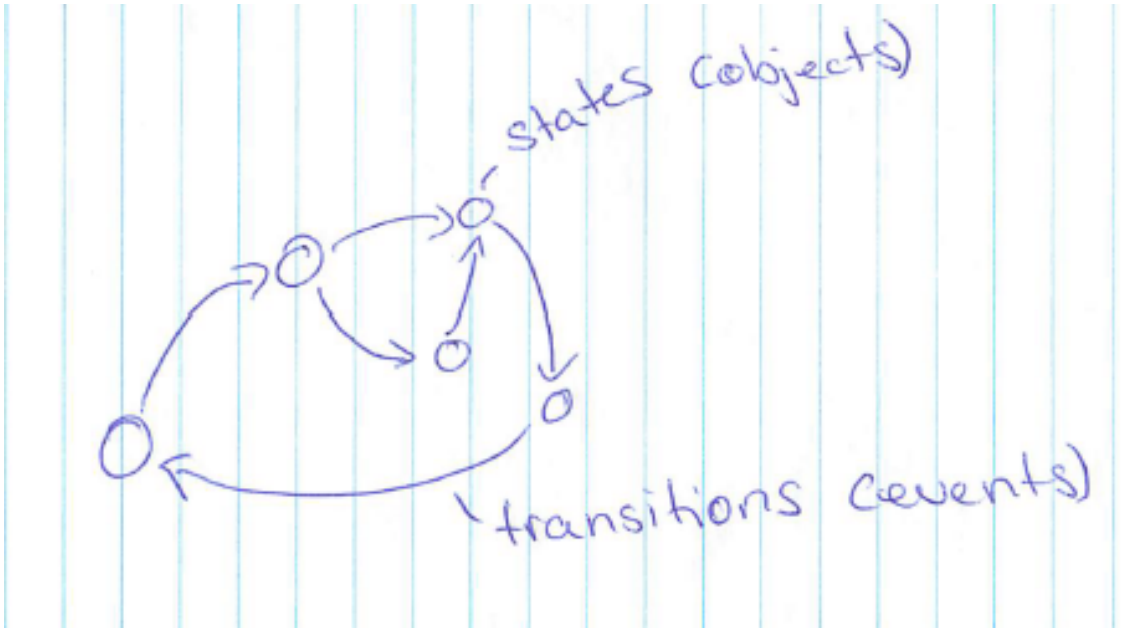
State Pattern

Nadine Dulisch

Contents

What is the State Pattern? 1/2

- comparison: finite state automaton



What is the State Pattern? 2/2

- way to allow an object to alter its behavior and functionality depending on its internal values
- state changes according to the values
- states represented by objects

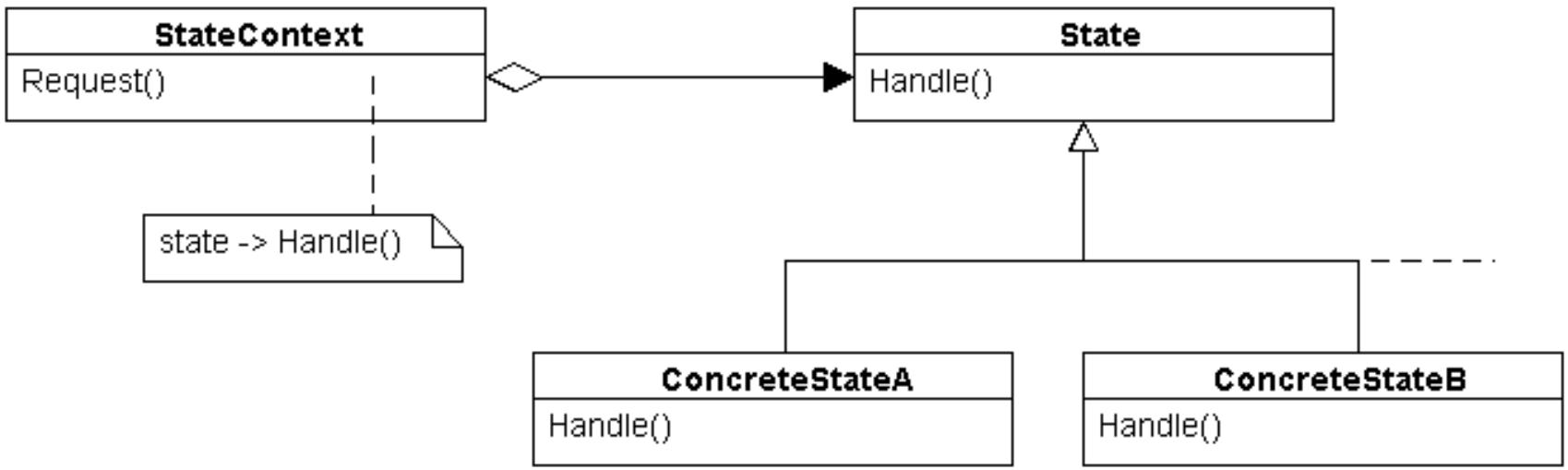
Motivation (example) 1/2

```
public class Product {
    private String name;
    private int numberInStock;
    private int itemsSold;

    public Product(String name, int numberInStock){
        this.name = name;
        this.numberInStock = numberInStock;
    }

    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public int getNumberInStock(){..

    public void sell(int itemsSold){
        if(itemsSold > numberInStock){
            itemsSold = numberInStock;
            this.itemsSold += itemsSold;
            numberInStock -= itemsSold;
        }
        else{
            this.itemsSold += itemsSold;
            numberInStock -= itemsSold;
        }
    }

    public void restock(int itemsRestocked){
        numberInStock += itemsRestocked;
    }
}
```

Motivation (example) 2/2

- Problem:
  - we want the object to respond differently to requests according to its current state
  - we have to determine if we are low on product or if we have enough, when changes are made to the object
  - to determine the state of the object, we have to perform calculations outside of it
  - with large and numerous conditional statements, code can become complex and confusing
- Key idea:
  - Use a state object, that can change itself as the context object (Product) changes to determine the context state

Structure 1/2



Structure 2/2

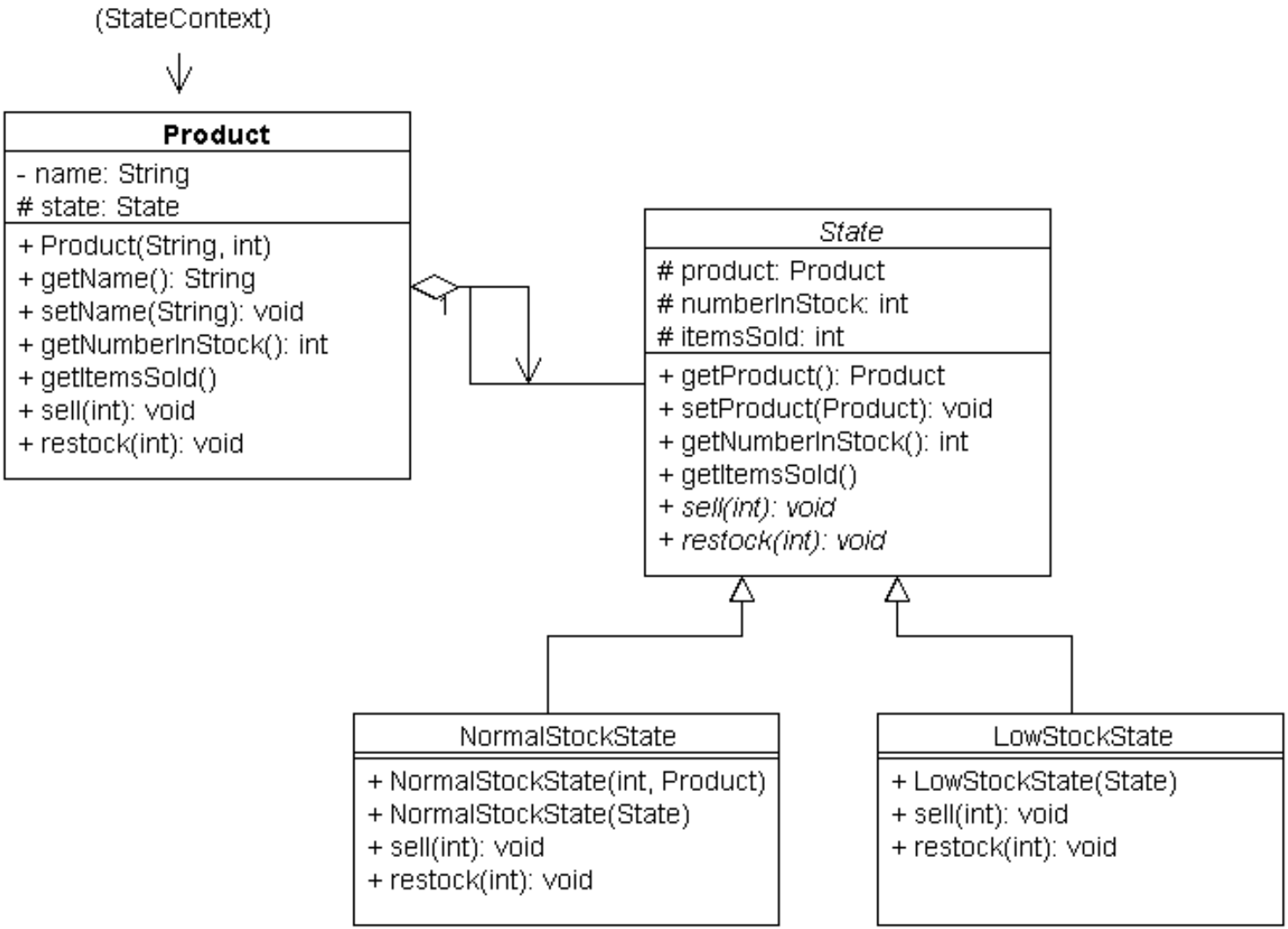
- Context
  - holds the State and is referenced by it
  - delegates state-specific requests to the current ConcreteState object
  - may pass itself to the State object (to give it access to the context)
  - used by clients
- State
  - abstract superclass for individual states
  - defines interface common to all concrete state classes
- ConcreteState
  - subclass for an individual state
  - implements behavior associated with the state it represents

$\Rightarrow$ either Context or ConcreteState subclasses decide about the transition between states

Context

- an object's behavior depends on its state (and changes it at runtime)
- objects have large, mulitpart conditional statements that depend on the state
  - state usually represented by constants
  - several operations often contain same conditional structure

Implementation 1/6



Implementation 2/6

```
public class Product {
    private String name;
    protected State state;

    public Product(String name, int numberInStock){
        this.name = name;
        state = new NormalStockState(numberInStock, this);
    }

    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public int getNumberInStock(){
        return state.numberInStock;
    }
    public int getItemsSold(){
        return state.itemsSold;
    }

    public void sell(int itemsSold){
        state.sell(itemsSold);
    }

    public void restock(int itemsRestocked){
        state.restock(itemsRestocked);
    }
}
```

Implementation 3/6

```
public abstract class State {
    protected Product product;
    protected int numberInStock;
    protected int itemsSold;


    public Product getProduct(){
        return product;
    }
    public void setProduct(Product product){
        this.product = product;
    }
    public int getNumberInStock(){
        return numberInStock;
    }
    public int getItemsSold(){
        return itemsSold;
    }

    abstract public void sell(int itemsSold);
    abstract public void restock(int itemsRestocked);
}
```

Implementation 4/6

```
public class NormalStockState extends State{

    public NormalStockState(int numberInStock, Product product){
        this.numberInStock = numberInStock;
        this.itemsSold = 0;
        this.product = product;
    }

    public NormalStockState(State state){
        this.numberInStock = state.numberInStock;
        this.itemsSold = state.itemsSold;
        this.product = state.product;
    }

    public void sell(int itemsSold){
        if(itemsSold >= this.numberInStock){
            this.product.state = new LowStockState(this);
            this.product.state.sell(itemsSold);
        }
        else{
            this.itemsSold += itemsSold;
            this.numberInStock -= itemsSold;
        }
    }

    public void restock(int itemsRestocked){
        this.numberInStock += itemsRestocked;
    }
}
```

Implementation 5/6

```
public class LowStockState extends State{

    public LowStockState(State state){
        this.numberInStock = state.numberInStock;
        this.itemsSold = state.itemsSold;
        this.product = state.product;
    }

    public void sell(int itemsSold){
        if(itemsSold >= numberInStock){
            itemsSold = numberInStock;
            this.itemsSold += itemsSold;
            numberInStock -= itemsSold;
        }
        else{
            this.product.state = new NormalStockState(this);
            this.product.state.sell(itemsSold);
        }
    }

    public void restock(int itemsRestocked){
        this.numberInStock += itemsRestocked;
        this.product.state = new NormalStockState(this);
    }
}
```

Implementation 6/6

- Demo!

Consequences 1/2

- localizes state-specific behavior and partitions behavior for different states
  - behavior associated with state in 1 object
  - easy addition of new states and transitions (define new subclass)
  - avoids large conditional or case statements
    - easier code maintenance, modification and extension
    - code more explicit
  - but increases number of classes, less compact

Consequences 2/2

- makes state transitions explicit
  - if state is only defined by internal data values, there's no explicit representation of the transitions
  - state objects can protect Context from inconsistent internal states
    1. state transitions atomic (from Context's perspective)
    2. rebinding only one variable (Context's state object)
- state objects can be shared (by contexts)
  - no instance variables
  - state represented by their type

Resources

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides John: *Design Patterns*, Addison-Wesley Publishing Company, 1995.

Lasater, Christopher G.: *Design Patterns*, Wordware Publishing, 2007.