

---

# Design Exercises

Wolfgang Pree

C. Doppler Laboratory for Software Engineering

University of Linz

---

# Motivation

What you will learn

## Reservation system

Object model definition

Generalization

## Mailing system

Object model definition

Pitfalls in OO design

## Simulation system

Object model definition

---

# Motivation

# What you'll learn

---

Hands-on experience in typical design situations:

- define initial object models
- beat abstract classes into shape
- apply a graphic notation (Booch/OMT)

---

# Reservation system

# Reservation system

---

## Exercise 1

Consider a hotel reservation system. The reservation desk has to answer customer requests such as

- Is a room available for two persons at a specific time period?
- How much does it cost?
- Which room categories can you offer?
- Could you please do a specific reservation?

When a guest checks out, an invoice has to be printed. Internally, the accounting system has to be fed with the corresponding data.

- (a) Design the interface of a class **HotelRoom** according to the rough analysis given above.
- (b) Produce relevant use cases/scenarios of how **HotelRoom** instances interact with other system parts

# Reservation system

---

- (c) Generalize `HotelRoom` and define an abstract class `RentalItem` that factors out commonalities of specific rental items (hotel rooms, cars, motor cycles, skis, etc.)

---

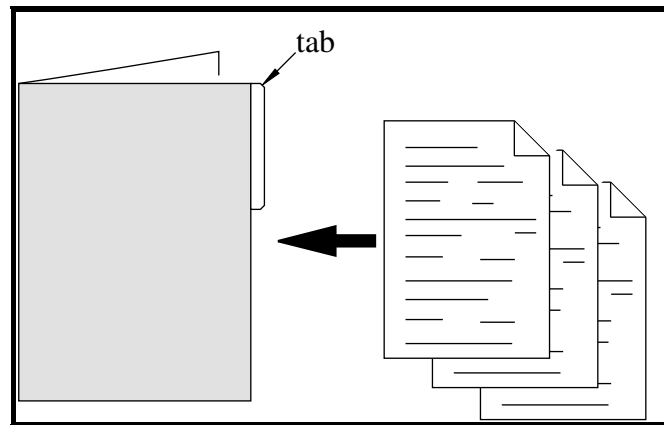
# Mailing system



# Mailing system

---

Basic objects:



Object model:

Folder	0 manages ► *	TextDocument
folderName: String		text: String docName: String
Init(name: String) SetFolderName(name: String) GetFolderName(): String AppendDoc(td: ^TextDocument) RemoveDoc(td: ^TextDocument) GetDoc(name: String): ^TextDocument		SetText(otherText: String) GetText(): String SetDocName(name: String) GetDocName(): String GetTextLength(): Integer

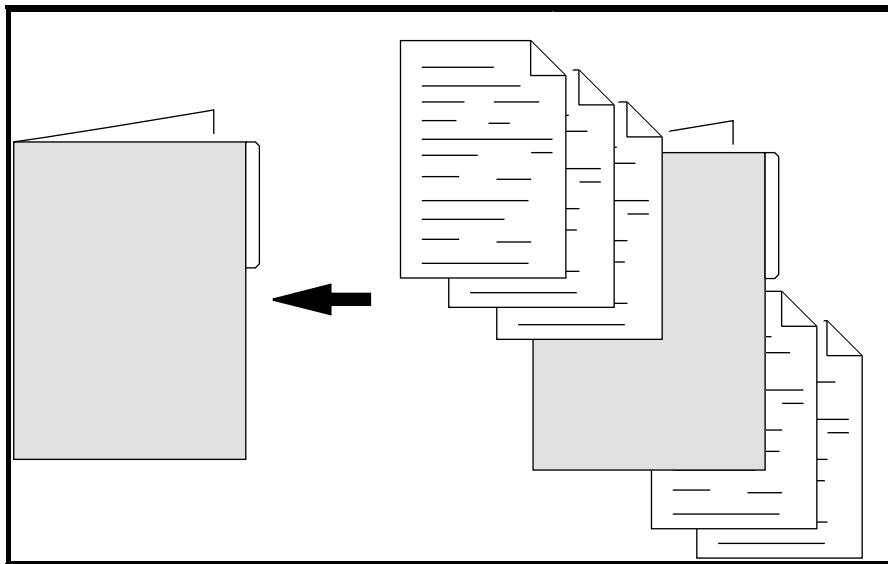
# Mailing system

---

## Exercise 2

Accomplish a local change of **Folder**'s behavior by defining a subclass **NestedFolder** (the interface only). This class should allow

- hierarchical nesting of folders

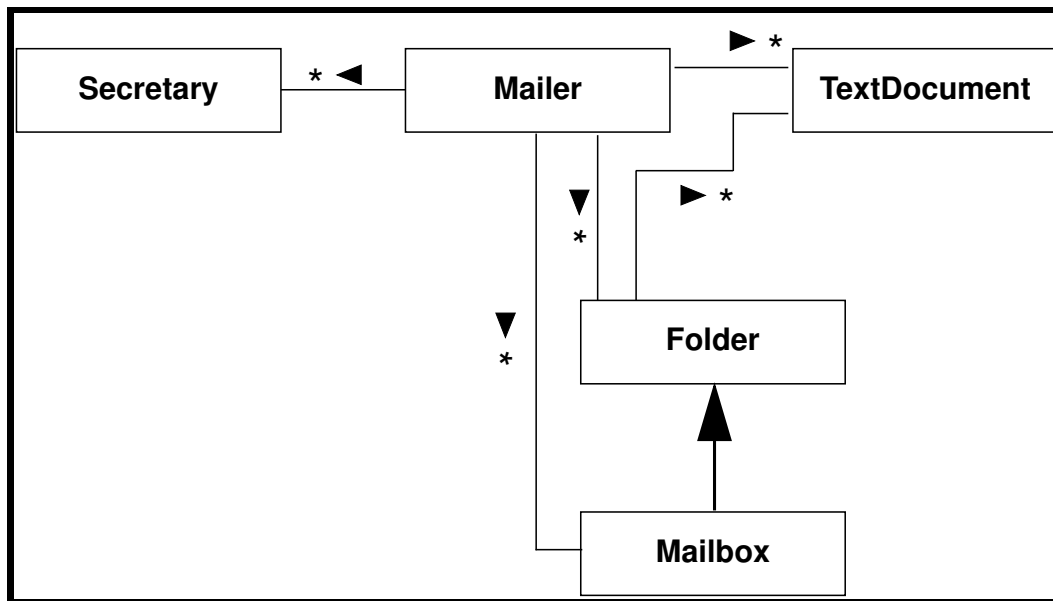


- limitation of the folder name to 30 characters
- calculation of the number of contained items

# Mailing system

---

Problem: you think you are OO,



but additional items, such as

- voice documents
- drawings

require another Folder subclass and changes in the Mailer component!

# Mailing system

---

**=> the system is too rigid and thus not extensible**

# Mailing system

---

**Hot spots** (= places where architecture flexibility is required) in the mailing system:

- items processed by the Mailer component
- employee types

**=> abstract classes have to be introduced**

## Exercise 3

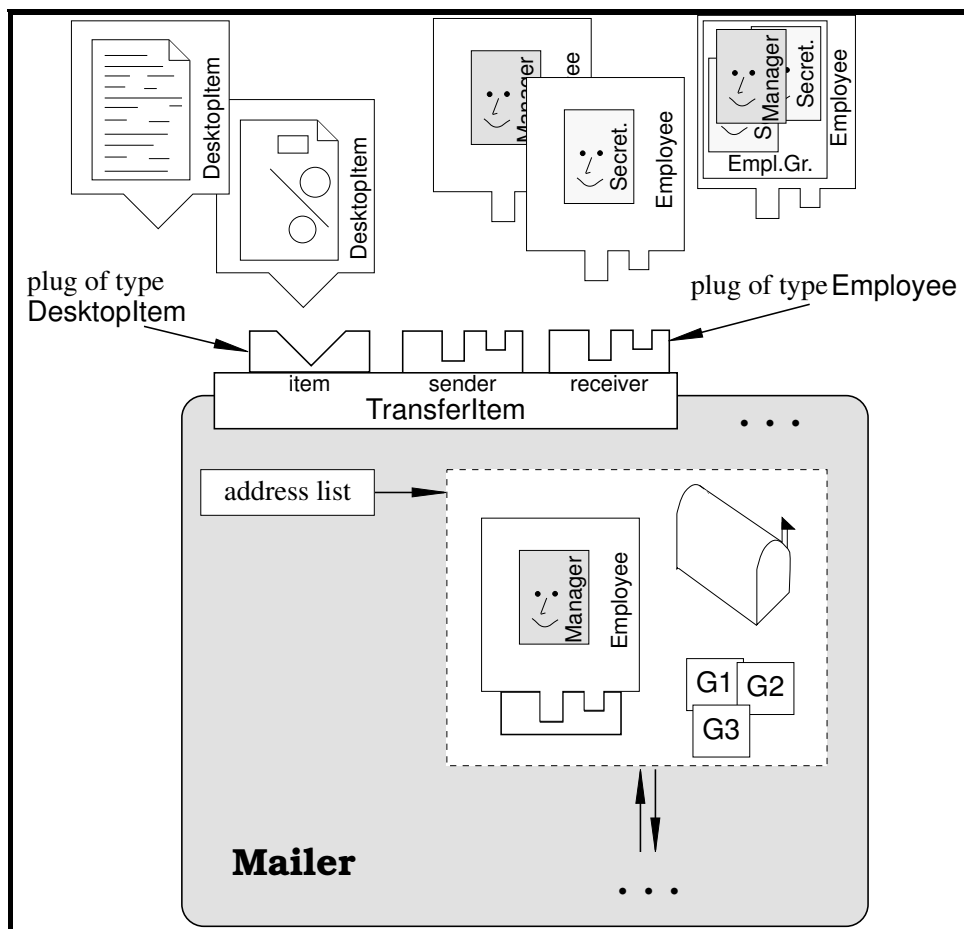
Define an abstract class **DesktopItem** that factors out commonalities of its specific subclasses (**Folder**, **TextDocument**). Additional future subclasses might be **VoiceDocument** and **DrawDocument**.

Discuss the advantages of this design. How does the new design relate to the design of **NestedFolder** in Exercise 2?

# Mailing system

---

Abstract classes can be viewed as plugs that are compatible to specific objects instantiated out of one of their subclasses:



---

# Simulation system

# Simulation system

---

Discrete event simulation is intuitive to understand and can be applied to a variety of problems:

- modeling of waiting lines in stores or banks
- forecast of energy consumption in energy management systems
- network configuration tools
- work flow systems

Events with an associated time represent a central entity in discrete event simulation. Events, such as customer arrivals, are scheduled by a random number generator.

The **simulation progresses from event to event** on the time axis and gathers relevant statistics.



# Simulation system

---

Events are one of the key abstractions in a simulation system.

In their presentation of a simulation framework, Reiser and Wirth call this key abstraction *actor*:

“The main simulation program always pairs an action with an event. ... We call that object an **actor**.”

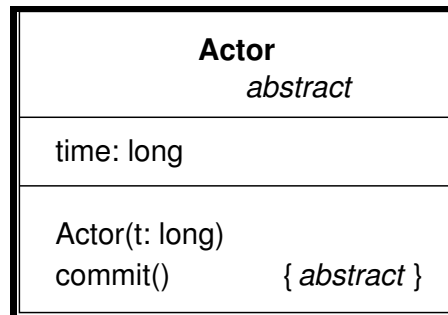
(from Reiser M, Wirth N: *Programming in Oberon—Steps Beyond Pascal and Modula*, Addison-Wesley/ACM Press, 1992)

**Actors know what to do when they receive certain messages.**

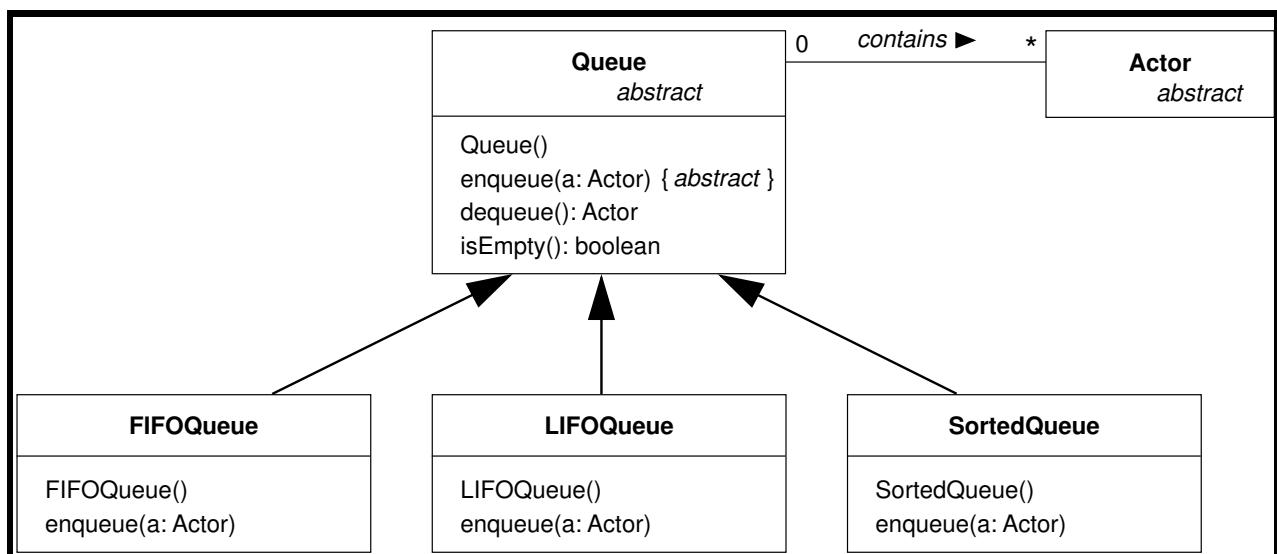
# Simulation system

---

Definition of abstract class Actor:



Utility classes:



# Simulation system

---

Design of the time-dependent notification mechanism:

Simulation		Actor <i>abstract</i>
time: long actors: SortedQueue	0 <i>manages</i> ► *	time: long
Simulation() schedule(a: Actor, time: long) simulate(duration: long) reset()		Actor(t: long) commit() { <i>abstract</i> }

# Simulation system

---

Java implementation of method simulate(...):

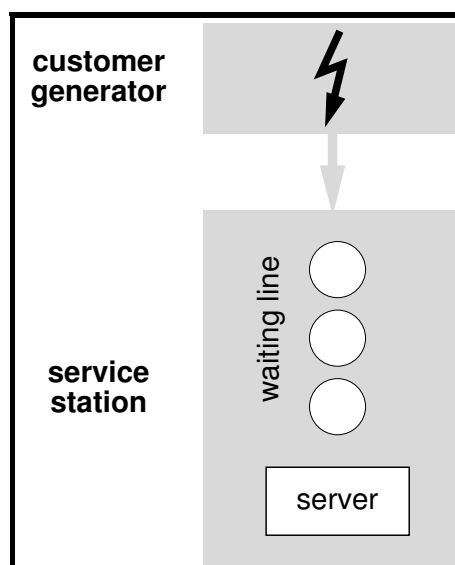
```
public class Simulation {  
    ...  
    public void simulate(long duration) {  
        Actor actor;  
        long endOfSimulation= time + duration;  
  
        do {  
            if (!actors.isEmpty()) {  
                actor= actors.dequeue();  
                time= actor.time;  
                actor.commit();  
            } else    // no more actors enqueued  
                time= endOfSimulation+1;    // exit loop  
        } while (time <= endOfSimulation);  
    }  
    ...  
}
```

# Simulation system

---

## Exercise 4

Extend the basic simulation framework in order to simulate a simple cash desk in a convenience store with a single waiting line:



**Customers** are created randomly by a **customer generator**. A server processes customer requests. If the server is busy or not available, customers queue in front of the server. Both the waiting line and the corresponding server form one unit. We refer to it as **service station**.

Define the corresponding classes and describe their interaction with use cases/scenarios/interaction diagrams.