Arbeitsblatt: State Pattern: Lösung

Wenn der Parser mit dem State-Pattern umgesetzt wird, dann muss als erstes das Interface für die State-Objekte definiert werden. Üblicherweise wird dabei pro Ereignis eine Methode definiert mit dem Namen *on<Event>* oder *handle<Event>*. Zudem muss das aktuelle State-Objekt jeweils angeben, welches sein Nachfolgezustand ist.

Die Daten, die von den State-Objekten bearbeitet werden, müssen entweder global abgelegt werden oder jeweils dem State als Parameter übergeben werden. Mit dem zweiten Ansatz ergibt sich für diese Anwendung folgendes Interface:

```
interface State {
    default State handleDigit(FloatData data, int val) { return ERROR; }
    default State handleE(FloatData data) { return ERROR; }
    default State handleDot(FloatData data) { return ERROR; }
    default State handlePlus(FloatData data) { return ERROR; }
    default State handleMinus(FloatData data) { return ERROR; }
}
```

wobei die zustandsübergreifenden Daten in einer Instanz der Klasse `FloatData` abgelegt werden:

```
class FloatData {
    double m = 0, quo=10;
    int exp = 0, exp_sign = 1;
    double getValue() { return m * Math.pow(10, exp_sign * exp); }
}
```

Der Parser sieht dann wie folgt aus:

```
private static double parseFloat(String str) {
    State s = S0; // initial state
    FloatData data = new FloatData();
    int pos = 0;
    while(s != ERROR && pos < str.length()){
        char ch = str.charAt(pos++);
        if(Character.isDigit(ch)) s = s.handleDigit(data,
                                        Character.getNumericValue(ch));
        else if(ch == '.') s = s.handleDot(data);
        else if(ch == '+') s = s.handlePlus(data);
        else if(ch == '-') s = s.handleMinus(data);
        else if(ch == 'E') s = s.handleE(data);
        else if(ch == 'e') s = s.handleE(data);
        else s = ERROR;
    }
    if(s == S3 || s == S6){
        return data.getValue();
    } else {
        throw new IllegalArgumentException();
    }
}
```

Der Zustand `S0` ist in der folgenden Klasse implementiert (wobei eine Instanz dieser Klasse über die statische Variable `S0` referenziert wird). Die anderen Zustände sehen analog aus.

```
class S0 implements State {
    @Override public State handleDigit(FloatData data, int value) {
        data.m = value; return S1;
    }
    @Override public State handleDot(FloatData data) {
        return S2;
    }
}
```

Diese Lösung ist im bereitgestellten Projekt in der Klasse `FloatConverter2` implementiert.

Im Projekt mit den Parser-Lösungen sind auch noch die Lösungen `FloatConverter3` sowie `FloatConverter4` enthalten.

Die Lösung `FloatConverter3` ist im wesentlichen dieselbe Lösung wie `FloatConverter2`, ausser dass das State-Interface und deren Implementierungen als `enum` realisiert sind. Der Vorteil ist, dass die State-Instanzen nicht instanziiert werden müssen.

Bei der Variante `FloatConverter4` haben die State-Objekte eine Referenz auf den Kontext. Diese Referenz wird in den Konstruktoren der State-Implementierungen gesetzt und verwendet, um im Kontext einen Zustandswechsel auszuführen. Die Methoden im State-Interface geben daher kein State-Objekt zurück:

```
private static interface State {
    void handleDigit(int val);
    void handleE();
    void handleDot();
    void handlePlus();
    void handleMinus();
}
```

Dass die Implementierungen des State-Interfaces einen bestimmten Konstruktor anbieten, kann in Java nicht im Interface definiert werden, dies wird in der abstrakten Basisklasse `AbstractState` eingeführt. Um den Zustand des Kontext zu ändern wird ebenfalls direkt auf diesen zugegriffen, d.h. der Parameter vom Typ `FloatData data` ist bei den Methoden des State-Interfaces ebenfalls nicht mehr vorhanden.

Diese Lösung hat jedoch den Nachteil, dass der Kontext und die State-Objekte stark aneinander gekoppelt sind, was durch Einführung eines Interfaces `Context` welches den Kontext abstrahiert etwas entschärft wird:

```
private static interface Context {
    void setState(State s);
    FloatData getData();
}
```