Software Engineering

Lesson Design Pattern 06
State, Singleton
v1.0a

Uwe Gühl

Fall 2007/ 2008

Contents

- State
- Singleton

Used sources:

- [GHJ04] Gamma, Helm, Johnson, Vlissides: Design Pattern, Addison Wesley, 2004
- [Hus08] Vince Huston: Design Pattern, www.vincehuston.org/dp/, 2008

State

- Intent:
  - Allow an object to change its behavior when its internal state changes
  - The object will appear to change its class
  - An object-oriented state machine
  - To use a collaborating wrapper with polymorph technique
  - ... is a Behavioral Pattern

State

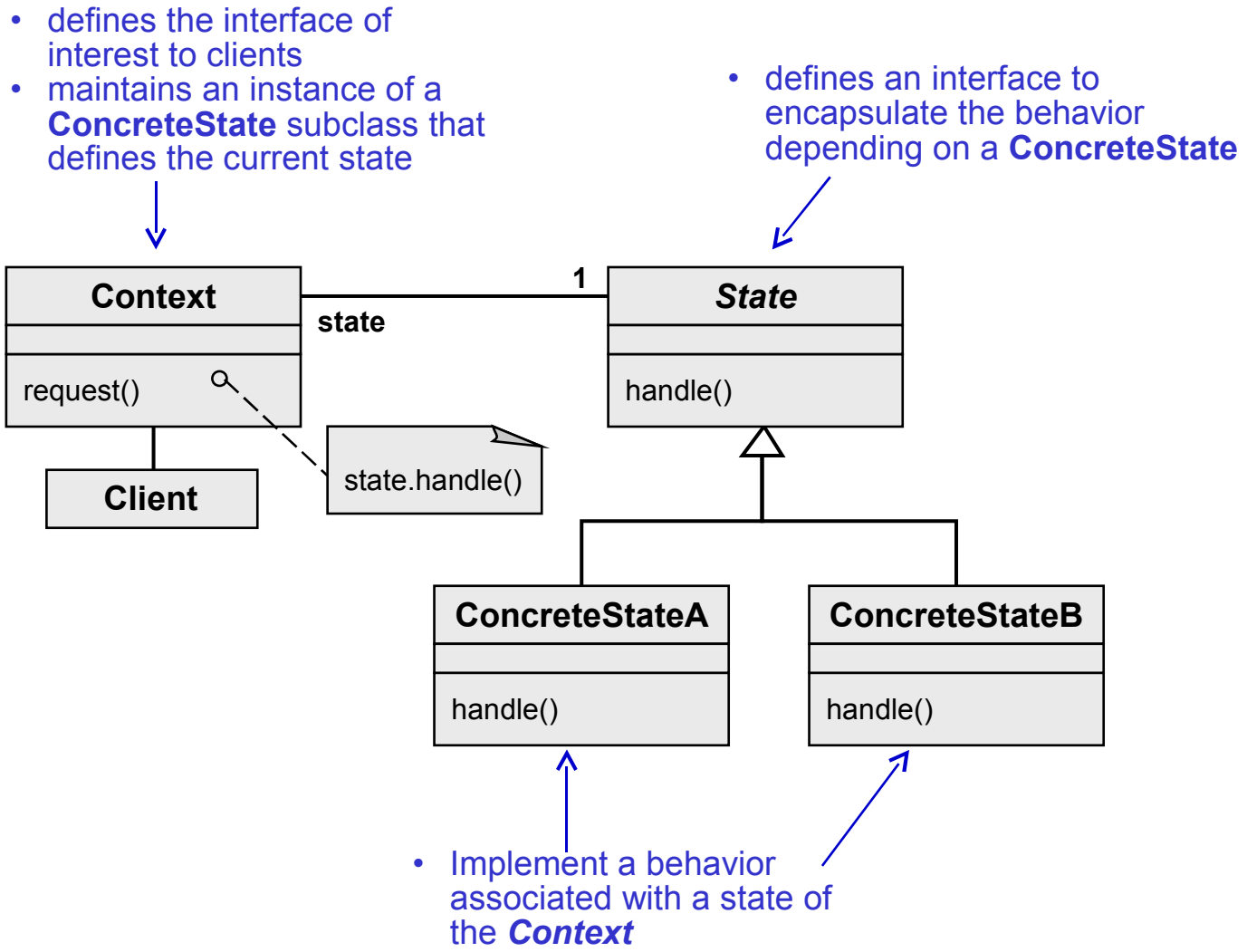
- Motivation
  - If an object has different behavior in different states, modelling of the behavior with if / else and case statements could get complex quickly
  - The program code should be clearly arranged and easy to maintain

State

- Ideas
  - Introduction of an abstract class representing different states
  - This abstract class defines a common interface for the concrete subclasses
  - Each subclass implements the behavior of a specific state

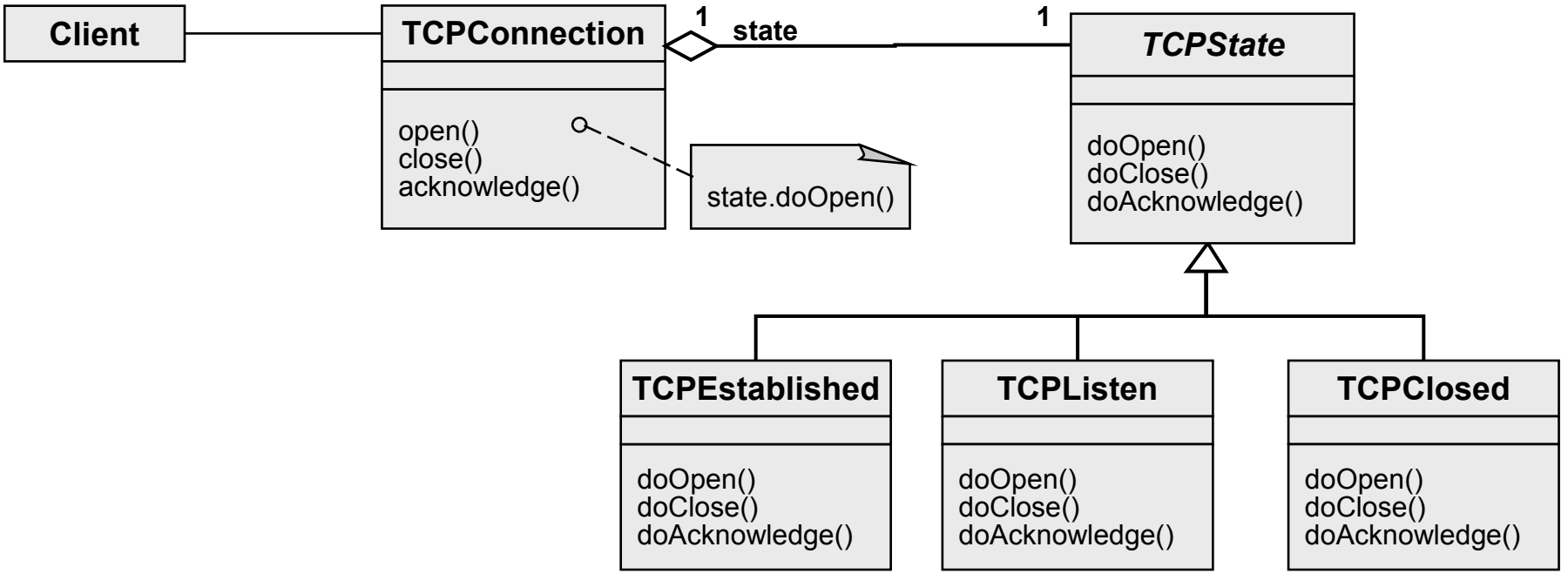
State

- Structure



State

- Example
  Management of states of a network connection



State

- Collaboration
  - **Context** delegates state-specific messages to the current **ConcreteState** object
  - **Context** could pass itself to the State object so that the State object could communicate with it if necessary
  - After configuration of a **Context** with State objects the client interacts with **Context** only
  - Succeeding States – could be decided either by **Context** or the **ConcreteState** subclasses

State

- Applicability
  Use the State Pattern if
  - an object's behavior depends on its state, and it must change at run-time depending on that state
  - operations have large conditional statements depending on an object's state

State

- Consequences

+ State depended behavior gets partitioned and localized (→ Polymorphy)
  - New states could be added easily with new subclasses
  - As this pattern distributes behavior for different states to subclasses, the number of classes in the system increases instead of large conditional statements in one class

State

- Consequences

+ State transitions get modeled explicitly
  - Transitions between the states are explicit
  - Protection of inconsistent internal states
+ It's possible to use state objects in common – if they don't have instance variables
  - Shared in this way the State objects are just Flyweights

State

- Implementation
  - Who defines the state transitions?
    Context or **State**?
    - Example:
      ```
      // every action returns the next state
      state = state.doOpen();
      ```
  - Alternative Implementation with tables to map inputs to state transitions

State

- Implementation
  - Lifecycle of state objects – two ideas:
    - Create State objects only when needed and destroy them afterwards
      - If states to be entered are not known at runtime and contexts change state often
      - Avoids creating not needed state objects
    - Creating them ahead of time and never destroy them
      - If state changes occur rapidly
      - If it is okay that instantiation costs are paid once in the beginning and that Context must keep references to all states

State

- Known Uses (see [GHJ+95])
  - TCP connection protocols [JZ91]
  - HotDraw Framework [Joh92]
  - UniDraw Framework

State

- Related Patterns [Hus08]
  - State, Strategy, and Bridge have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58]
    - State and Bridge use the same structure to solve different problems [Coplien, C++ Report, May 95, p58]
      - State allows an object's behavior to change along with its state
      - Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
    - The difference between State and Strategy is the intent
      - Strategy: The choice of algorithm is fairly stable.
      - State: A change in the state of the "context" object causes it to select from its "palette" of Strategy objects. [Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 1999, p253]

State

- Related Patterns
  - Flyweight
    - The Flyweight pattern explains when and how State objects could be shared
  - Interpreter
    - Interpreter can use State to define parsing contexts.
  - Singleton
    - State objects are often Singleton

Singleton

- Intent:
  - A Singleton is the combination of two essential properties:
    - Ensure a class only has one instance
    - Provide a global point of access to it
  - ... is a Creational Pattern

Singleton

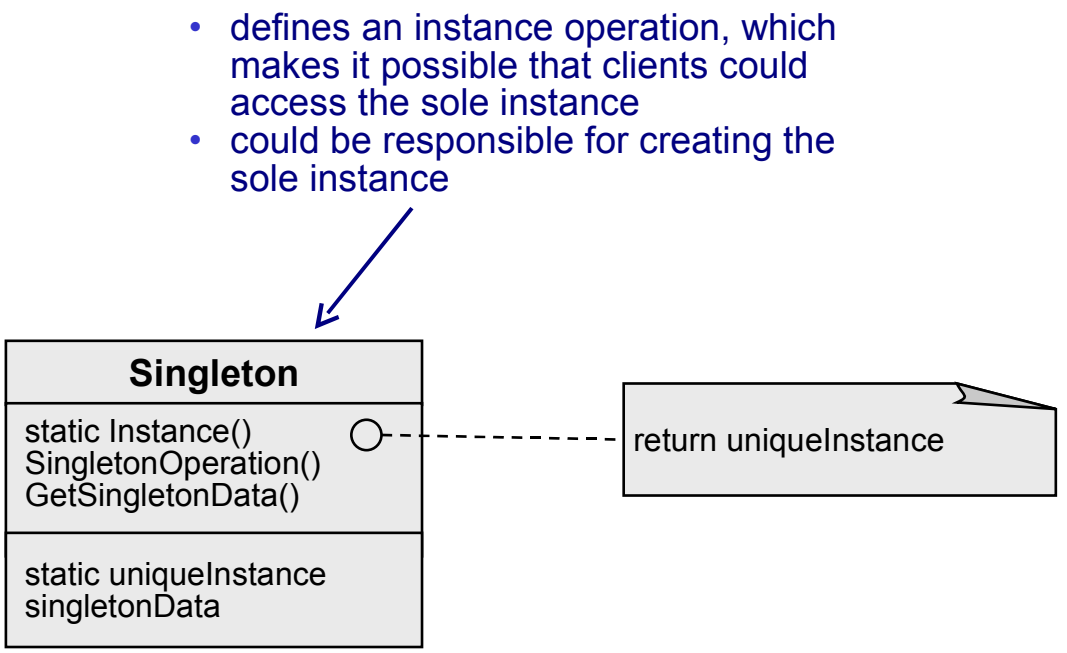
- Motivation
  - For some classes its important that they have exactly one instances
  - Example: For many printers should be only one spooler available

Singleton

- Applicability
  - Exactly one instance of a class must be available and this instance must be accessible by a client from a well known access point
  - The sole instance should be extensible by subclassing, and clients should be able to use an extended instance without changing their code

Singleton

- Structure



Singleton

- Collaboration
  - Clients access an **Singleton** only through an instance operation provided by it

Singleton

- Consequences

+ Controlled access to the only instance
+ The Singleton controls who uses when an instance
+ The global namespace gets not extended
  - Too many global variables in a system make it complex – the Singleton pattern offers an alternative to global variables
  - Singletons permit lazy initialization, where global variables typically consume always resources

Singleton

- Consequences

+ Subclassing allows to refine operations and representations
+ Realization of a variable number of instances possible
+ More flexibility than with class operations
  - The functionality of a Singleton could be achieved with other techniques like static functions in C++ or class methods in Smalltalk
  - But it's more difficult then to allow more than one instance of a class

Singleton

- Consequences
  - Inflexibility
    - The property “a class only has one instance” often has not real value - and it reduces flexibility in the system, e. g. if a second object of a “singleton” is available

Singleton

- Implementation

```
public class Singleton {

    // Private Class attribute, created with first use of class
    private static Singleton instance;

    // Constructor is private, may not be instantiated from external
    private Singleton() {}

    // Static method "getInstance()" returns the only instance
    // of the class. Lazy initialization.
    // Because it's synchronized it is safe for threads.
    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Each created instance will be same

Singleton

- Implementation

```
public class Singleton {

    // Private Class attribute, created with first use of class
    private static Singleton instance;

    // Constructor is private, may not be instantiated from external
    private Singleton() {}

    // Static method "getInstance()" returns the only instance
    // of the class. Lazy initialization.
    // Because it's synchronized it is safe for threads.
    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
            return instance;
        }
        return null;
    }
}
```

How to do exception handling if the singleton already exists? Here getInstance() returns null, what the client could test

Singleton

- Implementation

```
//fails at compile time because constructor is privatized
mySingleton = new Singleton();
```

Singleton

- Implementation
  - Here's a very simple implementation of a singleton Foo object*:

```
Foo globalFoo;  // Don't create any
                // other instances!!!
```

* Source: http://c2.com/cgi/wiki?SingletonPattern

Singleton

- Known Uses
  - In Smalltalk the relationship between classes and metaclasses are designed with Singletons
  - Math class is something like a Singleton class in the standard Java class libraries
    - is declared final
    - all methods are declared static, meaning that the class could not be extended.
    - Goal is to wrap a number of common mathematical functions such as sin and log in a class-like structure, because Java does not support functions that are not methods in a class.

Singleton

- Related Patterns
  - Many patterns can be implemented using the Singleton pattern like
    - Prototype
    - Abstract Factory
    - Builder
  - Facade objects could be Singletons if the Facade object should be unique
  - State objects are often Singletons