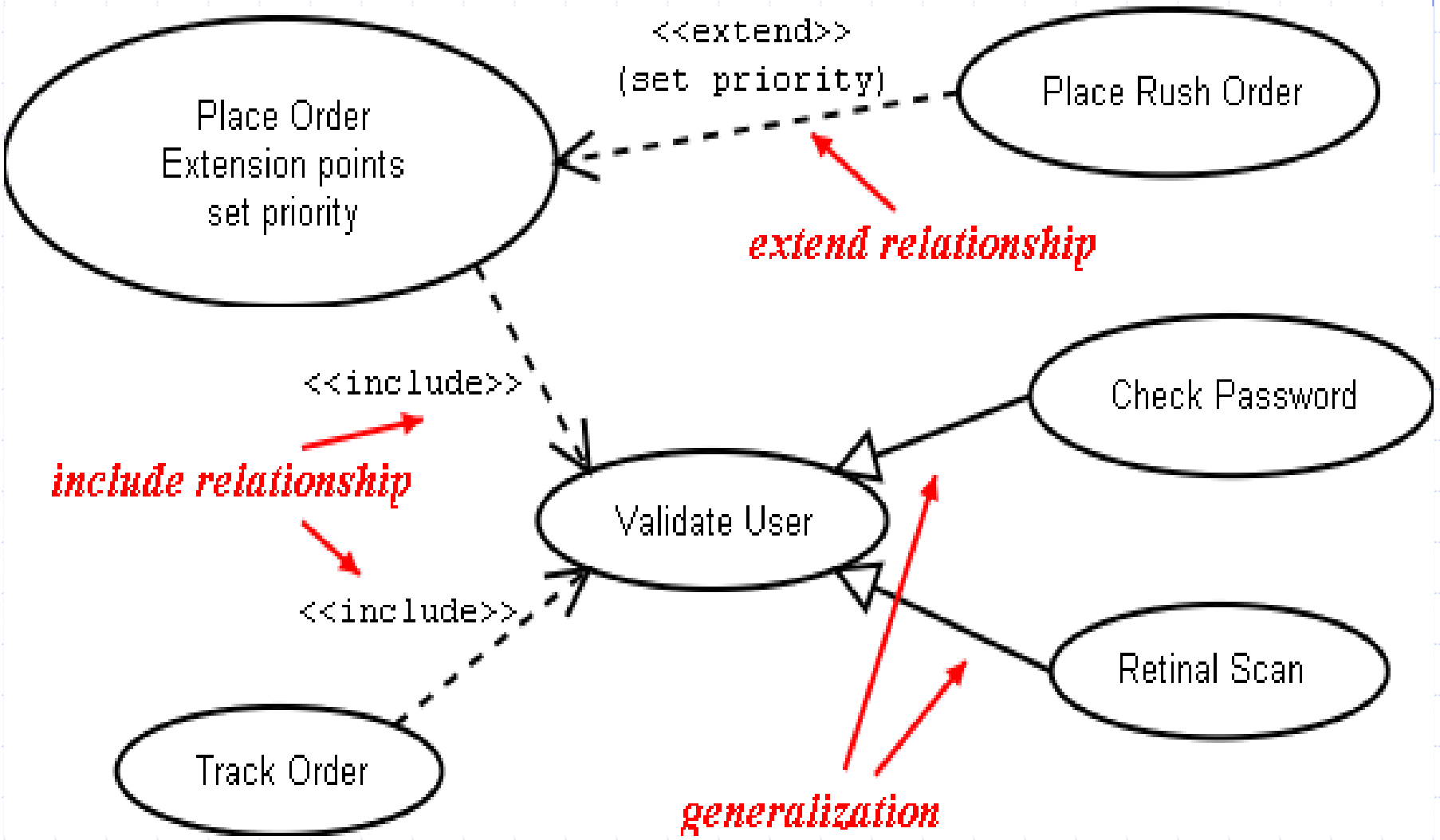


Use Cases: “Extend” relationship

- similar to generalization, but more restricted
- the extending use case may add behavior to the base use case, but:
  - the base use case must declare certain extension points
  - the extending use case may add additional behavior only at those extension points
- primary use: modeling optional system behavior



Example


<<extend>>
(set priority)
Place Rush Order
Place Order
Extension points
set priority
extend relationship
<<include>>
include relationship
<<include>>
Check Password
Validate User
Retinal Scan
Track Order
generalization

Use Case Relationships: Guidelines

Guidelines for choosing relationships (taken from Martin Fowler's UML Distilled):

- use include to avoid repetition when you are repeating yourself in two or more separate use cases
- use generalization when you are describing a variation on normal behavior, and you wish to describe it casually
- use extend when you are describing a variation on normal behavior and you wish to use the more controlled form, declaring your extension points in the base use case

Use Cases

- Q: What format should I use to describe my use cases?
- A: Please supply at least the following details:
  - name
  - main scenario (as numbered sequence of steps)
  - alternatives (if you like, you may turn these into separate use cases)
- optional:
  - primary actor
  - stakeholders & interests
  - special requirements
  - frequency of occurrence
  - preconditions & postconditions
  - technology & data variations
  - open issues
- most important: cover all functionality & all alternatives

Use Cases: Multiple Levels of Detail

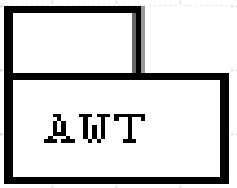
- Craig Larman's book "Applying UML and Patterns: An Introduction to OO Analysis and Design and the Unified Process" distinguishes 3 "levels" of precision for use cases:
  - brief: terse one-paragraph summary
  - casual: informal paragraph format
    - like Fowler, but in paragraph form without numbering
  - fully dressed: all steps & variations written in detail, with supporting sections such as preconditions, etc.

Packages

- packages are a general-purpose mechanism for organizing elements into groups.
  - not necessarily restricted to classes
  - hierarchical model: subpackages
  - anonymous “root” package contains all top-level packages
- a package provides a namespace
  - names of package elements are qualified using “::” (C++ style)

Packages (2)

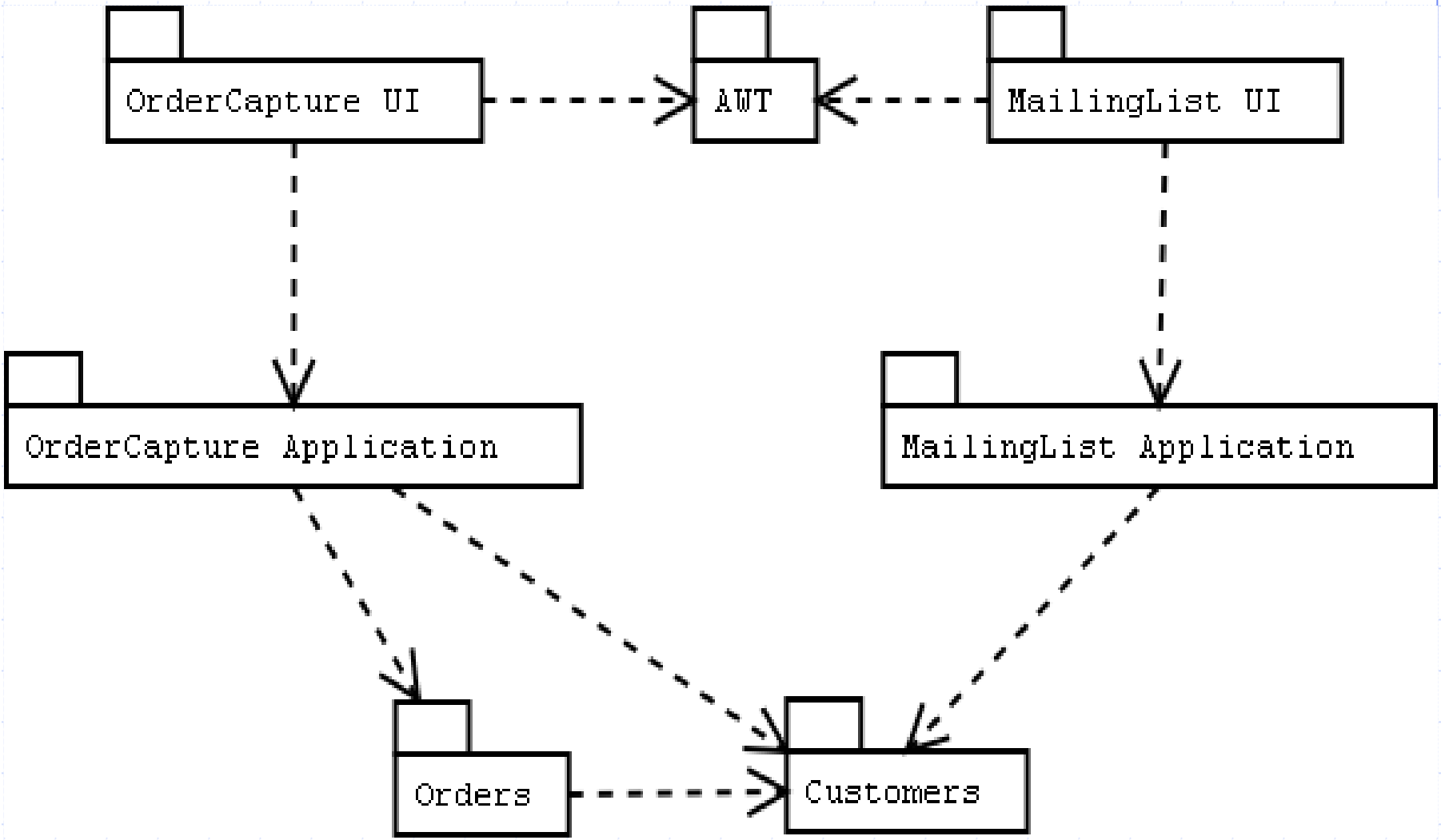
- when to use packages?
  - use packages for high-level design or architecture documents to describe a system's overall structure
  - use packages when class diagrams become too large or cluttered
  - also convenient units for testing
- Notation: tabbed folder



Package Diagrams

- packages
- dependencies between packages:
  - if changes to one package may cause changes to the other
  - reflect dependencies between classes in the packages
    - class in package A calls method in package B
    - class in package A has field of type in package B
    - method in package A has parameter/return type of package B
  - import relationships (also: access relationships)
  - use dashed arrows for dependencies, optionally annotated with `<<import>>` stereotype
- design consideration: minimize dependencies between packages, especially cycles
- note: dependency between packages is not transitive

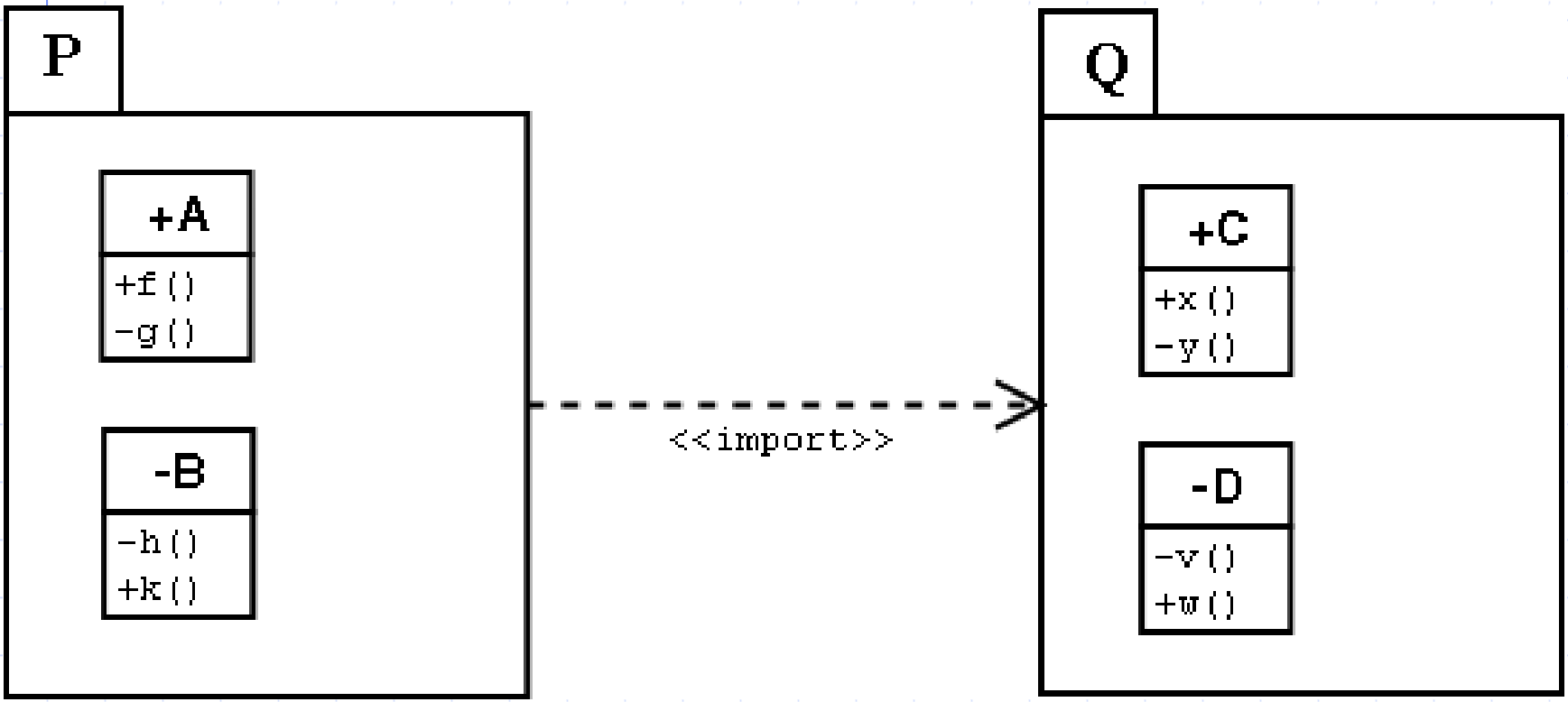
Package Diagram: Example



Example taken from “UML Distilled” by Fowler& Scott

Packages: Visibility Controls

- visibility controls at the package level
  - + public, - private
- the public parts of a package are called its exports

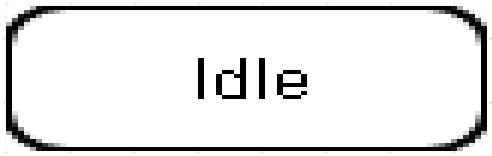


Statechart Diagrams

- a statechart diagram shows the flow of control from state to state
  - for a single object
  - shows how the state of the object changes as a result of events that occur
- elements:
  - states
  - transitions, guarded transitions
  - events
  - activities
- composite states: a single state consists of a state machine

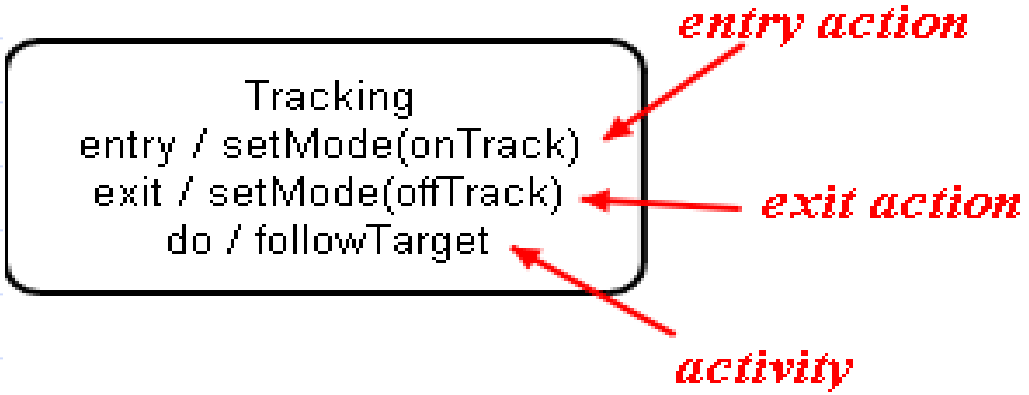
What's in a state?

- a state is a condition or situation in the life on an object during which it satisfies some condition, performs some activity, or waits for some event
  - typically described by a set of attribute values
- examples:
  - the state of a credit card account depends on current balance, payment history, ...
  - the state of an order can be pending, filled, onBackOrder, cancelled, delivered, ...
  - the state of a fax can be Idle, Sending, Receiving
- notation:
  - rectangle with rounded corners
  - usually only name shown



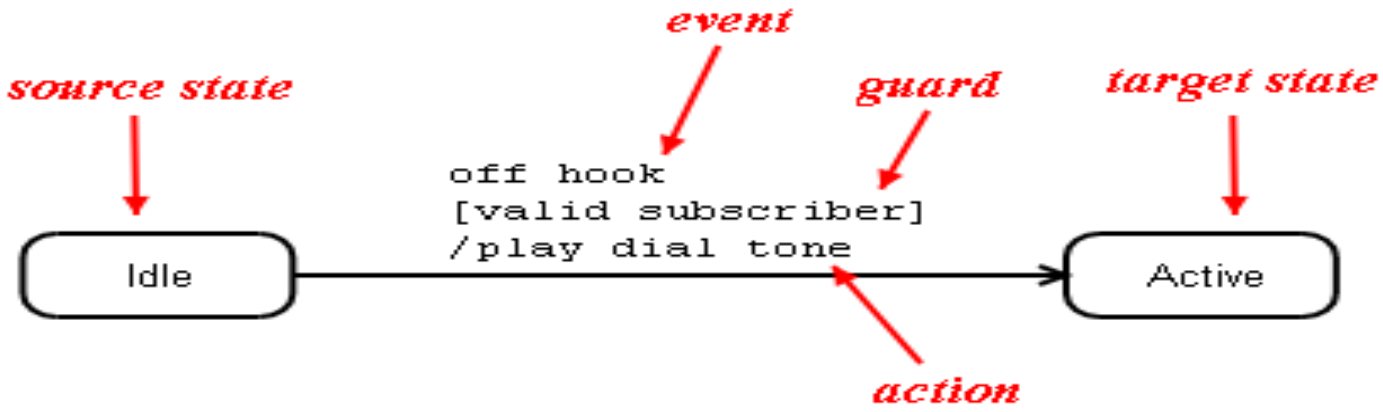
More State Notation

- notation for initial state: ●
- notation for final state: ◉
- states may also have:
  - entry action — entry / ...
  - exit action — exit / ...
  - activity — do / ...
    - activities may take longer, and may be interrupted by events



State transitions

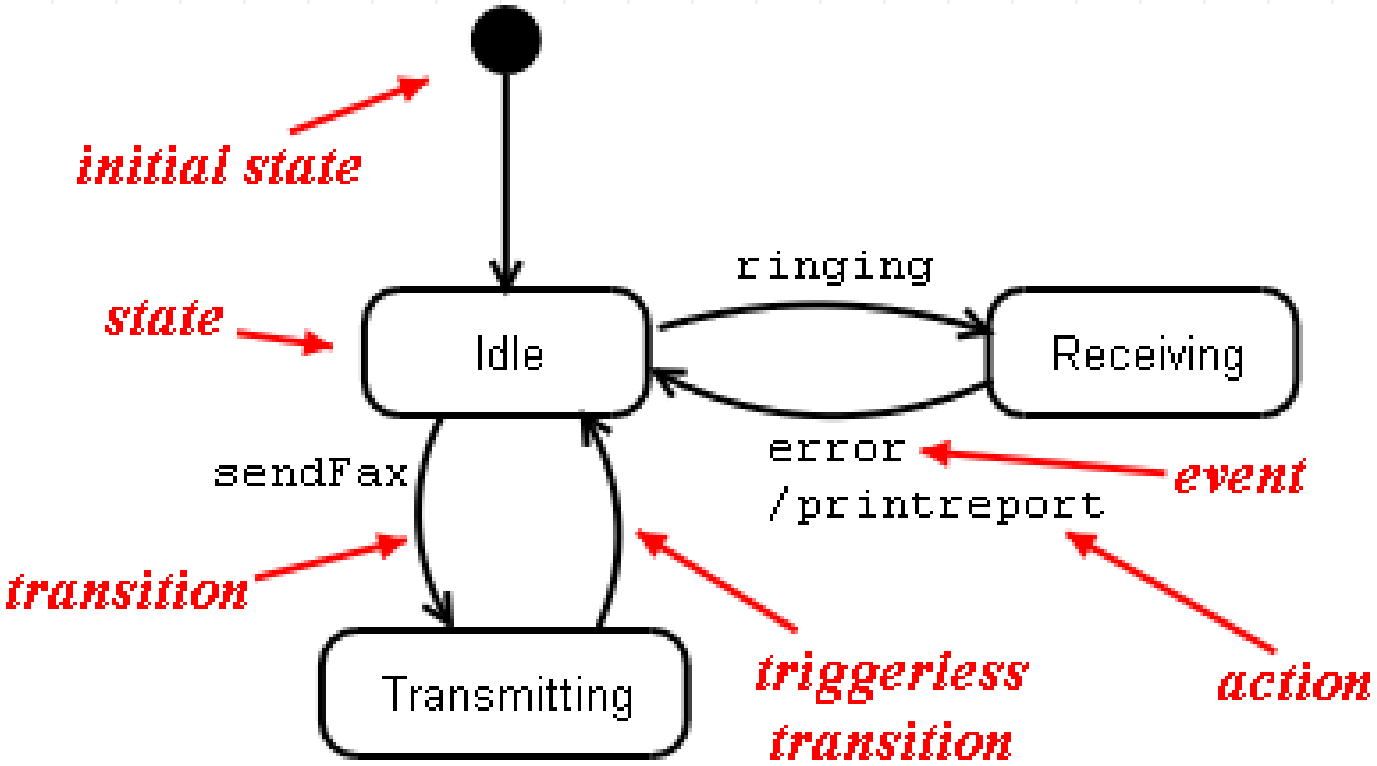
- a state transition occurs as a result of an event
  - a transition connects a source state to a target state
  - transitions are considered to be atomic
- labels of state transitions: *Event [Guard] / Action*
- event: the specification of a significant occurrence
  - a message or signal that is received
- an action is an executable atomic computation
  - a process that occurs quickly and is not interruptible
- a guard is a logical condition



States & State Transitions

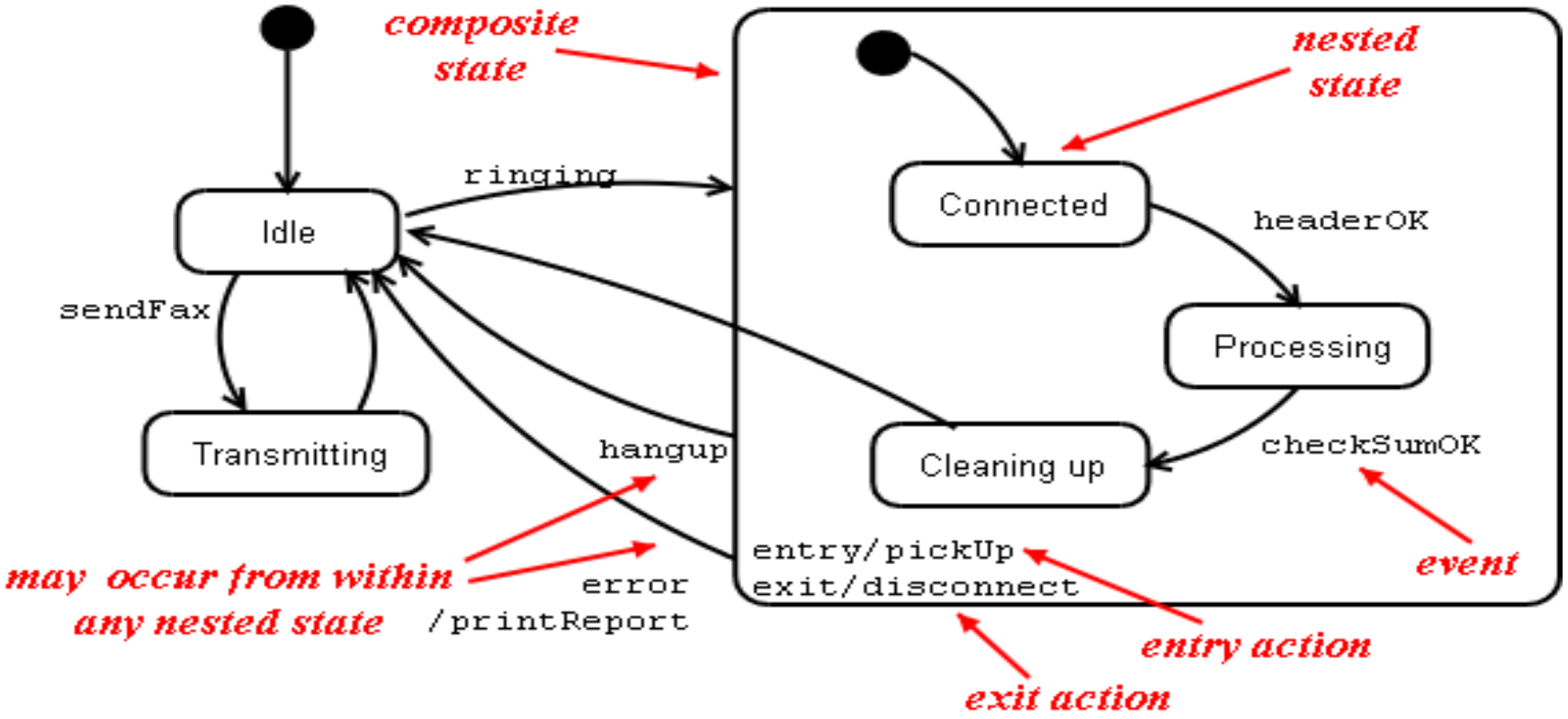
- examples of events
  - credit card payment received (on specific date)
  - payment received for order
  - fax machine receives a call
  - time events: take place at a specific time, or after a specified number of seconds/minutes/hours
    - example: after (2 seconds)
- transitions without an associated event are called a triggerless transition
- events that do not cause state change give rise to a self-transition

Statechart Diagram: Example

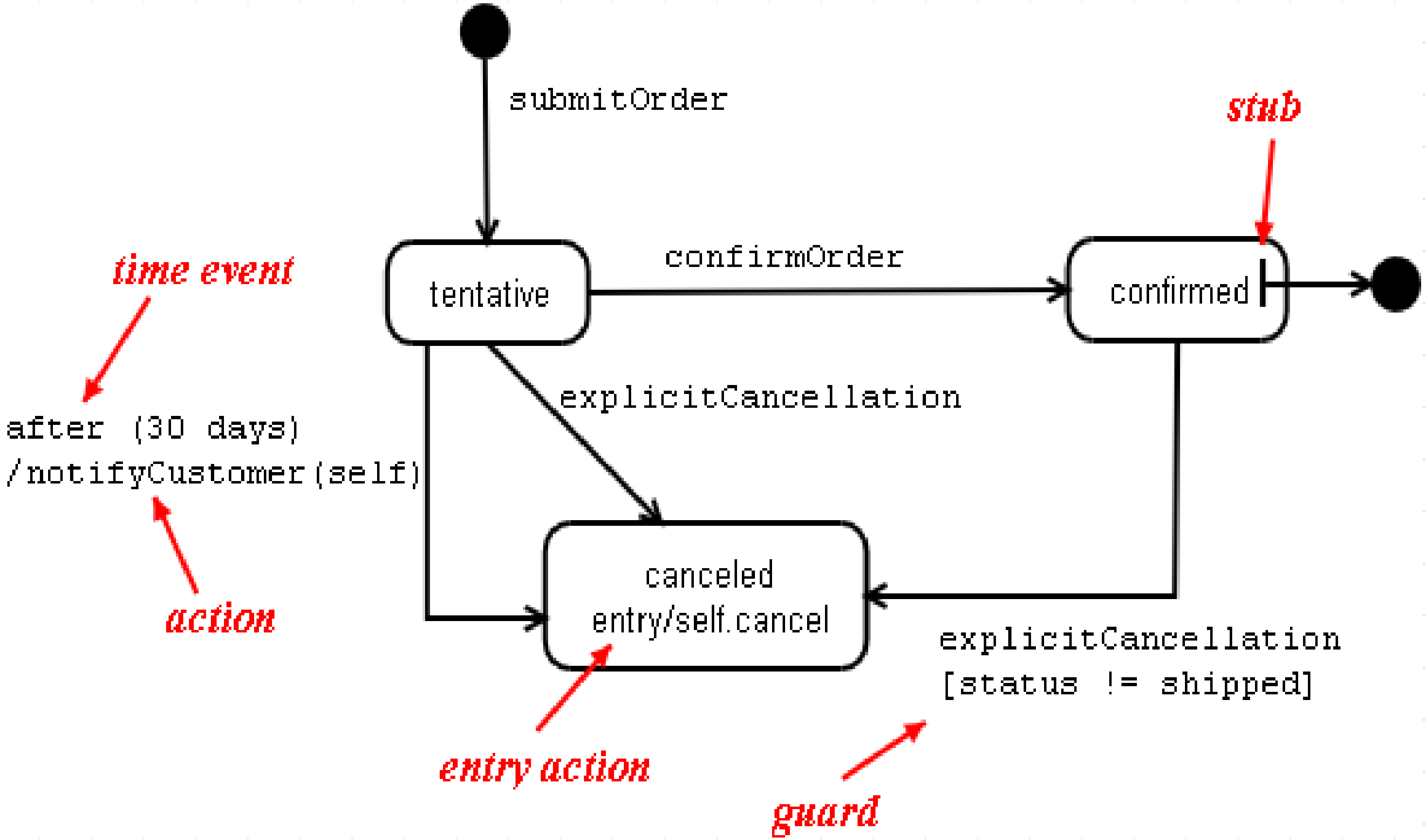


Composite states

- a composite state is a state that is itself a state machine
- convenient when many states have the same outgoing transitions
- reaching end state of composite state triggers transition with composite as source

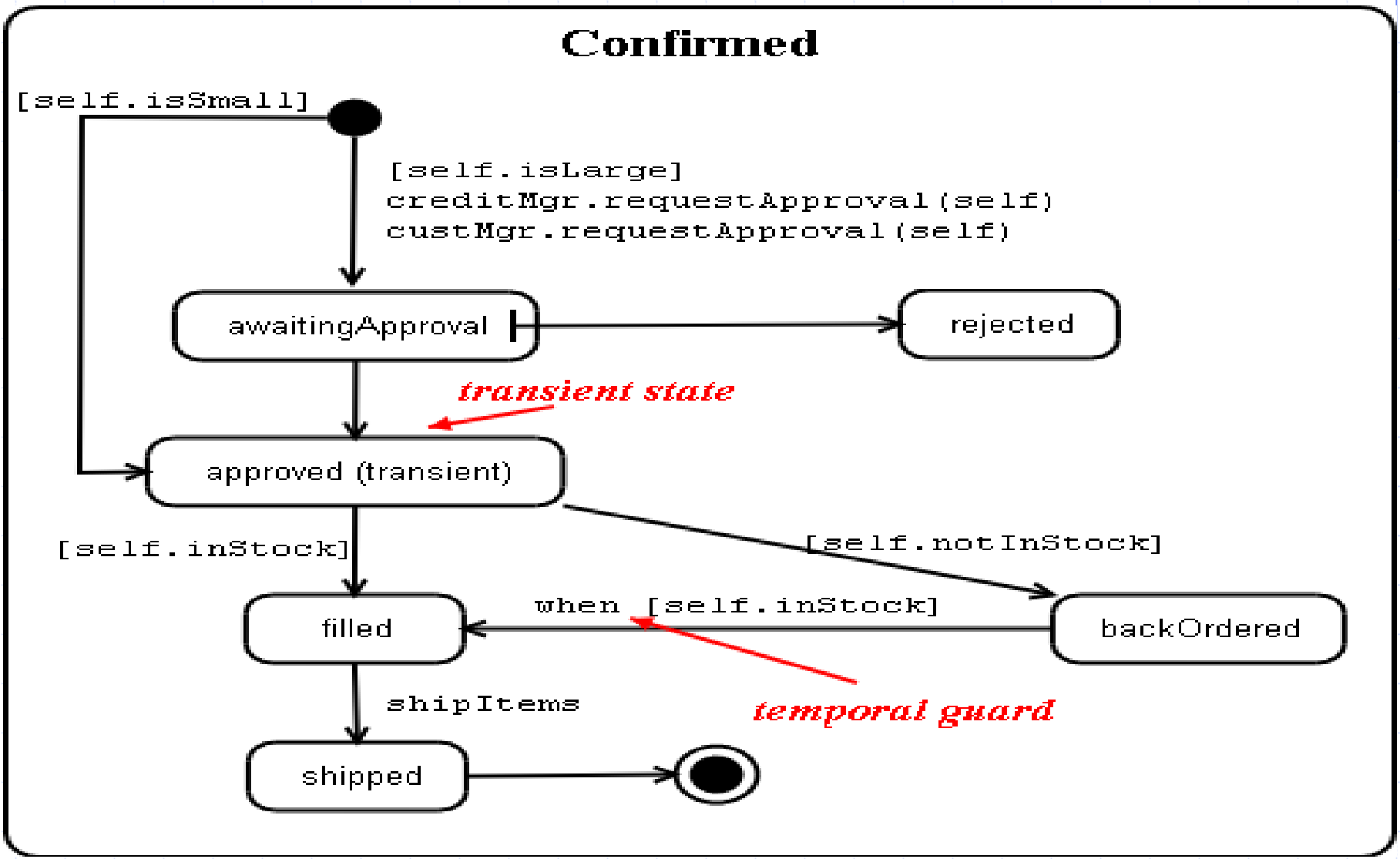


Another Example

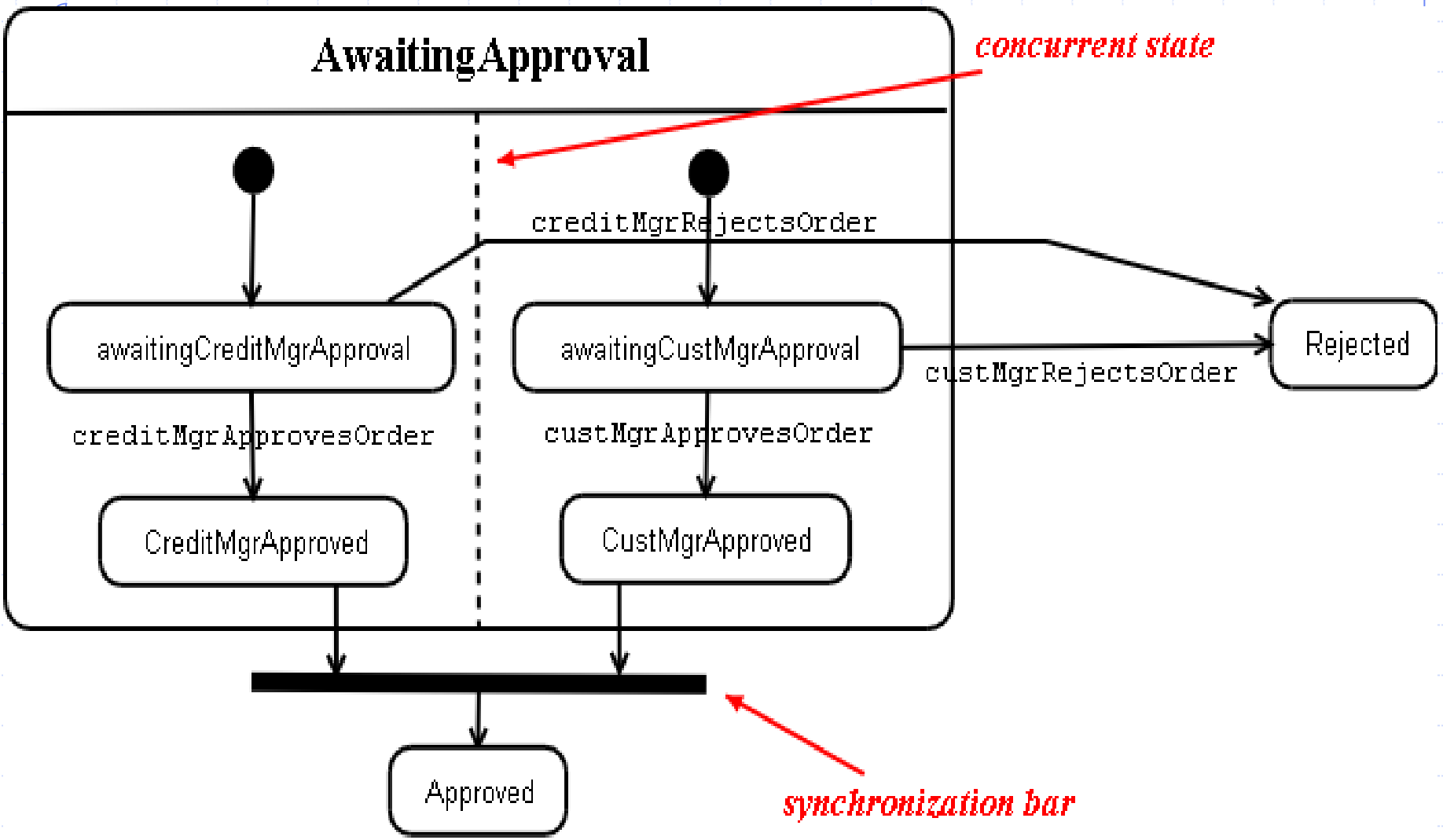


example taken from M. Page-Jones: "The Fundamentals of Object-Oriented Design in UML"

Inside the "Confirmed" State



Example of a Concurrent State



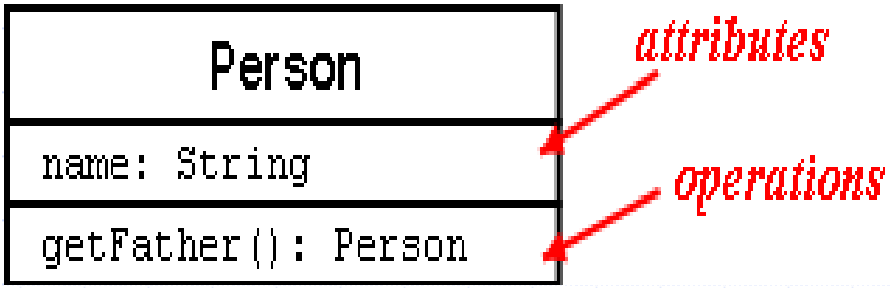
Statechart Diagrams: Guidelines

- keep it simple: if diagrams get too big:
  - consider using composite states
  - or...multiple objects
- trace through the states manually, compare against expected results
- check that all states are reachable under some combination of events
- check that no nonfinal state is a dead end

Classes in UML

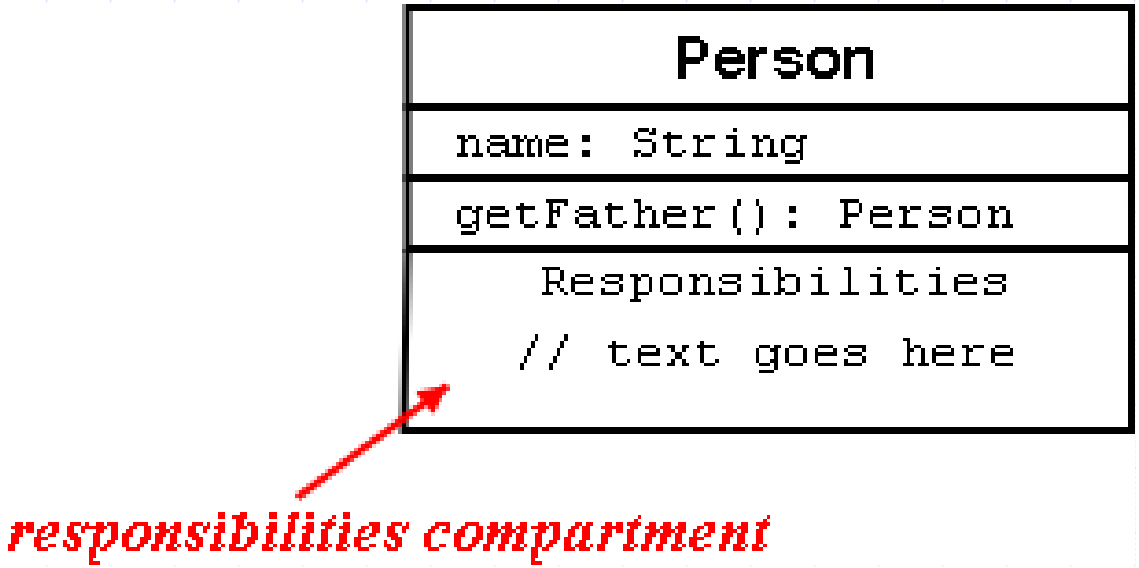
- A class is represented by a rectangle, with a name inside it

- or as a rectangle with 3 compartments for its name, attributes and operations:



Responsibilities

- classes also have an optional “responsibilities” compartment
  - contains free-form text explaining the responsibilities of this class



Perspectives on Classes

- conceptual (domain analysis)
  - shows concepts of the domain
  - implementation-independent
- specification (design)
  - general structure of the system
  - interfaces (types, not classes)
  - used in high-level design
- implementation (programming)
  - structure of the implementation
  - most often used

Always try to draw from a single perspective!

Attributes

[visibility] name [multiplicity] [: type] [= defaultValue] [{properties}]

- a class may have zero or more attributes
- an attribute is an abstraction of part of the state of an object. It represents a property shared by all objects of the class.
- a range of values may be specified for an attribute
- at any given moment, an object of a class will have a specific value for each of its attributes
- an initial value may be specified for an attribute
- the visibility of an attribute may be specified
- an attribute property must hold at all times for instances of the attribute:
  - predefined properties (e.g., “frozen”)
  - user-defined properties

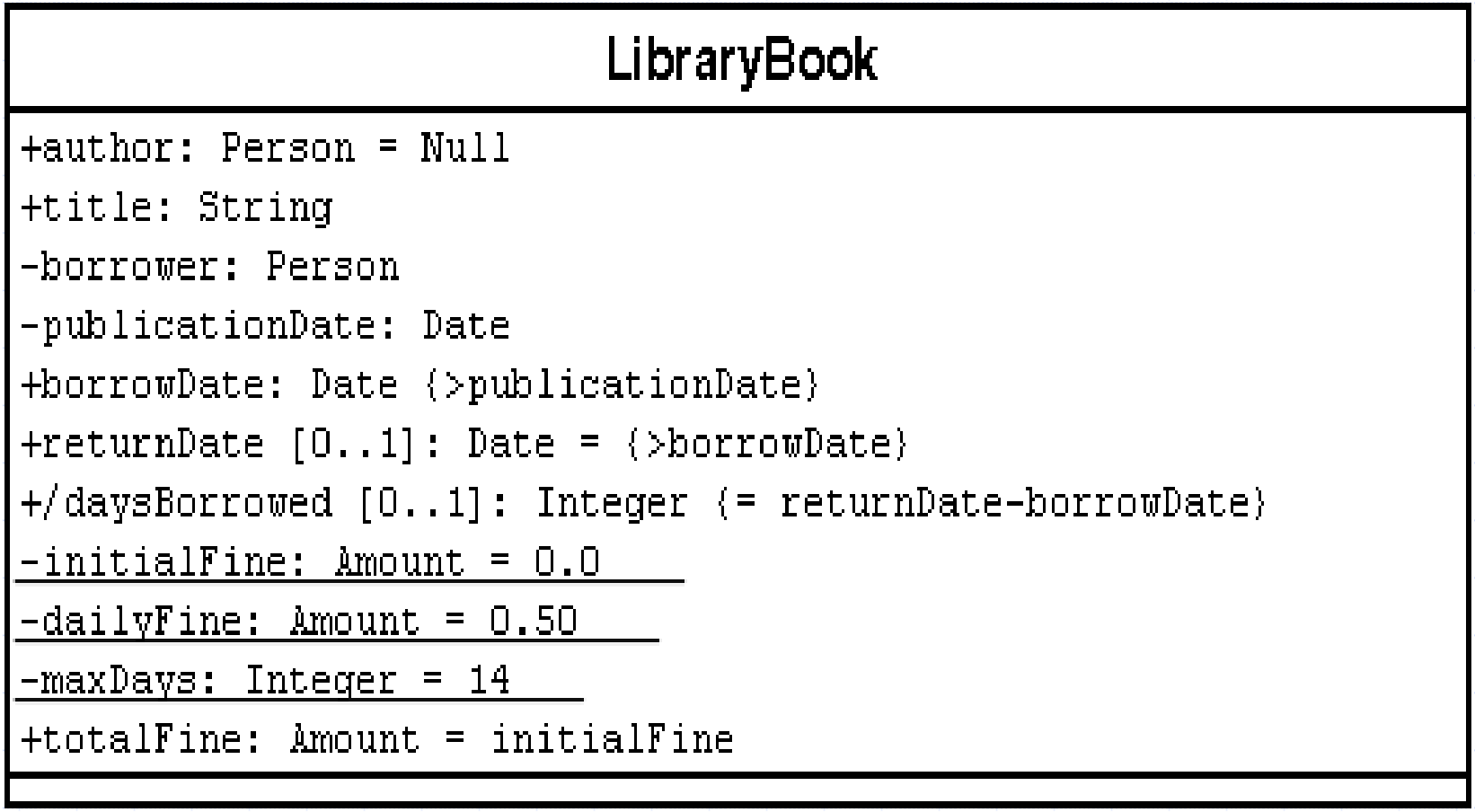
Attributes: Notation

- Each attribute has:
  - name
  - visibility: +,-, #
  - multiplicity: [low..high]
  - type e.g., Integer
  - initial value = Value
  - property string e.g, {frozen}

optional

- Scope of attributes
  - instance scope (one per object)
  - class scope (one per class, ~ Java's static field)
- Derived attributes: value derived from other attributes

Example: Attributes



Corresponding Java Code

```
public class LibraryBook {

  // implement attributes
  public Person author = null;
  public String title;
  private Person _borrower;
  private Date _publicationDate;
  private Date _borrowDate;
  private Date _returnDate;

  // note: daysBorrowed not implemented as a field,
  // because it is derived information, namely
  // computed from (returnDate – borrowDate)

  private static double initialFine = 0.0;
  private static double dailyFine = 0.50;
  private static int _maxDays = 14;
  private double totalFine = initialFine;
```

Implied Get/Set Methods

```
// get/set methods for private fields

  public Person getBorrower(){ return _borrower; }
  public void setBorrower(Person p){ _borrower = p; }

  public Date getPublicationDate(){
    return _publicationDate;
  }
  public void setPublicationDate(Date d){
    _publicationDate = d;
  }

  public Date getBorrowDate(){ return _borrowDate; }
  public void setBorrowDate(Date d){ _borrowDate = d; }

  public Date getReturnDate(){ return _returnDate; }
  public void setReturnDate(Date d){ _returnDate = d; }
```

Operations

[visibility] name ([arg: argType [= defaultValue]], ...) : returnType(s)

- A class can have zero or more operations.
- An operation is a service that can be requested from any object of its class.
- ...takes zero or more arguments (with associated types).
- Default values for args may be supplied, to be used in the absence of a corresponding actual parameter.
- The return type is the type of the value returned by the operation. UML allows multiple return types.
- Special kinds of operations: queries, modifiers, getters, setters.
- Class operations (underlined) operate on a class (instead of on an instance of a class).
- getters/setters are usually omitted

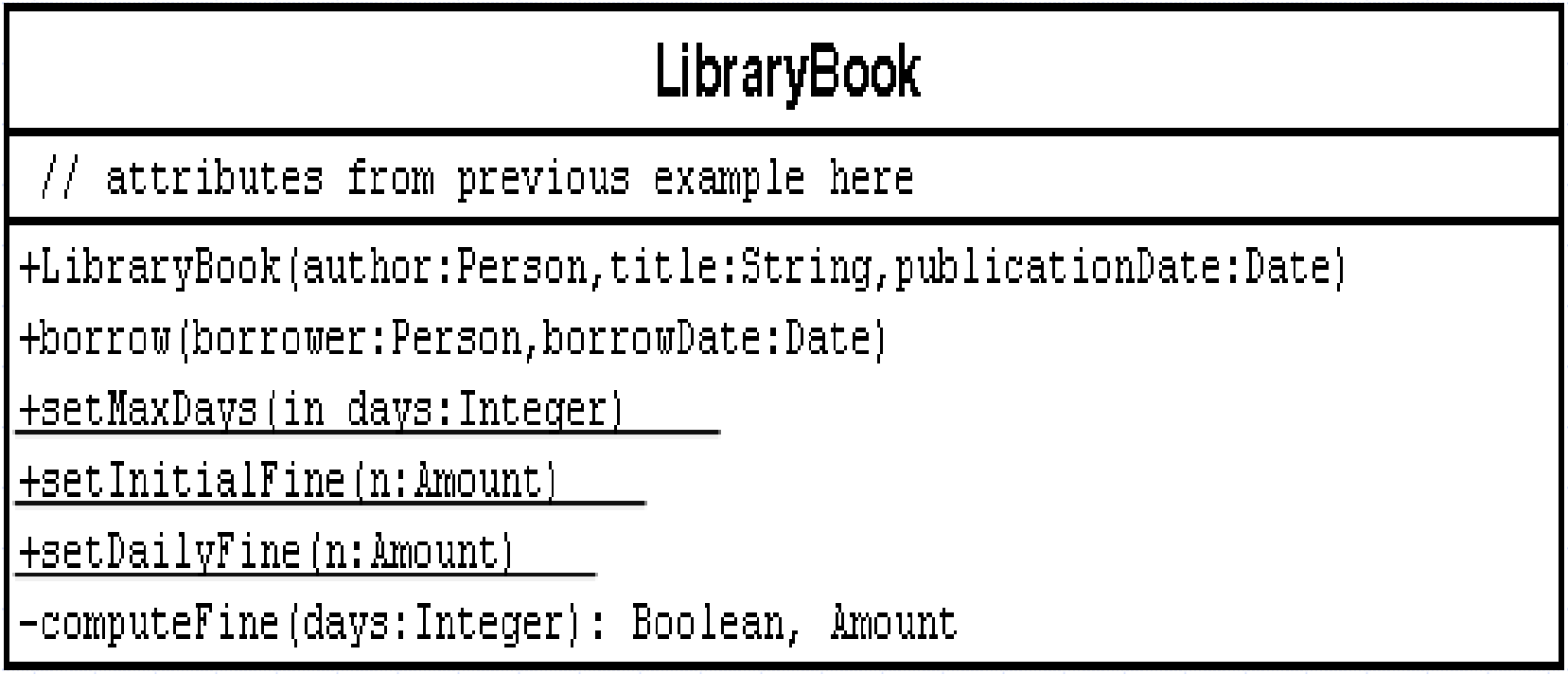
Operations: Notation

- Each operation has:
  - name
  - visibility: +,-, #
  - argument types e.g., name: String
  - initial values for arguments interestRate: Float = 2.5
  - return type(s) e.g, Person,Float
  - in/out/inout parameter: e.g., +setName(in name: String)

optional

- Scope of attributes
  - instance scope (one per object)
  - class scope (operate on a class, ~ Java's static method)

Operations: Example



Java Code for Operations

```
// constructor method
LibraryBook(Person author,
            String title,
            Date publicationDate){
  this.author = author;
  this.title = title;
  setPublicationDate(publicationDate);
}

public void borrow(Person borrower,
                   Date borrowDate){
  setBorrower(borrower);
  setBorrowDate(borrowDate);
}

public static void setMaxDays(int days){
  _maxDays = days;
}
```

Java Code for Operations (2)

```
// implementation of derived attribute
public int daysBorrowed(){
  long borrowTimeInMillis =
    getBorrowDate().getTime();
  long returnTimeInMillis =
    getReturnDate().getTime();
  long difference =
    returnTimeInMillis - borrowTimeInMillis;
  return (int) (difference/(1000*60*60*24));
}

// use two methods to model pair return type
public boolean fineApplicable(){
  return (daysBorrowed() > _maxDays);
}
public double computeFine(){
  return (_maxDays-daysBorrowed())*dailyFine;
}
```

Java Code for Operations (3)

```
// a simple test driver to test the daysBorrowed()
// method
  public static void main(String[] args){
    GregorianCalendar gc1 =
      new GregorianCalendar(1999, 5, 1, 0, 0, 0);
    Person fowler = new Person("Martin Fowler");
    LibraryBook book =
      new LibraryBook(fowler, "UML Distilled",
    gc1.getTime());
    GregorianCalendar borrowDate =
      new GregorianCalendar(2002, 5, 18, 0, 0, 0);
    book.setBorrowDate(borrowDate.getTime());
    GregorianCalendar returnDate =
      new GregorianCalendar(2002, 6, 1, 0, 0, 0);
    book.setReturnDate(returnDate.getTime());
    System.out.println("days = " +
                       book.daysBorrowed());
  }
}
```

A picture is worth a 1000 words...

- Note that a single, small UML class translates into several pages of Java code!
  - graphical notation is much more concise than actual code (mostly by omitting straightforward implementation details)
  - UML is useful for quickly communicating high-level designs between developers (especially if developers are not located at the same site)
- general recommendations:
  - focus on specifying the important operations
  - avoid cluttering diagrams with get/set methods, default constructors, especially if you run out of space.

Class Diagrams

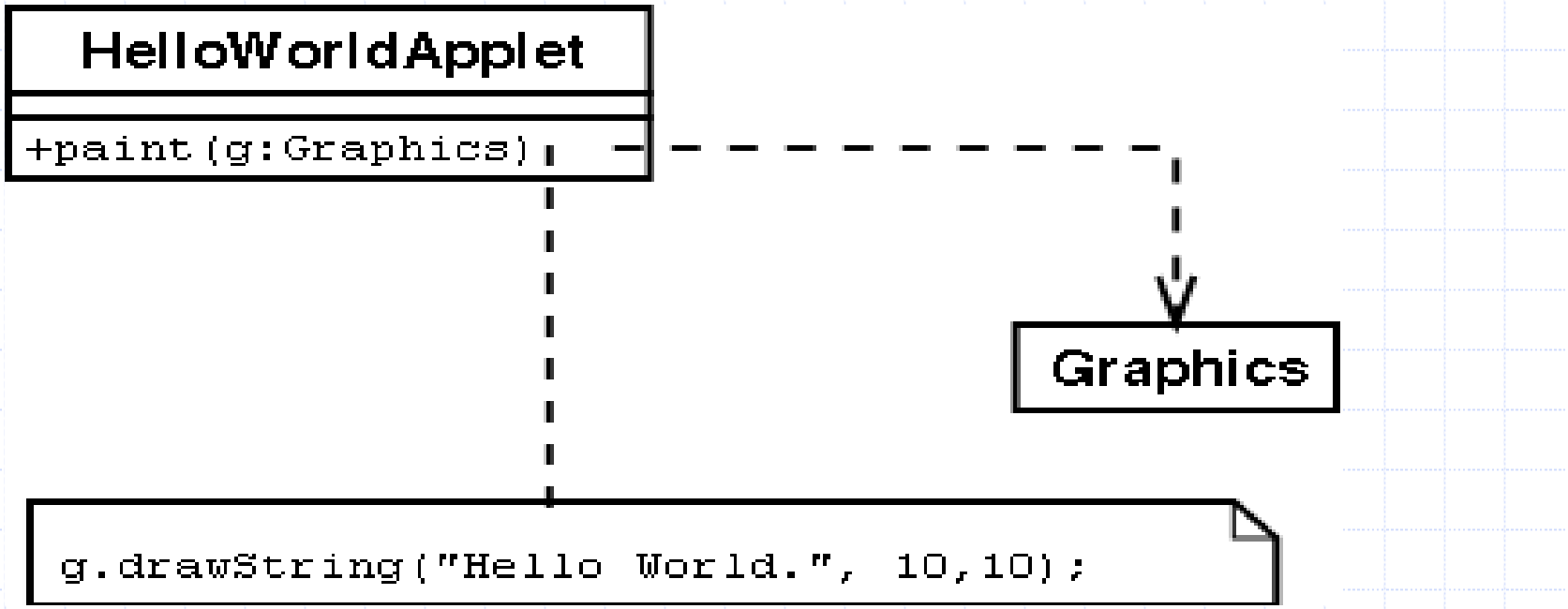
- standard notation for drawing classes and their relationships
- different perspectives:
  - conceptual
  - specification
  - implementation
- types of relationships between classes:
  - dependencies
  - associations
  - subtypes

Overview of UML Relationships

- dependency "A uses B"
  - a change in the specification of B may affect A
  - dashed arrow, with optional name
- generalization "B inherits from A"
  - B is a general thing, A is a more specific thing
  - solid directed line, large open arrow head
- association "B is part of A"
  - structural relationships
  - solid line, with optional name, direction indicator

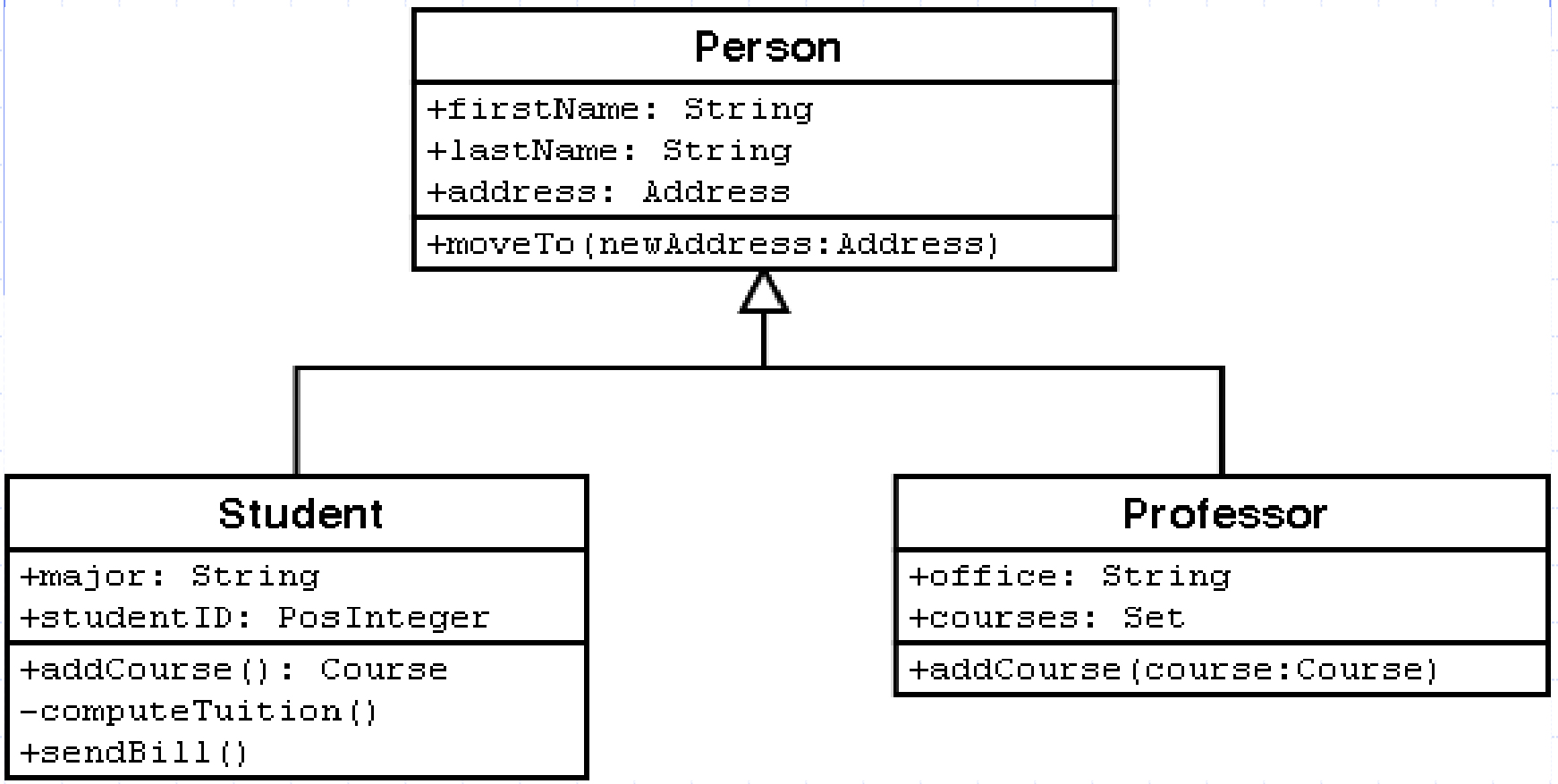
Dependencies

"a using relationship that states that a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse"



Generalization

- also called: inheritance, specialization



Associations

- Structural relationship: objects of one type are associated with objects of another type
  - solid line
  - label for name of relationship (optional)
  - arrowhead to indicate how to read this label (optional)
  - each class plays a specific role in the association, may be indicated in the diagram below the association line

Summary

- use cases & UML use case diagrams
- “package diagrams”
- UML statechart diagrams
- the bare essentials of UML class diagrams