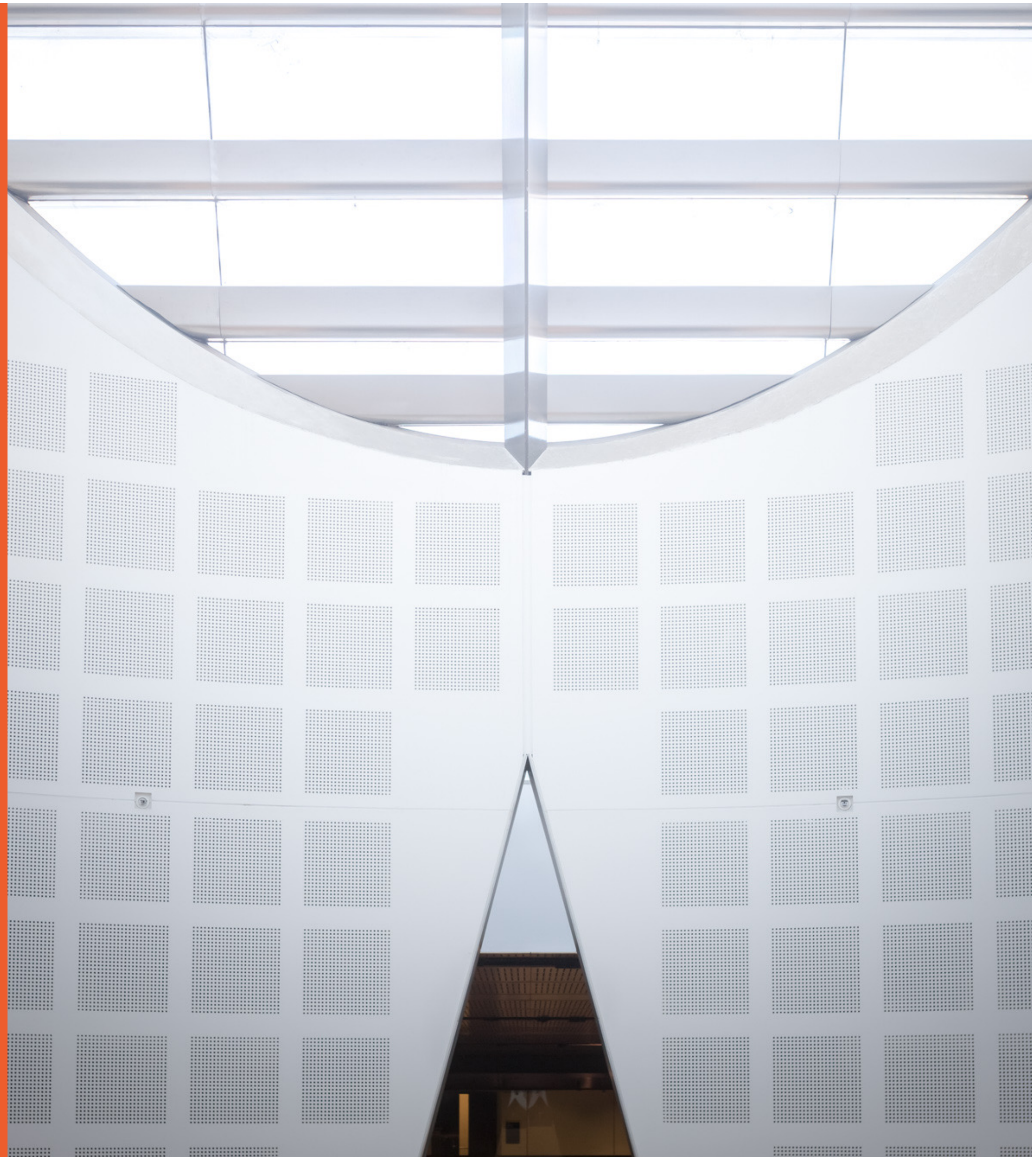
Software Design and Construction 1
SOFT2201 / COMP9201

Behavioural Design Patterns

Dr. Xi Wu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

Announcement

- Topics in the following weeks

Week	Contents		Week
7	Design Pattern: Adapter & Observer	Code Review	8
10	Design Pattern: Prototype & Memento	Testing	11
12	Design Pattern: Singleton, Decorator and Facade		
13	Unit Review		

- Slides with demo example will be updated to Canvas after each lecture

Agenda

- Behavioral Design Patterns
  - Strategy
  - State

Behavioural Design Patterns

THE UNIVERSITY OF SYDNEY

Behavioural Patterns

- Concerned with algorithms and the assignment of responsibilities between objects
- Describe patterns of objects and class, and communication between them
- Simplify complex control flow that's difficult to follow at run-time
  - Concentrate on the ways objects are interconnected
- **Behavioural Class Patterns (SOFT3202)**
  - Use inheritance to distribute behavior between classes (algorithms and computation)
- **Behavioural Object Patterns**
  - Use object composition, rather than inheritance. E.g., describing how group of peer objects cooperate to perform a task that no single object can carry out by itself
  - Question: how peer objects know about each other?

Design Patterns – Classification based on purpose

Scope	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Behavioural Patterns (GoF)

Pattern Name	Description
Strategy	**Define a family of algorithms, encapsulate each one, and make them interchangeable (let algorithm vary independently from clients that use it)**
Observer	Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
State	**Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class**
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
Other patterns	Interpreter, Iterator, Mediator, Chain of Responsibility, Template Method

Strategy Design Pattern

Object Behavioural Pattern

Algorithm design through encapsulation

Motivated Scenario

- Suppose you visit a store regularly to buy necessary things
  - Get the normal total price during the weekday
  - Get a discount on the total price during the weekend
  - Get a cashback on the total price during mid-year session

```java
public class SalePricing {
    1 usage
    public double getTotal (int quantity, double price) {
        return price * quantity;
    }
}
```

```java
public class SalePricingWithDiscount {
    1 usage
    private double discount = 0.0;

    public double getTotal (int quantity, double price) {
        return price * quantity * discount;
    }
}
```

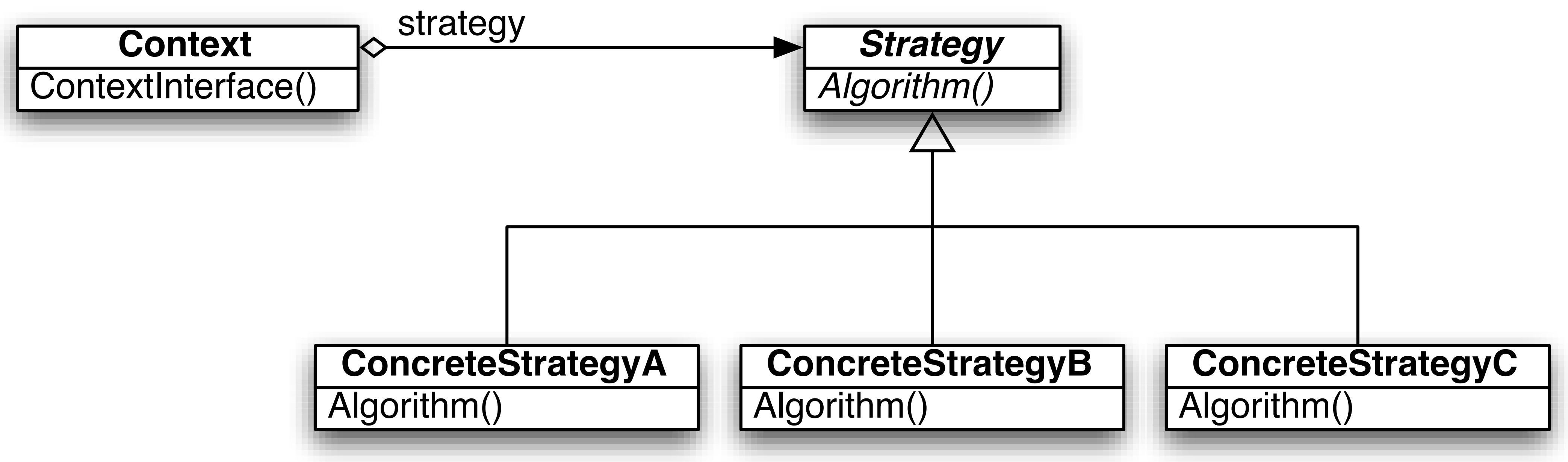
```java
public class SalePricingWithCashBack {
    1 usage
    private double discount = 0.0;
    1 usage
    private double threshold = 0.0;

    public double getTotal (int quantity, double price) {
        double total = price * quantity;
        if (total >= threshold) return total-discount;
        else return total;
    }
}
```

Strategy Design Pattern

- **Purpose/Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from clients that use it
  - Design for varying but related algorithms that are suitable for different contexts
    - Ability to change these algorithms
- **Known as**
  - Policy

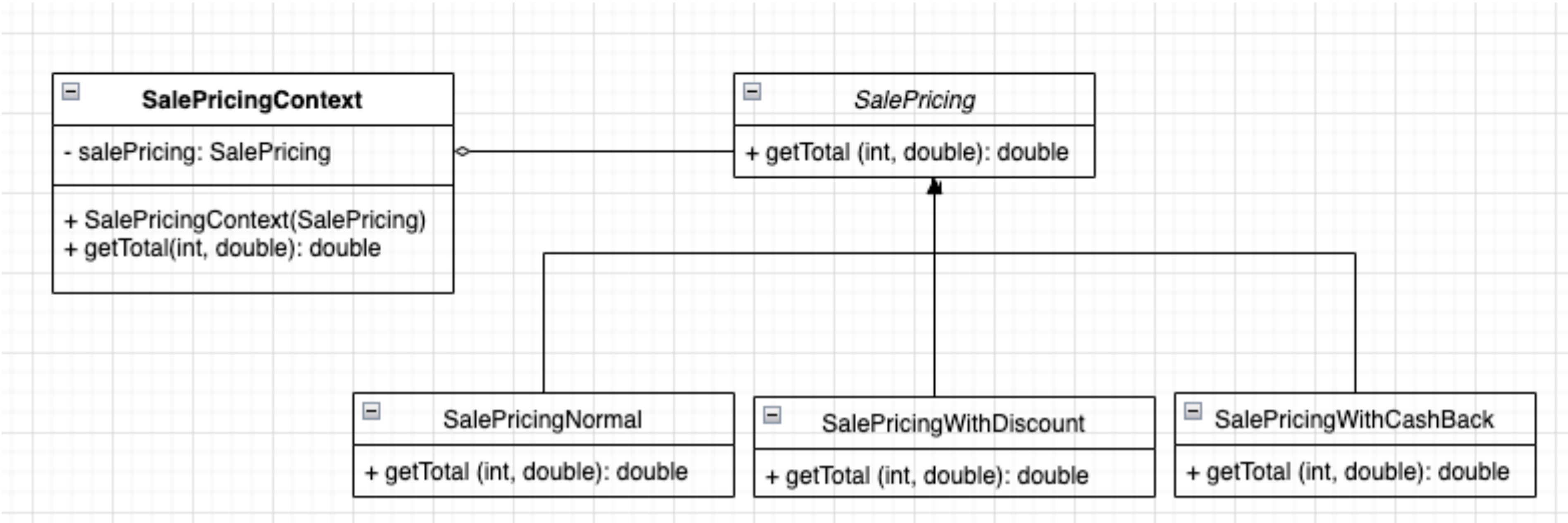
Strategy – Structure



Strategy – Participants

- Strategy
  - Declares an interface common to all supported algorithms
  - Used by context to call the algorithm defined by ConcereteStrategy
- ConcereteStrategy
  - Implements the algorithm using the Strategy interface
- Context
  - Is configured with a ConcereteStrategy object
  - Maintains a reference to a Strategy object
  - May define an interface that lets Strategy access its data

Revisit the Motivated Example



Client's perspective:

```
SalePricing salePricing = new SalePricingNormal();
SalePricingContext context = new SalePricingContext(salePricing);
double total = context.getTotal( quantity: 6, price: 4);
System.out.println("The total money you need to pay is: " + total);
```

Revisit the Motivated Example

```java
public interface SalePricing {
    1 usage   3 implementations
    public double getTotal (int quantity, double price);
}
```

```java
public class SalePricingNormal implements SalePricing{
    1 usage
    public double getTotal (int quantity, double price) {
        return price * quantity;
    }
}
```

```java
public class SalePricingContext {
    2 usages
    private SalePricing salePricing;
    public SalePricingContext (SalePricing salePricing){
        this.salePricing = salePricing;
    }
    public double getTotal(int quantity, double price){
        return salePricing.getTotal(quantity,price);
    }
}
```

```java
public class SalePricingWithDiscount implements SalePricing{
    1 usage
    private double discount = 0.0;

    1 usage
    public double getTotal (int quantity, double price) {
        return price * quantity * discount;
    }
}
```

```java
public class SalePricingWithCashBack implements SalePricing {
    1 usage
    private double discount = 0.0;
    1 usage
    private double threshold = 0.0;

    1 usage
    public double getTotal (int quantity, double price) {
        double total = price * quantity;
        if (total >= threshold) return total-discount;
        else return total;
    }
}
```

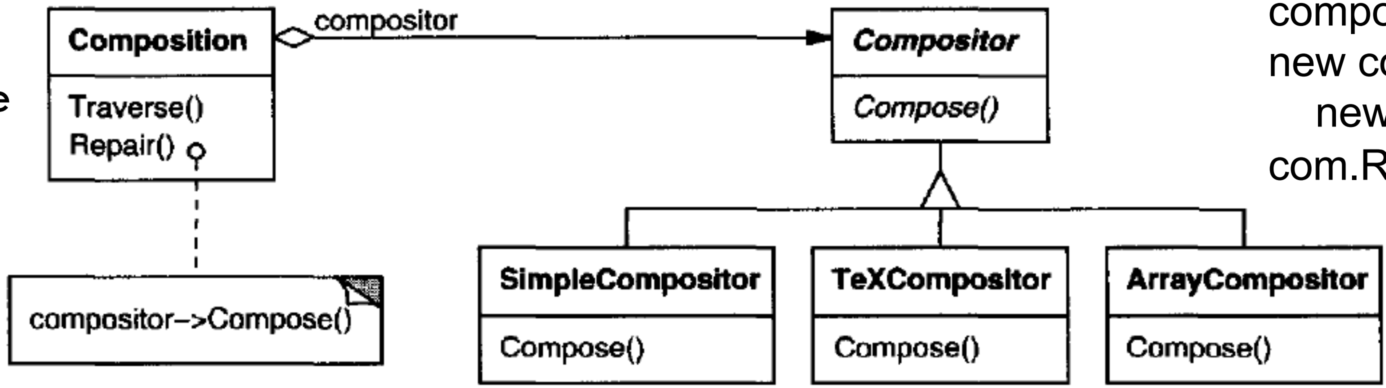
Strategy – Applicability

- Many related classes differ only in their behavior
- You need different variant of an algorithm
- An algorithm uses data that should be hidden from its clients
- A class defines many behaviors that appear as multiple statements in its operations

One more Example (Text Viewer)

- Many algorithms for breaking a stream of text into lines

Maintain & update the line breaks of text



Client perspective:
composition com =
new composition(
new SimpleCompositor( ));
com.Repair( );

Composition perspective:
private compositor com;
public composition (compositor c) {com = c;}
public void Repair ( ) { com.compose();}

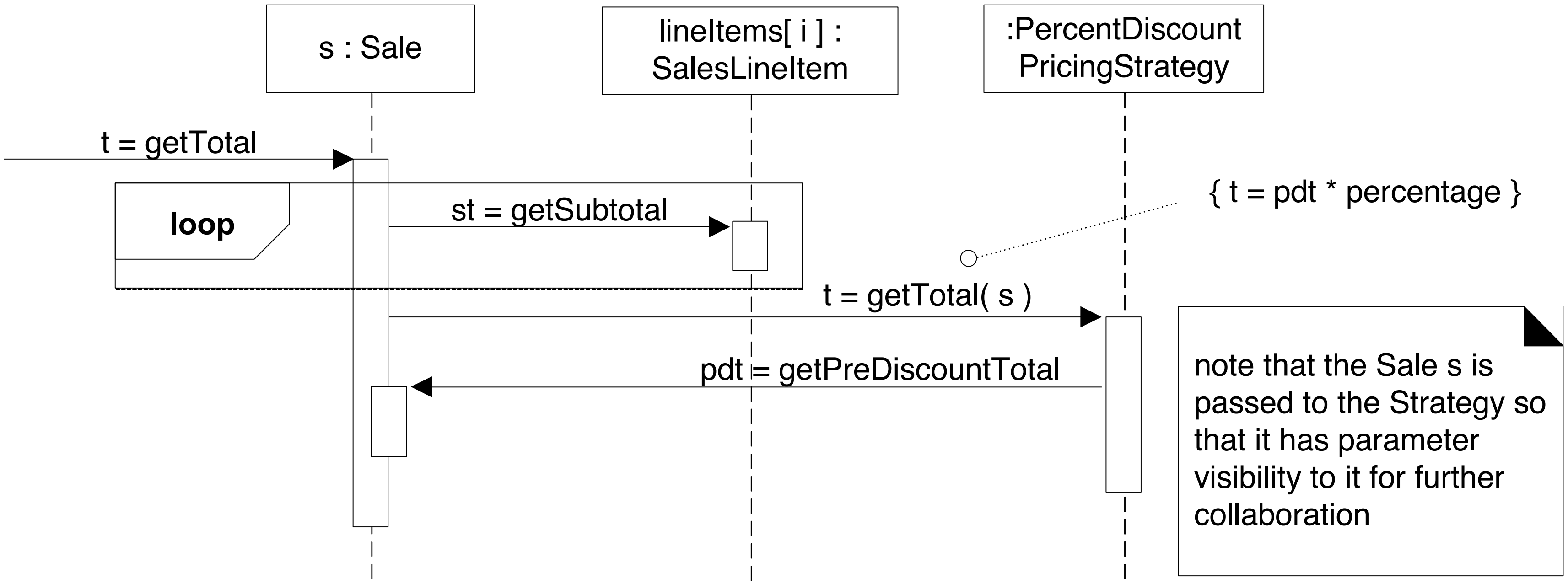
Different line breaking algorithms (strategies)

Strategy – Collaborations

- Strategy and Context interact to implement the chosen algorithm
  - A context may pass all data required by the algorithm to the Strategy
  - The context can pass itself as an argument to Strategy operations

- A context forwards requests from its clients to its strategy
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively

Strategy Collaboration – POS Example



Strategy – Consequences

- Benefits
  - Family of related algorithms (behaviors) for context to reuse
  - Alternative to sub-classing
  - Strategies eliminate conditional statements
  - Provide choice of different implementation of the same behavior
- Drawbacks
  - Clients must be aware of different strategies
  - Communicate overhead between Strategy and Context
  - Increased number of objects in an application

State Design Pattern

Object Behavioural Pattern

Taking control of objects from the inside

A structured way to control the internal behaviour of an object

THE UNIVERSITY OF SYDNEY

Motivated Scenario

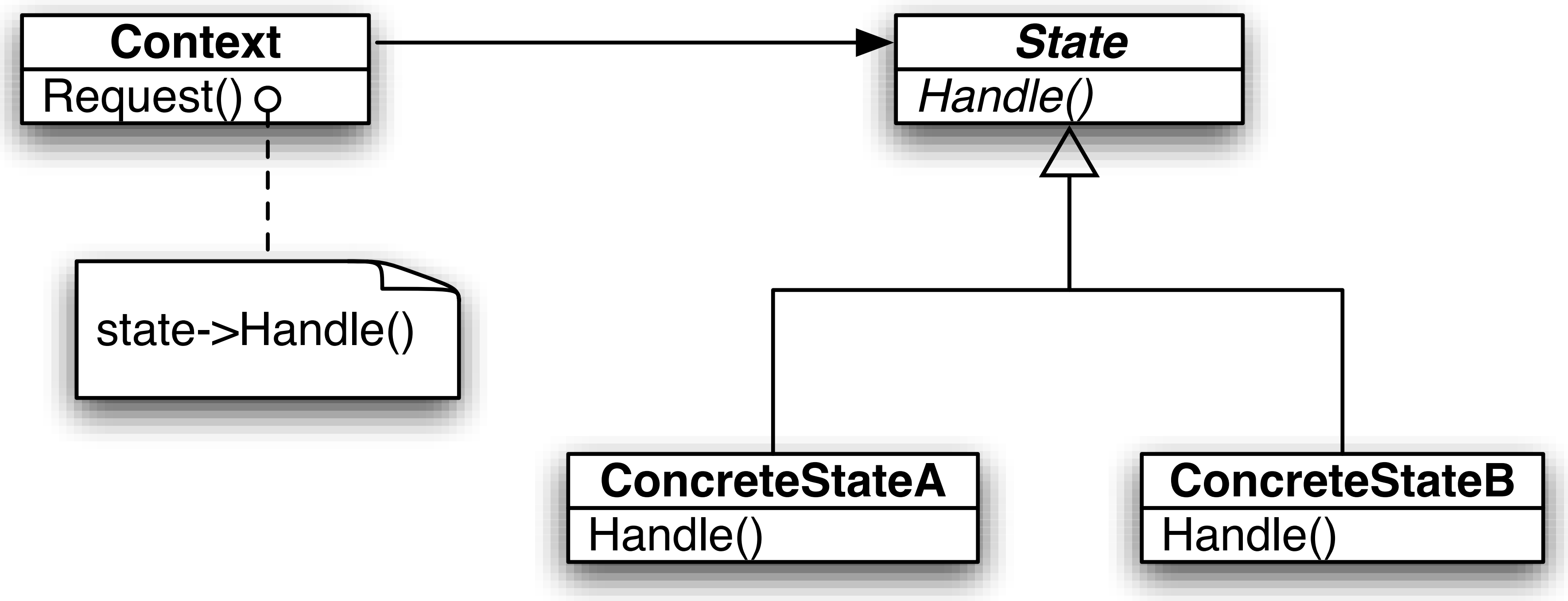
- Suppose you would like to connect to a TCP network and the TCP connection would respond based on its current state
  - Established
  - Listening
  - Closed

State Design Pattern

- **Purpose/Intent**
  - Allow an object to change its behaviour when its internal state changes
  - The object will appear to change its class when the state changes
  - We can use subtypes of classes with different functionality to represent different states, such as for a TCP connection with Established, Listening, Closed as states

- **Known as**
  - Objects for States

State Pattern – Structure



State Pattern – Participants

- Context
  - Defines the interface of interest to clients
  - Maintains an instance of a ConcreteState subclass that defines the current state

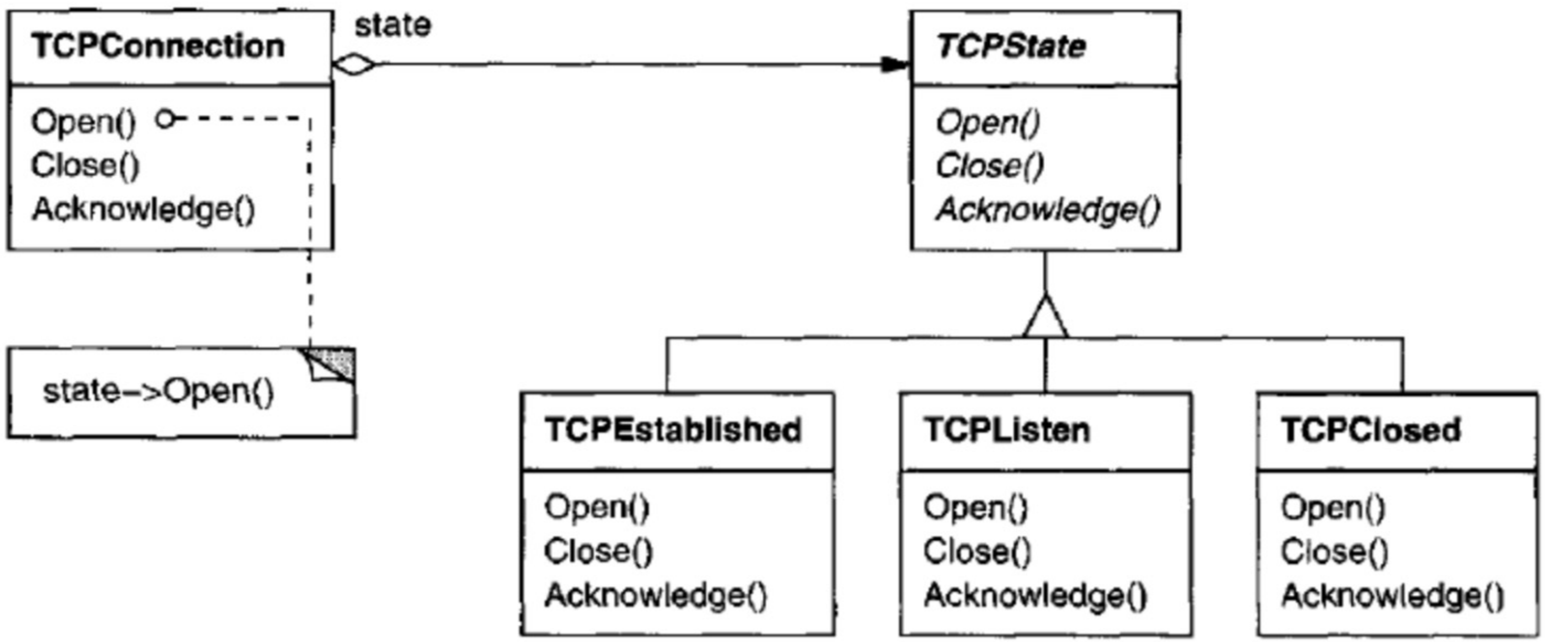
- State
  - Defines an interface for encapsulating the behaviour associated with a certain state of the Context

- ConcreteState subclasses
  - Each subclass implements a behaviour associated with a state of the Context

Revisited the Motivating Scenario

Context

State



ConcreteState

In Class Activity – Code Example

```java
interface State {
    void writeName(StateContext context, String name);
}
class LowerCaseState implements State {
    @Override
    public void writeName(StateContext context, String name) {
        System.out.println(name.toLowerCase());
        context.setState(new MultipleUpperCaseState());
    }
}

class MultipleUpperCaseState implements State {
    /* Counter local to this state */
    private int count = 0;

    @Override
    public void writeName(StateContext context, String name) {
        System.out.println(name.toUpperCase());
        /* Change state after StateMultipleUpperCase's writeName() gets invoked twice */
        if(++count > 1) {
            context.setState(new LowerCaseState());
        }
    }
}
```

In Class Activity – Code Example

```java
class StateContext {
    private State state;

    public StateContext() {
        state = new LowerCaseState();
    }

    /**
     * Set the current state.
     * Normally only called by classes implementing the State interface.
     * @param newState the new state of this context
     */
    void setState(State newState) {
        state = newState;
    }

    public void writeName(String name) {
        state.writeName(this, name);
    }
}
```

In Class Activity – Code Example

```java
public class StateDemo {
    public static void main(String[] args) {
        var context = new StateContext();

        context.writeName("Monday");
        context.writeName("Tuesday");
        context.writeName("Wednesday");
        context.writeName("Thursday");
        context.writeName("Friday");
        context.writeName("Saturday");
        context.writeName("Sunday");
    }
}
```

Question: What is the output of this application?

State Design Pattern

- Applicability
  - Any time you need to change behaviours dynamically, i.e., the state of an object drives its behavior and change its behavior dynamically at run-time
  - There are multi-part checks of an object's state to determine its behaviour, i.e., operations have large, multipart conditional statements that depend on the object's state
- Benefits
  - Removes case or if/else statements depending on state, and replaces them with function calls; makes the state transitions explicit; permits states to be shared
- Limitations
  - Does require that all the states have to have their own objects

State Pattern – Collaborations

- Context delegates state – specific requests to the current ConcreteState object
- A context may pass itself as an argument to the State object handling the request, so the State object access the context if necessary
- Context is the primary interface for clients
  - Clients can configure a context with State objects, so its clients don't have to deal with the State objects directly
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances

State Pattern – Consequences

- Localizes state-specific behaviour for different states
  - Using data values and context operations make code maintenance difficult
  - State distribution across different subclasses useful when there are many states
  - Better code structure for state-specific code (better than monolithic)

- It makes state transition explicit
  - State transitions as variable assignments
  - State objects can protect the context from inconsistent state

- State objects can be shared
  - When the state they represent is encoded entirely in their type

State Pattern – Implementation (1)

- Defining the state transitions
  - Let the state subclasses specify their successor state to make the transition (decentralized)
  - Achieves flexibility – easy to modify and extend the logic
  - Introduces implementation dependencies between subclasses

- Table-based state transitions
  - Look-up table that maps every possible input to a succeeding state
  - Easy to modify (transition data not the program code) but:
    - Less efficient than a functional call
    - Harder to understand the logic (transition criteria is less explicit)
    - Difficult to add actions to accompany the state transitions

State Pattern – Implementation (2)

- When to create and destroy state objects?
  - Pre-create them and never destroy them
    - Useful for frequent state changes (save costs of re-creating states)
    - Context must keep reference to all states
  - Only when they are needed and destroyed them thereafter
    - States are not known at run-time and context change states frequently

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Task for Week 6

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize time for assignment 2
  - JSON configuration text file (tutorial 3)
  - JavaFX and GUI (tutorial 4)

What are we going to learn next week?

- Structural Design Pattern
  - Adapter

- Behavioral Design Pattern
  - Observer