Design Patterns & Software Architecture
State

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- State

State design pattern

Let’s find a design pattern

Will now present, on the board,
and using Eclipse,
a solution that utilises
the state design pattern…

Case: Movie on demond system Requirements

A software house decides to develop a movie on demand program. The product manager identifies the following requirements:

A software house decides to develop a movie on demand viewing program. The product manager identifies the following requirements:

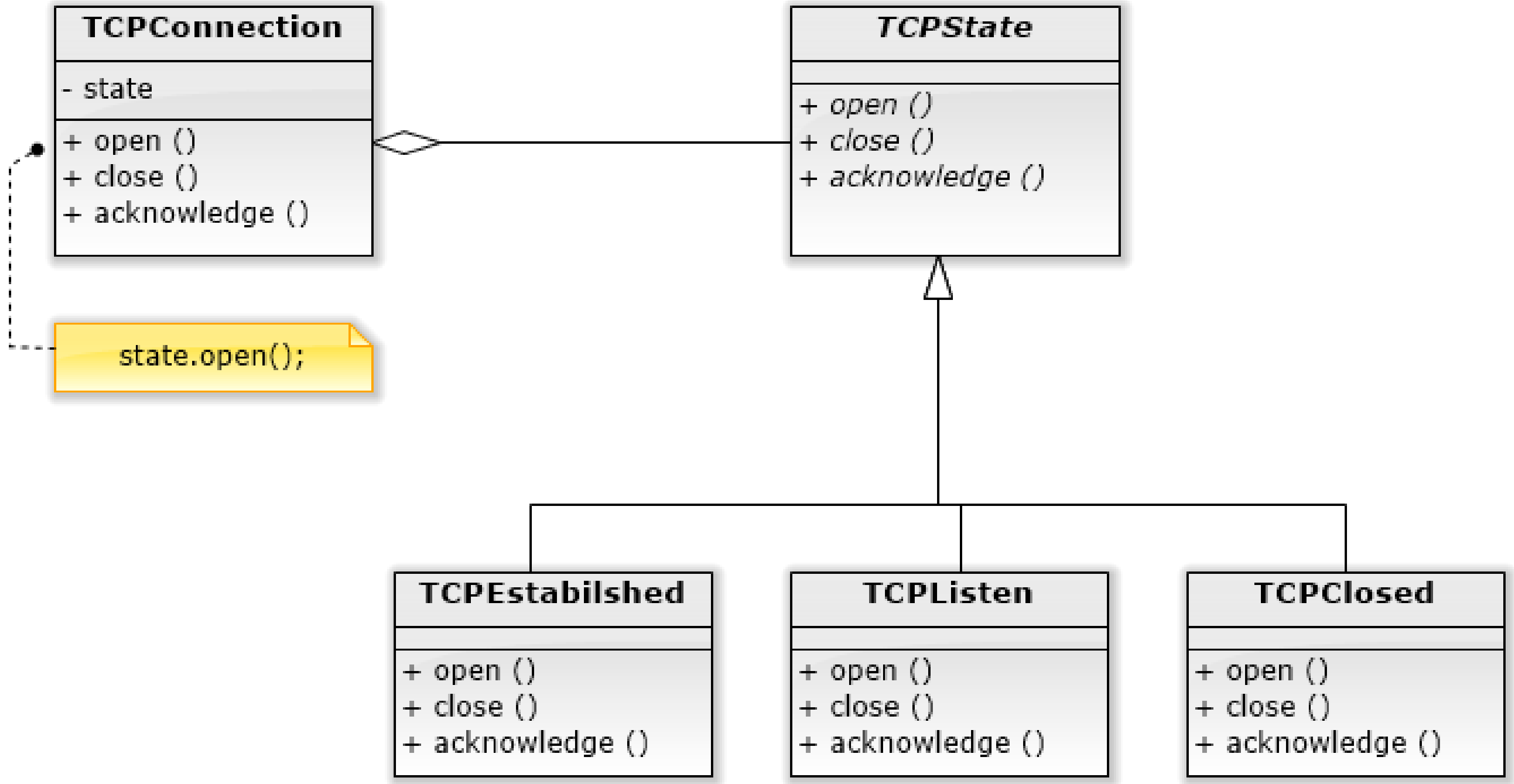
1. Every movie will have a title, one or more actors, a SIN (Standard Identification Number), and a viewing fee (see below).
2. Initially, the system will deal with three categories of movies: Children, Special and Regular.
3. The view charge for each is different:
   - Children: These movies cost € 1.50 for the first view and € 0.50 for each further view.
   - Special: These movies cost € 3 for every view, except on Saturday and Sunday, then the price is € 4 for every view.
   - Regular: These movies cost € 2 for the first view, € 1.50 for the next view and then € 0.50 for each subsequent view.
   - New Release: These movies cost € 5 for every view.
4. The program must handle new categories of movies in the future with minimal change to existing code *and also the possibility of category changing at runtime. Also charging structures will change.*
5. **A movie remains in the New Release state for N days after which it becomes one of the other categories: currently Children, Special, Regular.**

State pattern definition

- **Intent**: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

- **Motivation**: Class TCPConnection represents a network connection. It can be in one of several states: Established, Listening, Closed…and behaves accordingly. So we solve this in way shown on next slide…

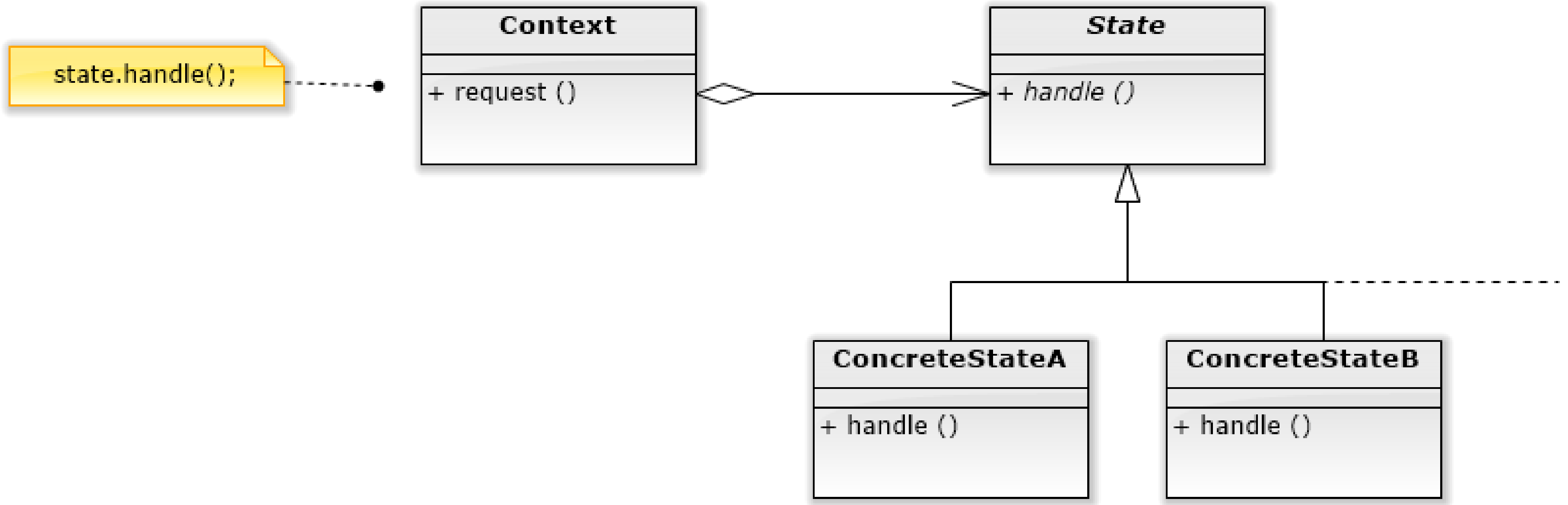
Motivation



TCPConnection
- state
+ open ()
+ close ()
+ acknowledge ()
state.open();
TCPState
+ *open ()*
+ *close ()*
+ *acknowledge ()*
TCPEstabilshed
+ open ()
+ close ()
+ acknowledge ()
TCPListen
+ open ()
+ close ()
+ acknowledge ()
TCPClosed
+ open ()
+ close ()
+ acknowledge ()

- **Applicability:** Use the state pattern in either of the following cases:
  - An object’s behaviour depends on its state, and it must change its behaviour at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object state…

- **Structure:**


state.handle();
Context
+ request ()
State
+ handle ()
ConcreteStateA
+ handle ()
ConcreteStateB
+ handle ()

- **Participants:**
  - Context (TCPConnection)
    - defines the interface of interest to clients.
    - maintains an instance of a ConcreteState subclass that defines the current state.
  - State (TCPState)
    - defines an interface for encapsulating the behavior associated with a particular state of the Context.
  - ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)
    - each subclass implements a behavior associated with a state of the Context.

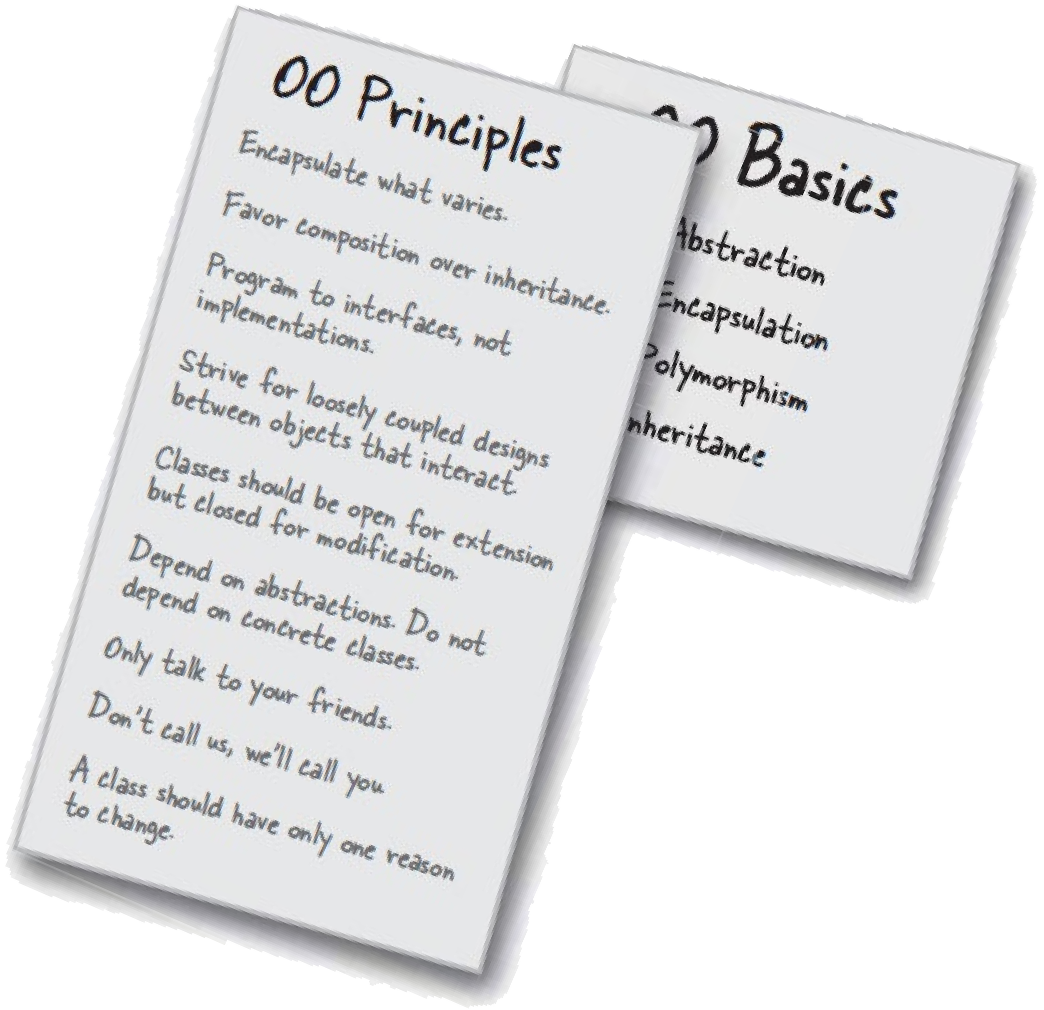
- **Collaborations:**
  - Context delegates state-specific requests to current ConcreteState object.
  - A Context may pass itself as an argument to the State object… letting the State object access the Context if necessary…
  - Context is the primary interface for clients.
  - Clients can configure context with State objects…
  - Either Context or the ConcreteState subclasses can decide on state transitions and the events that trigger them…

- **Consequences**: The state pattern leads to:
  - localization of state-specific behaviour for different states...
  - explicit state transitions explicit...
  - shareable State objects...
  - tightly coupling of concrete sub-classes of State.

- **Implementation**: Some issues
  - Who defines the state transitions?...
  - Use of hash tables…
  - Creating and destroying state objects…

Principles dealt with by this pattern

- Favour composition over inheritance
- Q. *What other principles are covered?*


OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

Only talk to your friends.

Don't call us, we'll call you

A class should have only one reason to change.

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

Reading

For this lesson please read:

- Chapter 10 (State of Things) of Head First Design Patterns.