State pattern (Zustandsmuster)

Hintergrund

Das Zustands-Entwurfsmuster ist eine Methode zur Modellierung von zustandsabhängigen Verhalten eines Objekts, resp. es soll sein äusseres Verhalten zur Laufzeit aufgrund seines Zustands ändern. Das Verhalten eines Objekts ändert sich entsprechenden seines internen Zustand. Für alle möglichen Zustände wird eine definierte Schnittstelle bereitgestellt. Die Schnittstelle wird für jeden einzelnen Zustand durch eine separate Klasse implementiert.

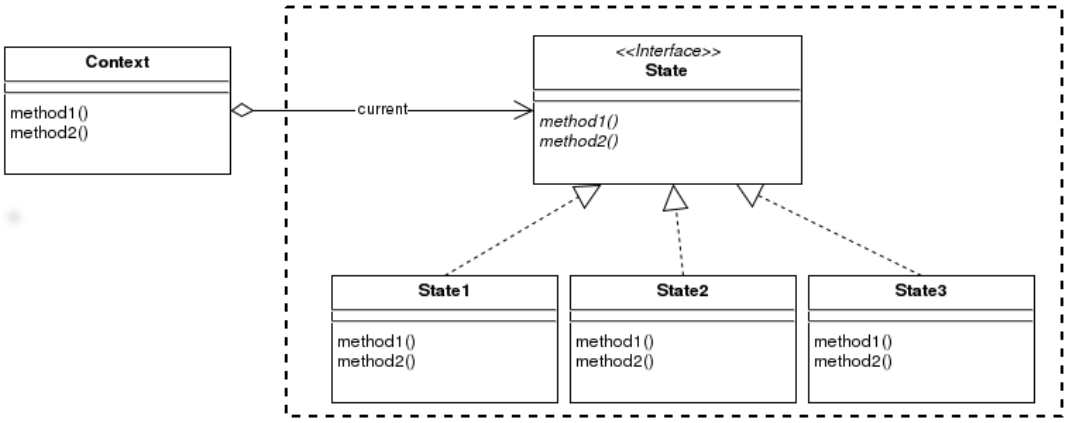

Abbildung 1: State pattern

Vor- und Nachteile

Weitere Zustände und Verhalten können einfach hinzugefügt werden. Die Weiterverwendung der Zustände und bessere Wartbarkeit sind ebenfalls Vorteile. Unleserlichkeit und Unübersichtlichkeit wird entgegen gewirkt, da umfangreiche if-then-else- und switch-Konstrukte vermieden werden.

Ein Nachteil ist, das bei einfachen Zustand ein erhöhter Implementierungsaufwand besteht, denn alle Aktionen und Zustände müssen definiert werden. Dies führt zu einer erhöhten Klassenanzahl, da die Schnittstelle komplett implementiert werden muss.

Mögliche Implementation

Für eine Türe zeigt nachfolgende Inplementation eine Anwendung des Zustandsmuster. Die Türe hat die folgenden Zustände und Aktionen:

- offen → schliessen
- geschlossen → abschliessen
- verschlossen → öffnen

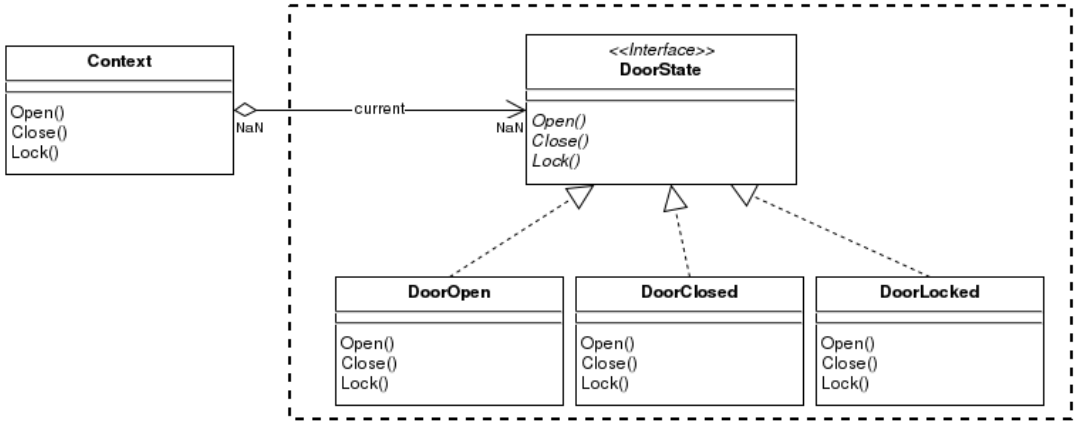

Abbildung 2: State pattern

Das Interface listet die möglichen Aktionen gemäss der obrigen Aufzählung auf.

Listing 1: Interface

```
// State of the door
public interface DoorState {
    public String close();
    public String lock();
    public String open();
}
```

Analog Abbildung 2 gibt es drei Zustände und darausfolgende Aktionen.

Listing 2: Zustand: Türe geöffnet

```
//Concrete State: DoorOpen
public class DoorOpen implements DoorState {
    public String close() {
        return "Can close the door.";
    }

    @Override
    public String lock() {
        return "Can not lock the door.";
    }

    @Override
    public String open() {
        return "Door is already open.";
    }
}
```

Listing 3: Zustand: Türe geschlossen

```
//Concrete State: DoorClosed
public class DoorClosed implements DoorState {
    @Override
    public String lock() {
        return "Lock the door.";
    }

    @Override
```

```
9       public String close() {
10          return "Door is already closed.";
11      }
12
13      @Override
14      public String open() {
15          return "Open the door.";
16      }
17  }
```

Listing 4: Zustand: Türe verschlossen

```
1   //Concrete State: DoorLocked
2   public class DoorLocked implements DoorState {
3       @Override
4       public String open() {
5           return "Can open the door. ";
6       }
7
8       @Override
9       public String close() {
10          return "Door is closed.";
11      }
12
13      @Override
14      public String lock() {
15          return "Door is alreay locked.";
16      }
17  }
```

Die Kontext-Klasse hat eine Verbindung zum Interface.

Listing 5: Context

```
1   // Context
2   public class Door implements DoorState {
3       DoorState doorState;
4
5       public Door(DoorState doorState) {
6           this.doorState = doorState;
7       }
8
9       public void setDoorState(DoorState doorState) {
10          this.doorState = doorState;
11      }
12
13      @Override
14      public String close() {
15          return doorState.close();
16      }
17
18      @Override
19      public String lock() {
20          return doorState.lock();
21      }
22
23      @Override
24      public String open() {
```

```
25          return doorState.open();
26      }
27  }
```

Die State-Klasse demonstriert die Verwendung.

Listing 6: State-Klasse

```
1   public class State {
2       public static void main(String[] args) {
3           Door door = new Door(new DoorOpen());
4           System.out.println("Door is open:");
5           System.out.println("- " + door.open());
6           System.out.println("- " + door.close());
7           System.out.println("- " + door.lock());
8   
9           door.setDoorState(new DoorClosed());
10          System.out.println("Door is closed:");
11          System.out.println("- " + door.open());
12          System.out.println("- " + door.close());
13          System.out.println("- " + door.lock());
14  
15          door.setDoorState(new DoorLocked());
16          System.out.println("Door is locked:");
17          System.out.println("- " + door.open());
18          System.out.println("- " + door.close());
19          System.out.println("- " + door.lock());
20      }
21  }
```

Je nach Zustand der Türe lassen sich natürlich nicht alle Aktion ausgeführt. Die nachfolgende Ausgabe zeigt dies.

Listing 7: Output

```
1   Door is open:
2   - Door is already open.
3   - Can close the door.
4   - Can not lock the door.
5   Door is closed:
6   - Open the door.
7   - Door is already closed.
8   - Lock the door.
9   Door is locked:
10  - Can open the door.
11  - Door is closed.
12  - Door is alreay locked.
```

Neben der Implementierung als Klassen, was vorheriges Beispiel zeigt, lässt sich das State-Pattern auch als Enumeration umsetzen.

Unsere Implementation

Wir benutzen das State-Pattern für die Darstellung einer angepassten Oberfläche um den unterschiedlichen Anforderungen und benötigten Berechtigungen der Benutzer gerecht zu werden.

Listing 8: UIPermission

```
1   package ch.bfh.ti.soed.white.mhc_pms.security;
2   public class UIPermission {
3       protected boolean isNewPatientAllowed = false;
```

```
4       protected boolean isNewCaseAllowed = false;
5       protected boolean isEditCaseDataAllowed = false;
6       protected boolean isNewPatientProgressEntryAllowed = false;
7       protected boolean isEditPatientProgressEntryAllowed = false;
8       protected boolean isDeletePatientProgressEntryAllowed = false;
9       protected boolean isNewDiagnosisAllowed = false;
10      protected boolean isEditDiagnosisAllowed = false;
11      protected boolean isDeleteDiagnosisAllowed = false;
12      protected boolean isNewMedicationAllowed = false;
13      protected boolean isEditMedicationAllowed = false;
14      protected boolean isDeleteMedicationAllowed = false;
15
16      public boolean isNewPatientAllowed() {
17          return this.isNewPatientAllowed;
18      }
19
20       [...]
21  }
```

Exemplarisch wird nachfolgend die Berechtigung für ein Therapist gezeigt. Die Klasse für die Berechtigungen des Therapist erweitert die Basis-Berechtigungen.

Listing 9: Erweiterte Berechtigungen MedicalStaff

```
1   class ExtendedMedicalStaffPermission extends UIPermission {
2       protected ExtendedMedicalStaffPermission() {
3           this.isNewPatientAllowed = true;
4           this.isNewCaseAllowed = true;
5       }
6   }
```

Listing 10: Erweiterte Berechtigungen Therapist

```
1   class TherapistPermission extends ExtendedMedicalStaffPermission {
2       protected TherapistPermission() {
3           this.isEditCaseDataAllowed = true;
4           this.isNewPatientProgressEntryAllowed = true;
5           this.isEditPatientProgressEntryAllowed = true;
6           this.isDeletePatientProgressEntryAllowed = true;
7           this.isNewDiagnosisAllowed = true;
8           this.isEditDiagnosisAllowed = true;
9           this.isDeleteDiagnosisAllowed = true;
10      }
11  }
```

Verwendet werden die Daten in der UI-Klassen, welche die Berechtigungen prüft. Gezeigt wird nur der relevante Abschnitt.

Listing 11: UI-Klasse

```
1     [...]
2           UIPermission permission = UserSingleton.getUser().getUIPermission();
3           this.btnNewPatient.setEnabled(permission.isNewPatientAllowed());
4           this.btnNewCase.setEnabled(permission.isNewCaseAllowed());
5     [...]
```

Das Benutzer-Objekt erzeugt das State-Objekt.

Listing 12: Benutzer-Klasse

```
1   public abstract class MhcPmsUser {
```

```
2
3       private UIPermission uiPermission;
4       protected MhcPmsDataAccess dataAccess;
5
6       protected <E extends MhcPmsUser> MhcPmsUser(Class<E> clazz) {
7           this.uiPermission = UIPermissionFactory.createUIState(clazz);
8           this.dataAccess = MhcPmsDataAccess.getInstance();
9       }
10
11      public <E extends MhcPmsItem> boolean incrementCurrentItem(Class<E> clazz) {
12          return false;
13      }
14
15      public <E extends MhcPmsItem> boolean decrementCurrentItem(Class<E> clazz) {
16          return false;
17      }
18
19      public <E extends MhcPmsItem> boolean setCurrentItem(Class<E> clazz, Object va
20          return false;
21      }
22
23      public final UIPermission getUIPermission() {
24          return this.uiPermission;
25      }
26  }
```

Die UIPermissionFacotry-Klasse erzeugt daraufhin das Permission-Objekt.

Listing 13: UIPermissionFacotry-Klasse

```
1   class UIPermissionFactory {
2       private static Map<String, UIPermission> permissionsMap;
3       static {
4           permissionsMap = new HashMap<String, UIPermission>();
5           permissionsMap.put(Psychiatrist.class.getName(), new PsychiatristPermissio
6           permissionsMap.put(Therapist.class.getName(), new TherapistPermission());
7           permissionsMap.put(ExtendedMedicalStaff.class.getName(), new ExtendedMedi
8           permissionsMap.put(MedicalStaff.class.getName(), new UIPermission());
9       }
10
11      static <E extends MhcPmsUser> UIPermission createUIState(Class<E> clazz) {
12          return permissionsMap.get(clazz.getName());
13      }
14  }
```