State Pattern

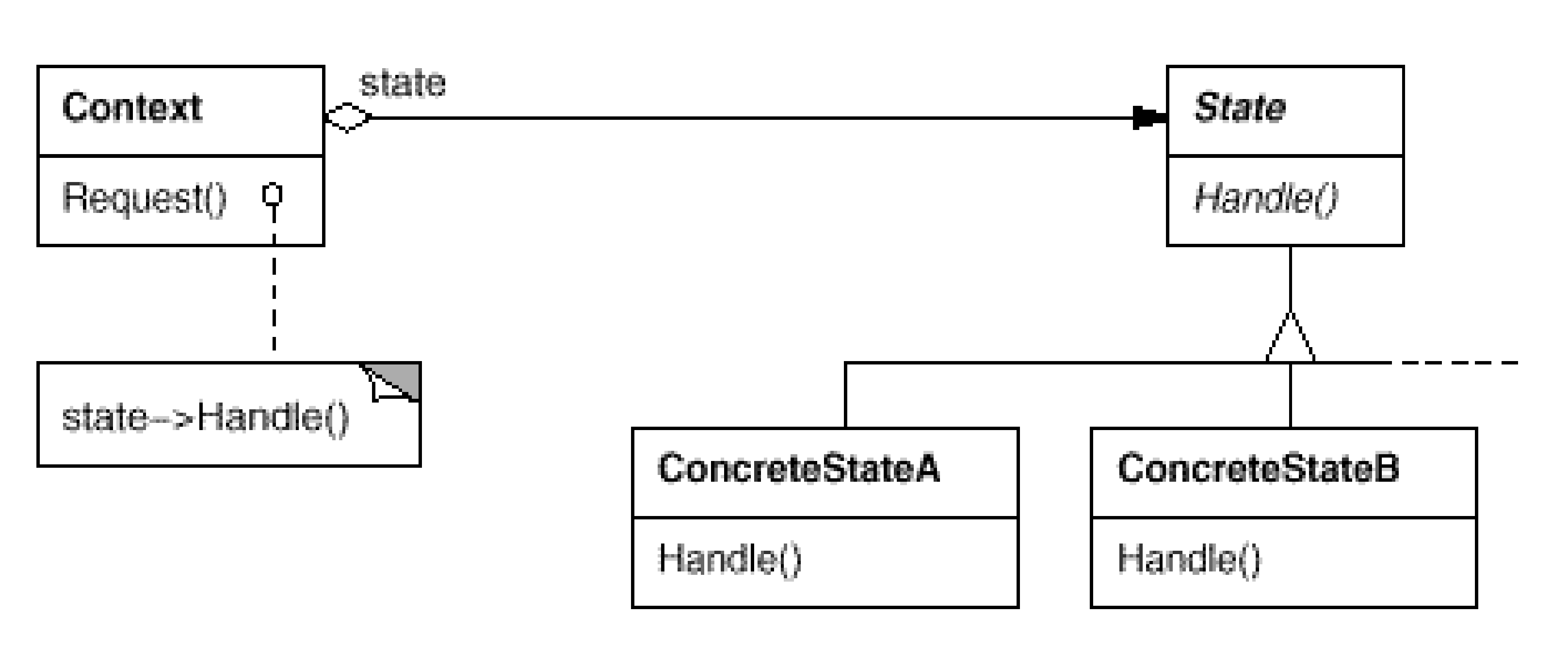
Chapter Content

- State Pattern Overview
- State Pattern UML Diagram
- State Examples
- State machine - Advantages
- State machine - Warnings
- Exercise

State Pattern Overview

- A class encapsulates a *state* object representing its current state (may be one of several possible states).
- Class behavior depends on its current state:
  - Method calls are forwarded to the state object, which decides how to handle them.
  - Some would say it’s the closest thing to having an object change its type at runtime.
- Some would call it Proxy on steroids …

State Pattern UML Diagram



Context - object holds a current state (which may be one of several possible states).

Method calls are forwarded to that state (each state may have a different *Handle()* method).

State-machine for chat client:

drawScreen(): welcome, please login

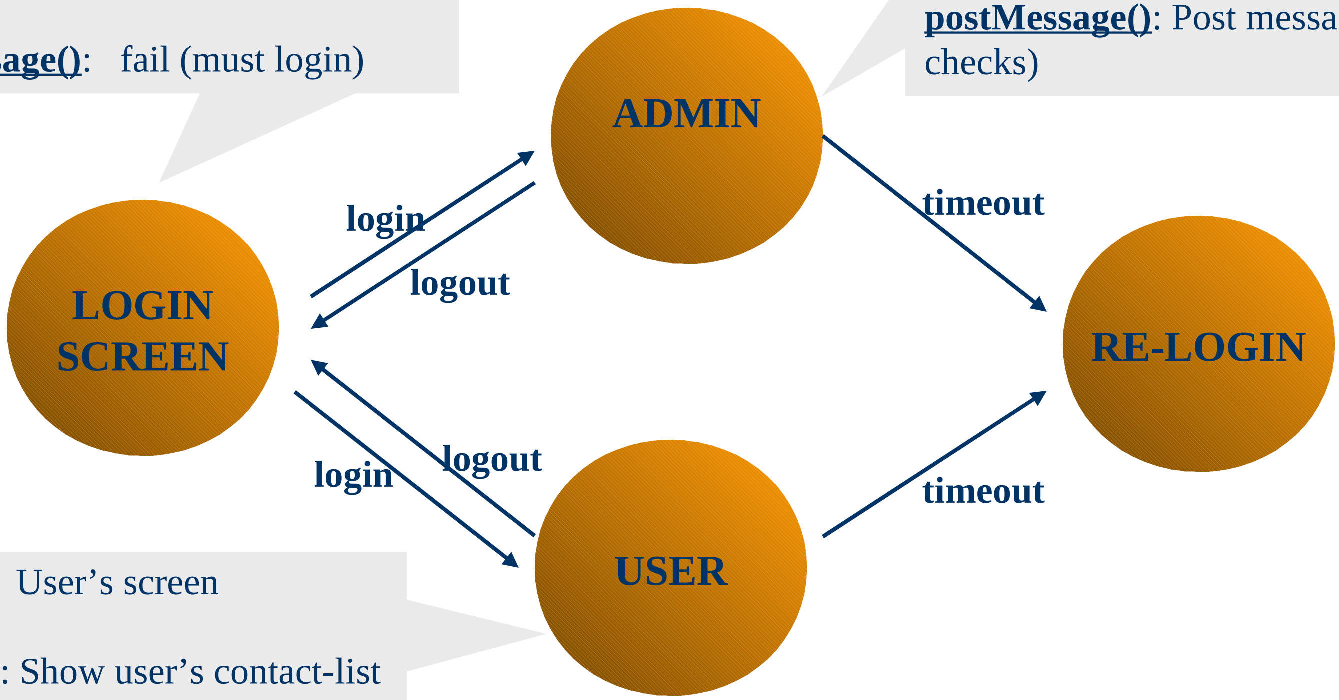
whoIsOnline(): show users who wish to make their details publicly-known

postMessage(): fail (must login)

drawScreen(): Admin screen

whoIsOnline(): Show all users

postMessage(): Post message (no checks)



drawScreen(): User’s screen

whoIsOnline(): Show user’s contact-list

postMessage(): Post message. Possibly run it through a “bad-language filter”

Chat Client (cont.)

```
Abstract class State {
    int code;               // Various common variables…
    protected ChatMachine chat; // State-machine containing this state
    abstract  void  drawScreen();
    abstract  List  whoIsOnline();
    abstract  void  postMessage(String msg);
}

class UserState extends State {
    void drawScreen()   {
        ... // draw user screen
    }
    List whoIsOnline()  {
        ... // show user’s contact-list
    }
    void postMessage(String msg)  {
        ... // post message, after running through filter
    }
}
class AdminState extends State ...
```

Chat Client (cont.)

```
public class ChatMachine {
    private State currentState;
    private State[] possibleStates = {new LoginState(this),
        new UserState(this), new AdminState(this), new TimeoutState(this) };

    public void drawScreen()   {
     currentState.drawScreen();
    }
    public List whoIsOnline()  {
     return currentState.whoIsOnline();
    }
    public void postMessage(String msg)  {
         currentState.postMessage(msg);
    }
    protected void setCurrentState(int stateCode){
     ...  // possibly allow previous state to do clean-up
     currentState= possibleStates[stateCode];
    }
    // Now, how shall we control transition between states …?
}
```

Who controls transition?

- Several approaches as to where to define the logic controlling transition between states:
  - The state controls the transition logic, indicating which state should be next.
  - Some other class (e.g. the state-machine) holds the transition logic.
  - In simple cases: tables.

State transition #1: decision by state

```
// States themselves tell the machine when to go next:

class UserState extends State {
    ...
    void logoutRequested(){
     chatMachine.setCurrentState( ChatMachine.LOGIN_STATE_CODE);  // =0
    }

}
class LoginState extends State {
    ...
    void loginRequested(String username, String pswd) {
     User user = loadUser(username, pswd);
         if (! user.isValid())
         chatMachine.setCurrentState(ChatMachine.LOGIN_STATE_CODE);
     else {
         if (user.isAdmin())
             chatMachine.setCurrentState(ChatMachine.ADMIN_STATE_CODE);
         else
             chatMachine.setCurrentState(ChatMachine.USER_STATE_CODE);
     }
    }

}
```

State transition #2: by machine

```
class UserState extends State {
    ...
    void logoutRequested(){
     chatMachine.setCurrentState(chatMachine.getNextState());
    }
}


public class ChatMachine {

    private State currentState;

    ...

    int getNextState(){

     switch(currentState.getCode()) {

         case LOGIN_STATE_CODE:

              User user = loadUser(getInputUserName(), getInputPswd());

          if (!user.isValid())    return LOGIN_STATE_CODE:

          else {
              if (user.isAdmin()) return ADMIN_STATE_CODE);
              else                return USER_STATE_CODE;
         ...
```

State transition #3: table

- Generalization: encode state-transition rules in some general data structure (table or some other graph representation).
- Required some careful consideration:
  - How to describe conditional decisions.
  - Can data be serialized into a txt/xml file ?

- Aims for a general multi-purpose state machine. May be more difficult to develop.

State machine advantages:

- May be easy to maintain & trace.
- Easy to extend with new states
  - Consider how one can **combine states** (E.g. a sprite that is shooting-while-jumping).
- Code is distributed between states, thus avoiding huge“switch” clauses.
  - Except maybe for the state transitions.
- Changing the state changes the machine’s behavior – like changing type at runtime.

State machine - warnings

- Beware of code duplication, when several states have similar behaviors.
- More allocations (you don’t only allocate the class, but also its states).
- Communication between states !
  - You’re likely to need some common context for storing shared data (e.g. User details are loaded by Login-State, but they’re required by following states as well).