GUI-Programmierung in Java

Rahel Lanz, Patrick Bucher
12. Dezember 2016

Ablauf

1) OOP-Folien und Beispiele
   a) Rahel Lanz: GUI-Komponenten, Layout-Manager, Ereignisgesteuerte Programmierung
   b) Patrick Bucher: Plattformunabhängigkeit, Toolkits allgemein und Swing im Speziellen, Code-Beispiel in Swing
   c) Rahel Lanz: JavaFX, MVC, Code-Beispiel in JavaFX
   d) Patrick Bucher: GUI-Designer, Demo NetBeans GUI-Builder (Swing)
   e) Rahel Lanz: Demo JavaFX GUI-Designer
   f) Patrick Bucher: Empfehlungen und Zusammenfassung

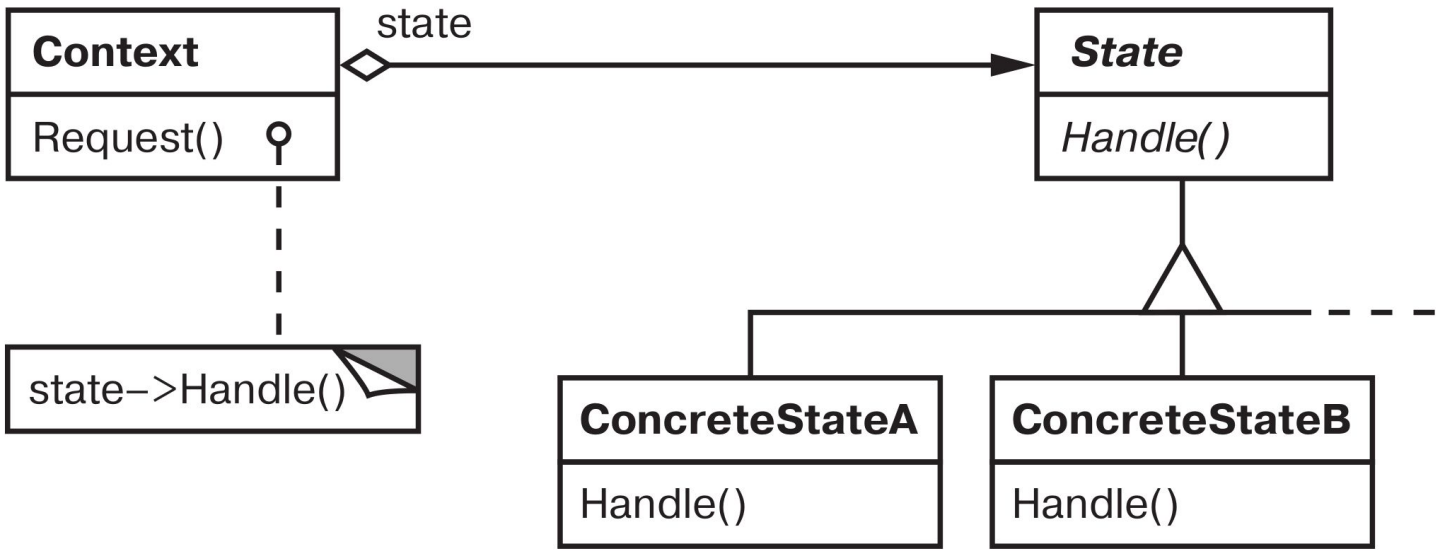
2) Eigene Beispiele
   a) Rahel Lanz: Refactoring einer JavaFX-Applikation bestehend aus einer einzigen Klasse nach MVC
   b) Patrick Bucher: Demonstration des State-Patterns anhand eines Texteditors (mit Aufgaben)

Das State-Pattern

GoF 305

State-Pattern (1) – Theorie

State (305) Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class. (GoF)



State-Pattern (2) – Fallbeispiel Texteditor

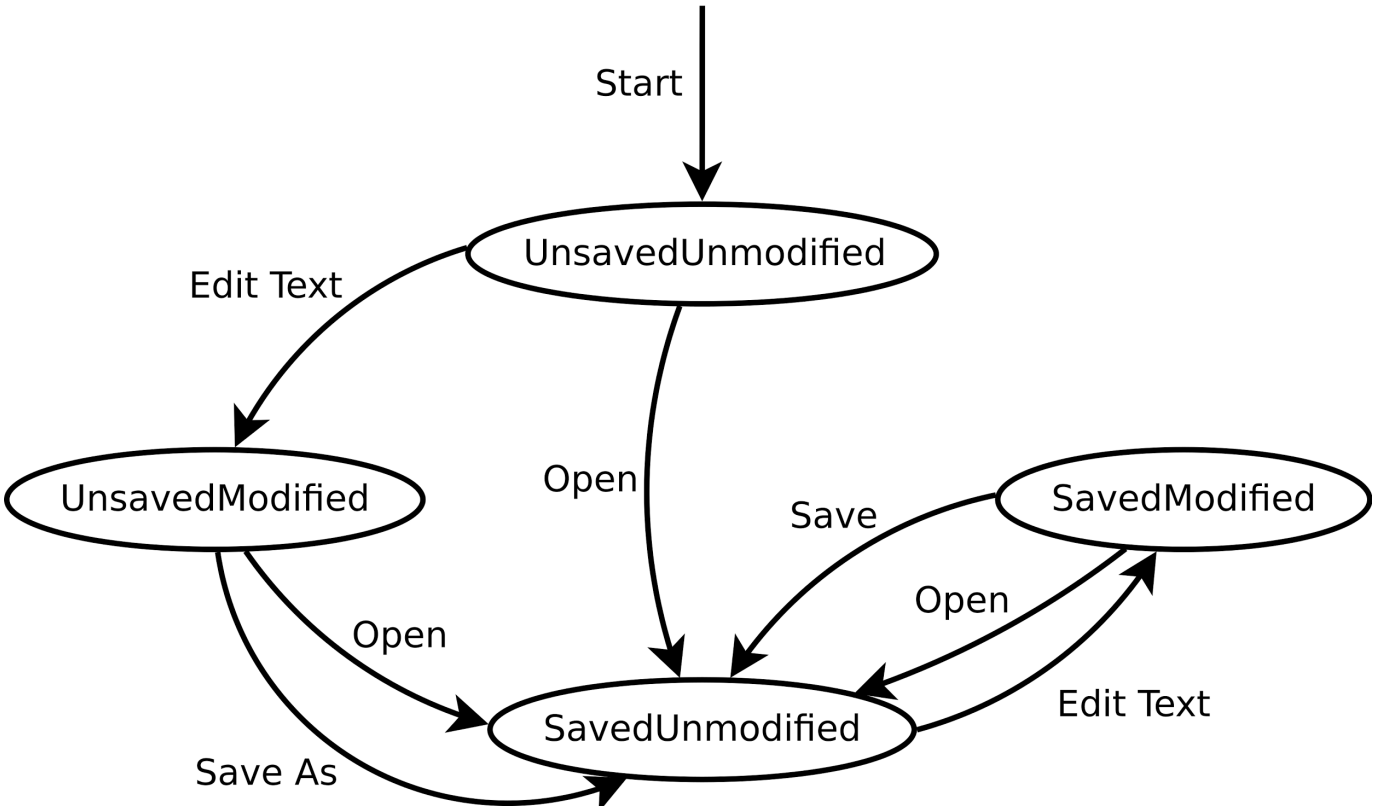
Der Inhalt eines Texteditors kann entweder modifiziert oder unmodifiziert sein. Er ist entweder als Datei abgespeichert oder nicht abgespeichert.

Wir unterscheiden nach zwei Kategorien (Modifikation und Speicherung) mit je zwei Zuständen und erhalten vier Klassen:

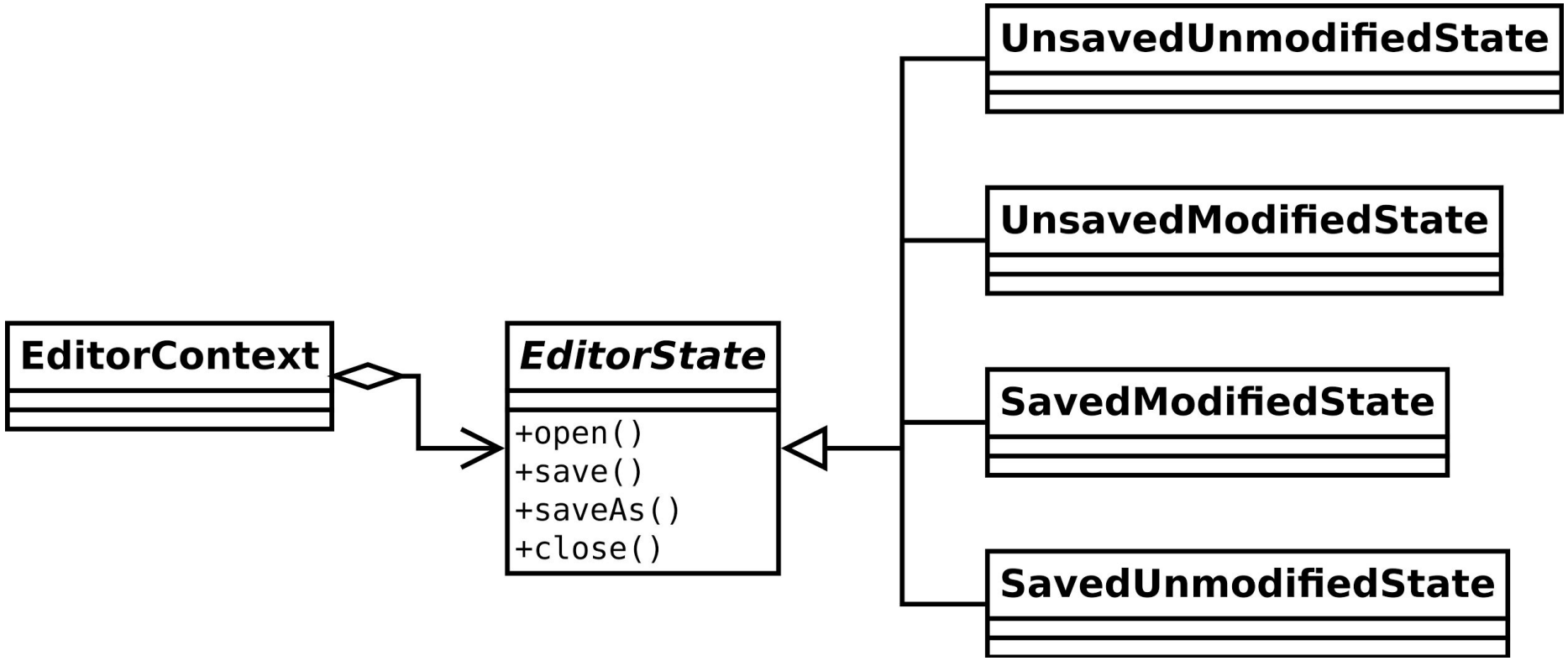
1. `UnsavedUnmodifiedState`
2. `UnsavedModifiedState`
3. `SavedUnmodifiedState`
4. `SavedModifiedState`

Frage: Was würde passieren, wenn eine weitere Kategorie dazu käme?

State-Pattern (3) – Endlicher Automat



State-Pattern (4) – Klassendiagramm



State-Pattern (5) – Implementierung

[Code]

State-Pattern (6) – Konsequenzen

+ Der Texteditor hat immer einen expliziten Status.
+ Das Verhalten jeder Aktion ist immer genau definiert.
+ Die Event-Handling-Methoden sind sehr schlank geworden.
+ `EditorContext` und `EditorState` sind lose über ein Interface gekoppelt…
- …greifen jedoch vielerorts und mit verschiedenen Methoden aufeinander zu.
- Es wurden viele Klassen erstellt, die eine ähnliche Funktionalität haben.
- Dadurch kommt es zu Code-Duplikation – oder zur Auslagerung von Code in “Utility”- oder “Helper”-Klassen (*code smell*).

Statt eines State-Interfaces könnte eine abstrakte Klasse implementiert werden, die gemeinsam genutzten Code beinhaltet.

Eine lose Kopplung ist schwer zu erreichen, wenn GUI-Funktionalität auf verschiedene Klassen verteilt wird.

State-Pattern (7) – Aufgabe 1

Lege einen neuen Branch (`aufgabe-1`) an. Implementiere dann den Menü-Eintrag *New* und die Operation `new()` im neuen Branch.

1. Welche neuen Statusübergange ergeben sich dadurch?
2. Was ist zu tun, wenn der Benutzer Änderungen am Text vorgenommen hat und auf *New* klickt?
3. Welche Klassen müssen verändert und/oder erweitert werden?
4. Reflexion: Waren diese Änderungen einfach zu bewältigen? Erleichtert oder erschwert das State-Pattern die Erweiterbarkeit?

Zusatzaufgabe: Was passiert, wenn das Fenster nicht über *Close* geschlossen wird? Muss das Verhalten erweitert werden? Wenn ja, wie?

State-Pattern (8) – Aufgabe 2

Lege einen neuen Branch (`aufgabe-2`) an. Ersetze im neuen Branch das Interface `EditorState` durch eine abstrakte Klasse.

1. Welcher Code der `EditorState`-Implementierungen (bzw. neu: Unterklassen) lässt sich in die abstrakte Klasse verschieben?
2. Wäre der Code von `DialogUtils` auch besser in der abstrakten Klasse aufgehoben?
3. Muss für diese Änderungen etwas an `EditorContext` oder an `TextEditor` verändert werden?
4. Reflexion: Wie hat sich diese Änderung auf den Code ausgewirkt? Was funktioniert hier besser: eine abstrakte Klasse oder ein Interface?

State-Pattern (9) – Refactoring

Wer glaubt, das State-Pattern führe zu unnötig kompliziertem und schlecht erweiterbarem Code, der kann ein Refactoring vornehmen:

1. Neuer Branch `refactoring` (ausgehend von `master`!)
2. Löschen der Interfaces `EditorContext` und `EditorState`
3. Handhabung der Operationen direkt in den Event-Handling-Methoden bzw. in neuen Zusatzmethoden der Klasse `TextEditor`
4. “Ausschlachten” und Löschen aller `EditorState`-Implementierungen

Ist der Code nun verständlicher?

Lassen sich der Menü-Eintrag *New* und die Operation `new()` vor oder nach dem Refactoring einfacher einfügen? (Versuche Aufgabe 1 auch im Branch `refactoring` zu lösen.)