# Stateful View Controllers in iOS

Für die Entwicklung einer zustandsbasierte Benutzungsschittstelle eines mobilen Gerätes zeigen wir drei verschiedene Ansätze auf. Wir vergleichen die Ansätze miteinander und führen einen Ansatz genauer aus, welchen wir im Rahmen der Entwicklung eines Störmeldesystems auf dem iPhone entwickelt haben. Dieser Ansatz basiert auf dem State Pattern und verwendet eine einzige einfache tabellarische Darstellung, welche dynamisch mit den zustandsabhängigen Elementen eingefüllt wird.

Moritz Dietsche | moritz.dietsche@fhnw.ch

Das in diesem Artikel beschriebene User Interface einer iOS App ist im Rahmen des KTI-Projekts „App für Störmeldesysteme“ entwickelt worden[1]. Diese mobile Anwendung ermöglicht berechtigten Personen den Empfang von Statusmeldungen von Störmeldegeräten. Die erforderliche Zugriffsberechtigung erfolgt über codebasiertes Geräte-Pairing zwischen dem Störmeldesystem und der App.

**Problemstellung**

Um das oben angesprochene Pairing zweier Geräte durchzuführen, wird eine Benutzeroberfläche benötigt (Abb. 1), welche auf Desktopsystemen oft in Form eines Assistenten oder „Wizards“ umgesetzt wird. Ein solcher Assistent bietet eine Möglichkeit, Schritt für Schritt einen Automaten zu durchlaufen, wobei die Zustandsübergänge entweder vom Benutzer (Auswählen einer Option) oder durch das System (zum Beispiel eine Netzwerkanfrage) ausgelöst werden.

Abbildung 2 zeigt einen Automaten, welcher als Grundlage für einen Assistenten dienen könnte. Er verfügt über Verzweigungen und über Zustandsübergänge, welche – im Gegensatz zu rein benutzergesteuerten Assistenten – entweder durch den Benutzer oder durch äussere Einflüsse wie eingehende Push Notifications ausgelöst werden. Wichtig zu beachten ist hierbei, dass die Zustände verschiedene User Interfaces repräsentieren, auch wenn diese gemäss der Geschäftslogik einem einzigen Zustand entsprechen. Je nach Ausführungspfad endet der Automat in einem anderen Zustand.

Bei der Umsetzung eines Assistenten der zuvor geschilderten Art auf einer Mobilplattform werden üblicherweise folgende Arbeitsschritte realisiert:

- Erstellung der einzelnen Views;
- Navigation zwischen den Views;
- Definition der Logik, d. h. welche Zustandsübergänge sind möglich;
- Kommunikation zwischen den Zuständen und der übergeordneten Einheit;
- Erweiterbarkeit des Automaten.


Abbildung 1: View von PairingStateWaitForResponse nach Annahme durch die Gegenseite

**Lösungsansätze**

Wir beschränken uns im Folgenden auf zwei Kriterien, nach welchen wir die verschiedenen Lösungsansätze beurteilen. Diese sind einerseits die Platzierung der Logik des Automaten, zentral oder auf verschiedene Zustände verteilt. Andererseits unterscheiden wir zwischen Lösungen mit einer und mit mehreren Views. Wir erhalten die vier Ansätze in Tabelle 1.

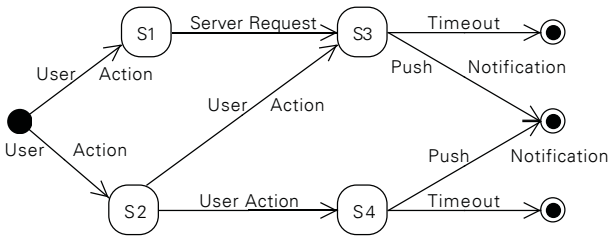


Abbildung 2: Ein Automat, welcher in Form eines Assistenten implementiert werden könnte

1 KTI-Projektnummer 13637.1PFFLE-ES

| | eine View | mehrere Views |
|---|---|---|
| verteilte Logik | Lösungsansatz 1: einfach | hier nicht sinnvoll |
| zentrale Logik | Lösungsansatz 2: grosse View | Lösungsansatz 3: TableView |

Tabelle 1: Verschiedene Lösungsansätze in Abhängigkeit der Anzahl Views und der Verteilung der Logik der Zustandsmaschine

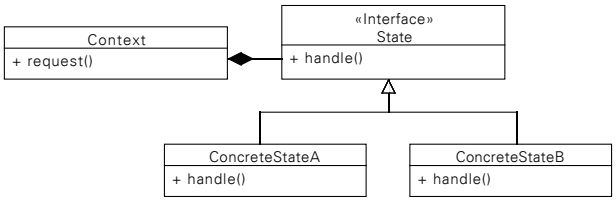


Abbildung 3: UML Klassendiagramm des State Patterns

Wir beginnen mit einem ersten einfachen Ansatz. Dieser besteht darin, die einzelnen Schritte als separate Views zu implementieren. Diese könnten nun manuell gestaltet werden. Für die Geschäftslogik fügen wir jeder View noch einen View Controller hinzu. Dieser übernimmt die Steuerung der View und verfügt über eine Methode *getNext()*, welche den nächsten Zustand zurückgibt.

In einem nächsten Schritt fügen wir die erstellten Controller einem Container Controller hinzu, welche die Aufgabe hat, die Zustandsübergänge auszuführen. Dies reicht eigentlich schon, um die oben genannten Anforderungen zu erfüllen. Wir benötigen für $n$ Zustände lediglich $n$ Views und $n$+1 Controller. Die Vorteile dieses Ansatzes sind:

- simple Implementierung der Zustände;
- überschaubarer Aufwand für kleinere Anwendungen (Anzahl Views und Controller steigt linear mit der Anzahl Zustände);
- grafische Umsetzung der Views ermöglicht einfaches Ausdrucken des Storyboards unter anderem für Usability-Testing.

Dennoch bleiben ein paar mögliche Probleme:

- Die Logik des Automaten ist auf die verschiedenen View Controller verteilt. Dies vermindert die Übersichtlichkeit ohne weitere Vorteile zu bieten.
- Der Aufwand für eine grosse Anzahl Zustände ist sehr gross, da alle Views von Hand erstellt werden müssen.

In dieser Auflistung nicht berücksichtigt ist die Benachrichtigung des Assistenten darüber, dass nun ein Zustandsübergang ausgeführt werden soll. Diese Problematik besteht allerdings bei allen Ansätzen und kann unterschiedlich gelöst werden. Wir verzichten deshalb auf die Betrachtung dieses Aspekts.

Ein zweiter Lösungsansatz besteht in der Verwendung einer einzigen View. Wir stellen uns diese als eine lange Ansicht aller Schritte vor. Auf einem Smartphone ist allerdings immer nur ein Abschnitt der View sichtbar. Bei einem Zustandsübergang wird lediglich vertikal gescrollt, damit der korrekte Abschnitt sichtbar ist.

Gegenüber dem einfachen Ansatz bietet diese Lösung folgende Vorteile:

Es kann auf die Erstellung mehrerer einfacher Views inklusive ihrer View Controller verzichtet werden. An die Stelle des Container Controllers rückt ein einziger View Controller mit der langen View.

Da nur ein Controller verwendet wird, ist die gesamte Logik darin gekapselt. Leider entsteht bei diesem Ansatz auch ein neuer Nachteil:

- Während die Vorstellung einer langen View sehr einfach wirkt, ist ihre Umsetzung nicht trivial. Auf beiden grossen Smartphone-Plattformen wäre dies mit viel Handarbeit verbunden.

In unserem dritten Ansatz bauen wir auf dem State Pattern auf (Abb. 3). Die verschiedenen Zustände bieten ein einheitliches Interface gegen aussen an. Bei einem Zustandsübergang wird lediglich die implementierende State-Klasse ausgetauscht. In unserem Beispiel könnten wir folgendes State-Interface anbieten:

- + *getUIComponents()* liefert eine Liste der darzustellenden Komponenten;
- + *isStepComplete()* gibt an, ob der Schritt abgeschlossen ist;
- *customStatusInfo* sind benutzerdefinierte Informationen.

Wenn wir nochmals zum ersten Ansatz mit den einzelnen Views zurückgehen und das State Pattern anwenden, enden wir wiederum bei $n$+1 Controllern für $n$ Zustände. Die Controller für die Zustände müssen aber nun das oben definierte Interface implementieren. Die Views ersetzen wir durch eine neue View, welche beim jeweiligen State-Controller mit *getUIComponents()* abfragt, welche UI-Komponenten angezeigt werden sollen.

Die Methode *isStepComplete()* und die Statusinformationen sind auch in den beiden vorherigen Ansätzen vorhanden, wir haben sie hier ledig-

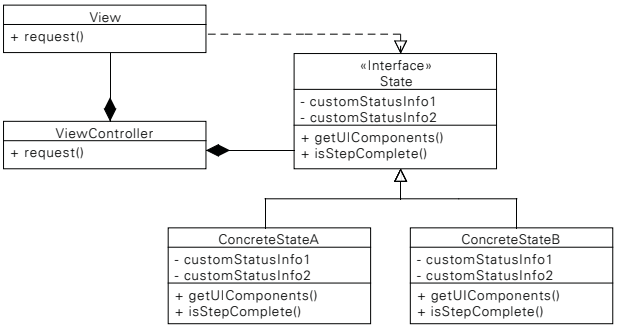


Abbildung 4: Klassendiagramm des implementierten State Patterns

```
// Header-File UITableViewDataSource
– tableView:cellForRowAtIndexPath:  // required method
– numberOfSectionsInTableView:
– tableView:numberOfRowsInSection:  // required method
– tableView:titleForHeaderInSection:

// Header-File UITableViewDelegate
– tableView:heightForRowAtIndexPath:
– tableView:didSelectRowAtIndexPath:
```

Listing 1a: Auszug aus dem Header-File von UITableViewDataSource. Das DataSource-Objekt ist für die Inhalte der TableView verantwortlich und bietet dazu unter anderem die Methode tableView:cellForRowAtIndexPath: an, welche die angegebene Zelle der Tabelle tableView zurückgibt.
Listing 1b: Die zwei wichtigsten Methoden aus dem Header von UITableViewDelegate. Das Delegate-Objekt ist für die Darstellung und die Interaktion verantwortlich. Es liefert unter anderem die Höhe für jede Zelle (tableView:heightForRowAtIndexPath:) und reagiert auf Benutzereingaben (tableView:didSelectRowAtIndexPath:)

lich formalisiert. Abbildung 4 zeigt das erweiterte Klassendiagramm.

Wie schlägt sich der dritte Ansatz im Vergleich zu den beiden anderen Ansätzen? Die Vorteile des dritten Ansatzes sind:

- Die Logik des Automaten ist zentral im Controller des Assistenten implementiert, womit die Implementation der Zustände einfach gehalten werden kann.
- Die Views lassen sich wiederverwerten (z. B. durch vordefinierte Components-Sets für *getUIComponents()*).

Wir handeln uns als direkte Konsequenz aber mindestens das Problem der erschwerten Testbarkeit der Benutzeroberfläche ein, da die Views erst zur Laufzeit erzeugt werden. Zudem bestehen auch bei dieser Lösung noch technische Herausforderungen:

- Wie kann eine solch flexible View implementiert werden?
- Wie kann die View bei einem Zustandsübergang neu gezeichnet werden?

Während die Gewichtung der einzelnen Aspekte der vorliegenden Lösungsansätze von Plattform zu Plattform und von Problemstellung zu Problemstellung variieren kann, erscheint uns die Lösung mit Hilfe des State Patterns auf Grund der vielen ähnlichen Zustände am sinnvollsten.

Im nächsten Abschnitt werden wir genauer auf die Umsetzung dieser Variante auf iOS eingehen. iOS bietet Möglichkeiten, welche die beiden zuvor genannten Herausforderungen elegant lösen lässt. Im Grundsatz kann eine solche Lösung aber auch auf Android oder Windows Phone angewandt werden.

```
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  return 1;
}
#pragma mark - Table view data source
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
  return 1;
}
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
  return @”Request Code”;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  // #1
  PairingRequestCodeCell *requestCodeCell = (PairingRequestCodeCell *)
[tableView dequeueReusableCellWithIdentifier:@"RequestCodeCell" forIndexPath:indexPath];
  // Konfiguration der Zelle
  return requestCodeCell;
}

#pragma mark - Table view delegate
- (void)tableView:(UITableView*)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
  // Wird hier nicht benötigt.
}
```

Listing 2: Auszug aus einem View Controller State


Abbildung 5: Der erste Zustand in unserem Pairing-Assistenten. Der Benutzer wird aufgefordert einen Pairing-Code einzugeben.

```
@interface PairingViewController :
  UITableViewController

@property (nonatomic, strong)Pairing
  *pairing;

@property (nonatomic)BOOL isEstablishing;

@property (nonatomic, strong)
  PairingViewControllerState *state;

@end
```

Listing 3: Header des Container View Controllers

Als Bespiel soll uns die entwickelte App für Störmeldesysteme dienen. Wir werden aber nicht näher auf ihre Funktionalität eingehen.

Der Einsatz einer *TableView* für ein beliebiges UI ist keine derart abenteuerliche Wahl und wird in vielen iOS-Apps verwendet. Wir verweisen auf die Mail-App für den Posteingang, die Kalender-App für die Stunden in der Tagesansicht oder die Nachrichten-App für den Verlauf der Konversation, um nur die offensichtlichsten Beispiele zu nennen. Der Benutzer nimmt diese Table Views unter Umständen gar nicht als Table Views wahr.

**Stateful UITableView**

Wir beginnen gleich bei der wichtigsten Komponente unseres Assistenten: der *UITableView*. Eine *UITableView* ist eine Liste von Einträgen, also eine einspaltige Tabelle.[2] Eine *UITableView* verfügt über zwei wichtige Referenzen: eine auf eine *DataSource* und eine auf ein *Delegate*. Beide Referenzen zeigen auf Objekte, welcher der *TableView* zur Laufzeit Verhalten hinzufügen.

Es dürfte nun klar werden, weshalb sich eine *TableView* gut zur Darstellung eines Assistenten eignet. Einmal erzeugt bezieht sie ihre Inhalte aus der *DataSource* mit Hilfe der Methode *tableView:cellForRowAtIndexPath:* aus Listing 1a und verhält sich wie in der *Delegate*-Methode *tableView:didSelectRowAtIndexPath:* implementiert (Listing 1b). Es reicht also aus die Implementierenden Klassen der beiden Interfaces[3] *UITableViewDelegate* und *UITableViewDataSource* auszutauschen um Inhalt und Verhalten der *TableView* komplett zu ändern. Folglich sind es nun auch diese beiden Klassen, welche wir in verschiedene Zustände implementieren möchten.[4]

Wir betrachten zur Veranschaulichung der Verwendung von *Delegates* und *DataSources* einen Auszug aus dem State, welcher einen Pairing-Code entgegennimmt. Wie in Abbildung 5 ersichtlich besteht die *TableView* aus einer *Section* mit einer einzigen Zelle. In Listing 2 ist der entsprechende Code ersichtlich.

Wie in Listing 2 ersichtlich, ist die Implementierung eines View Controller States sehr übersichtlich. Wir benötigen lediglich drei Methoden aus *UITableViewDataSource* und eine aus *UITableViewDelegate*. Auch wenn komplexere Schritte im Assistenten mehr Aufwand erfordern, so bleibt die Grundstruktur dennoch bei jedem State dieselbe. Spezielle Erwähnung verdient der Code-Ausschnitt #1: dort wird eine mit dem Interface Builder grafisch erstellte Zelle erzeugt. Somit erfüllt diese Lösung auch eine weitere unserer Anforderungen, die Wiederverwendbarkeit von UI-Elementen. Dieses Vorgehen schont ausserdem die beschränkten Systemressourcen des Smartphones, da von einem Zellentyp nur so viele Instanzen erzeugt werden, wie angezeigt werden können. Scrollt der Benutzer nach unten, werden die nicht mehr sichtbaren Objekte rezykliert.

Als Container für unsere View dient ein View Controller. Dieser muss in der Lage sein, den Automaten auf Grund der Zustandsinformationen zu durchlaufen und dabei die korrekten Informationen anzuzeigen. In Listing 3 ist der Header eines Container Controllers angegeben. Wie die erste Zeile

```
@interface PairingViewController :
  UITableViewController
```

bereits zeigt, ist der Container View Controller von *UITableViewController* abgeleitet und erbt somit auch dessen Properties dataSource und delegate. Bei state handelt es sich um eine Referenz auf den aktuell gültigen Zustand. *isPairing* und *pairing* sind die angesprochenen Zustandsinformationen. Sie sind in unserem Fall ausreichend, um in jedem Fall den korrekten nächsten Zustand anzusteuern.

2 Ihr Name ist eine Anlehnung an ihren grossen Bruder NSTableView auf OS X, welche mit mehreren Spalten umgehen kann.

3 Bei UITableViewDataSource und UITableViewDelegate handelt es sich genau genommen um Objective-C-Protokolle. Diese Unterscheidung ist für uns aber nicht relevant, weshalb wir bei der allgemein besser bekannten Bezeichnung Interface bleiben[TV].

4 Die hier vorgestellten Ideen lassen sich auch auf eine UICollectionView anwenden.

```
- (void)setState:(PairingViewControllerState *)state {
  _state = state; // _state bezeichnet das Property state aus dem Header
  // #1
  self.tableView.delegate = _state;
  self.tableView.dataSource = _state;
  [self.tableView reloadData];
}
```

Listing 4: Die Methode setState: führt eine Zustandsänderung aus.

Um den aktuellen Zustand zu ändern, ist es ausreichend das Delegate und die DataSource der TableView zu aktualisieren und die TableView zu aktualisieren. Listing 4 zeigt die Implementierung des Setters von state in der Klasse des Container Controllers.

Wir haben nun fast alle Elemente zusammen. Es bleibt die Frage zu klären, wie der Container erfährt, dass er eine Zustandsänderung ausführen soll, das heisst, dass er die Methode setState: (Listing 4) ausführen soll. Dies lässt sich in Objective-C beispielsweise mittels Key-Value-Observing [KVO] lösen. Dazu ergänzen wir die Methode *setState:* an der Stelle #1 mit dem folgenden Aufruf.

```
[_state addObserver:self
  forKeyPath:@“stateComplete“
  options:0 context:NULL]
```

Hier senden wir dem aktuellen Zustand *_state* die Message *addObserver:forKeyPath:options:context:* mit den benannten Parametern *forKeyPath*, *options* und *context*. Als Argumente geben wir eine Referenz auf uns selbst (*self*) und den zu observierenden Property-Name als String an. Wir benötigen keine Optionen und keinen unterscheidbaren Kontext, weshalb wir diese Parameter nicht setzen.

Nun wird der Container bei jeder Änderung des Property *stateComplete* benachrichtigt und kann, falls der neue Wert *YES* ist, die Zustandsinformationen auswerten und dann die Methode *setState:* mit dem entsprechenden Zustand aufrufen.

Somit haben wir alle Anforderungen der Problemstellung gelöst, ohne die Nachteile der anderen Ansätze in Kauf zu nehmen. Konkret haben wir eine einfache Methode gesehen, um Zustandsänderungen auszuführen, wir können grosse Teile des UI an verschiedenen Stellen einsetzen und dennoch sind alle implementierten Klassen so knapp, dass die Übersicht nicht verloren geht. Des Weiteren können wir neue Zustände hinzufügen und müssen dabei lediglich die Zustandslogik entsprechend erweitern.

**Zusammenfassung**

Wir haben nun einen kleinen Einblick in die Entwicklung eines zustandsbasierten UI auf dem iPhone erhalten. Wir haben weder Bibliotheken von Drittherstellern, noch sonderlich fortgeschrittene Funktionen der iOS-Plattform eingesetzt. Die verwendeten Funktionen werden in den meisten iOS-Apps angewendet. Dies gilt auch für Key Value Observing, obwohl einigen Entwicklern vermutlich nicht bewusst ist, dass dieses Prinzip unter der Haube für viele Annehmlichkeiten der Plattform verantwortlich ist.

**Referenzen**

[KVO] Key-Value Observing Programming Guide/ Introduction to Key-Value Observing Programming Guide: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html, 07.11.2013.

[TV] Table View Programming Guide for iOS/ About Table Views in iOS Apps: http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/tableview_iphone/AboutTableViewsiPhone/AboutTableViewsiPhone.html, 07.11.2013.