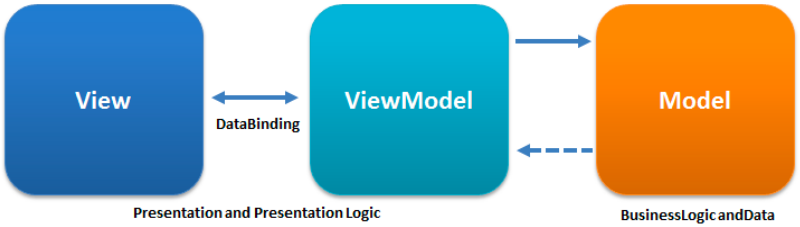
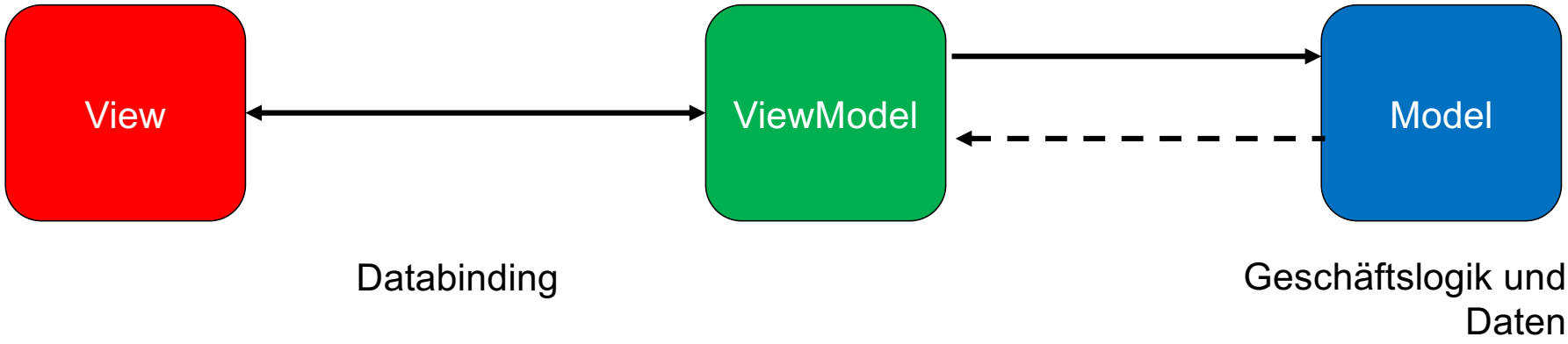
Fachhochschule
Nordwestschweiz

iOS Modul
MVVM und Swift Type System



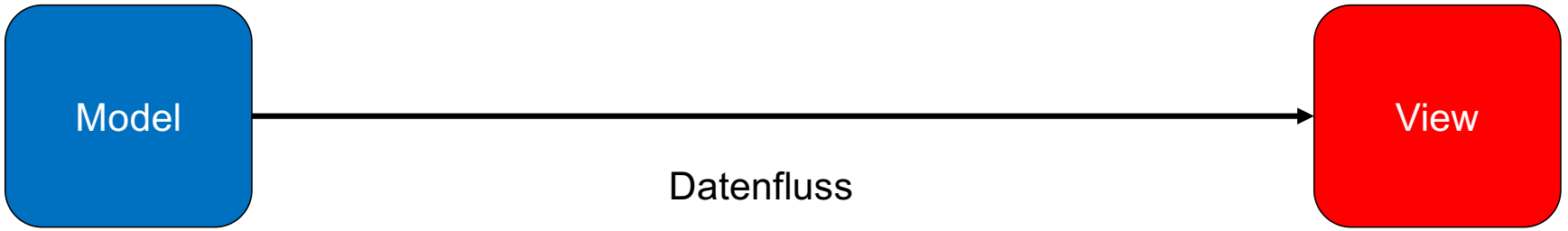
Model-View-ViewModel

- Entwurfsmuster und eine Variante des MVC-Patterns (Model-View-Controller)
- MVC wird überwiegend in UIKit (Vorgänger von SwiftUI) verwendet
- MVVM ist das Default-Pattern für SwiftUI und unterstützt das Konzept von “Reactive» User-Interfaces



Richtung: Model → View

MVVM

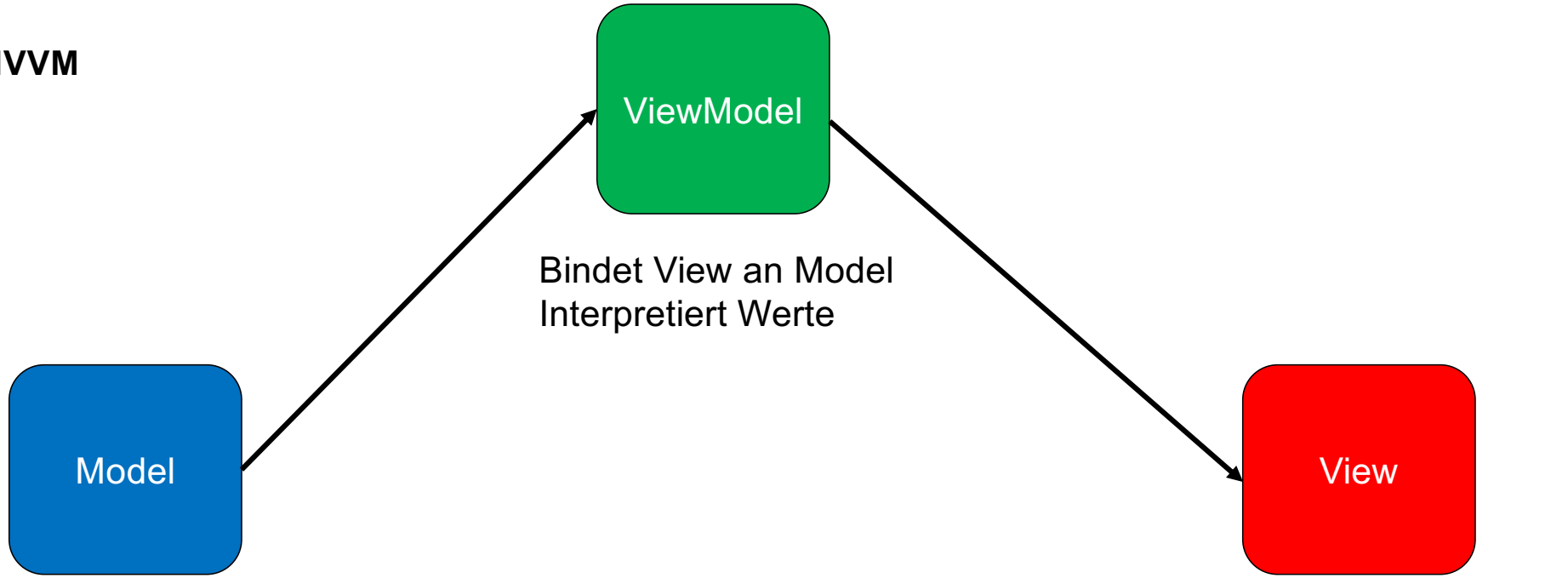


UI Unabhängig
Daten und Logik
„Die Wahrheit“

Spiegelt das Model
Stateless
Deklarativ
Reaktiv

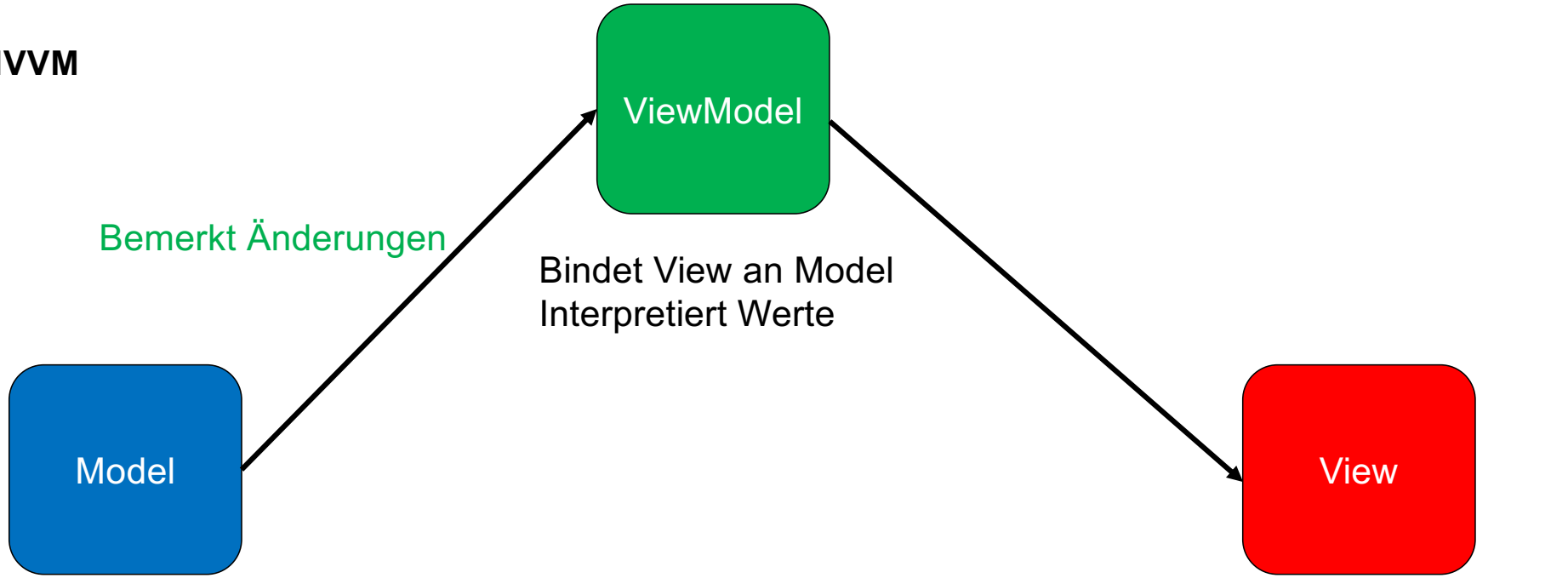
Richtung: Model → View

MVVM



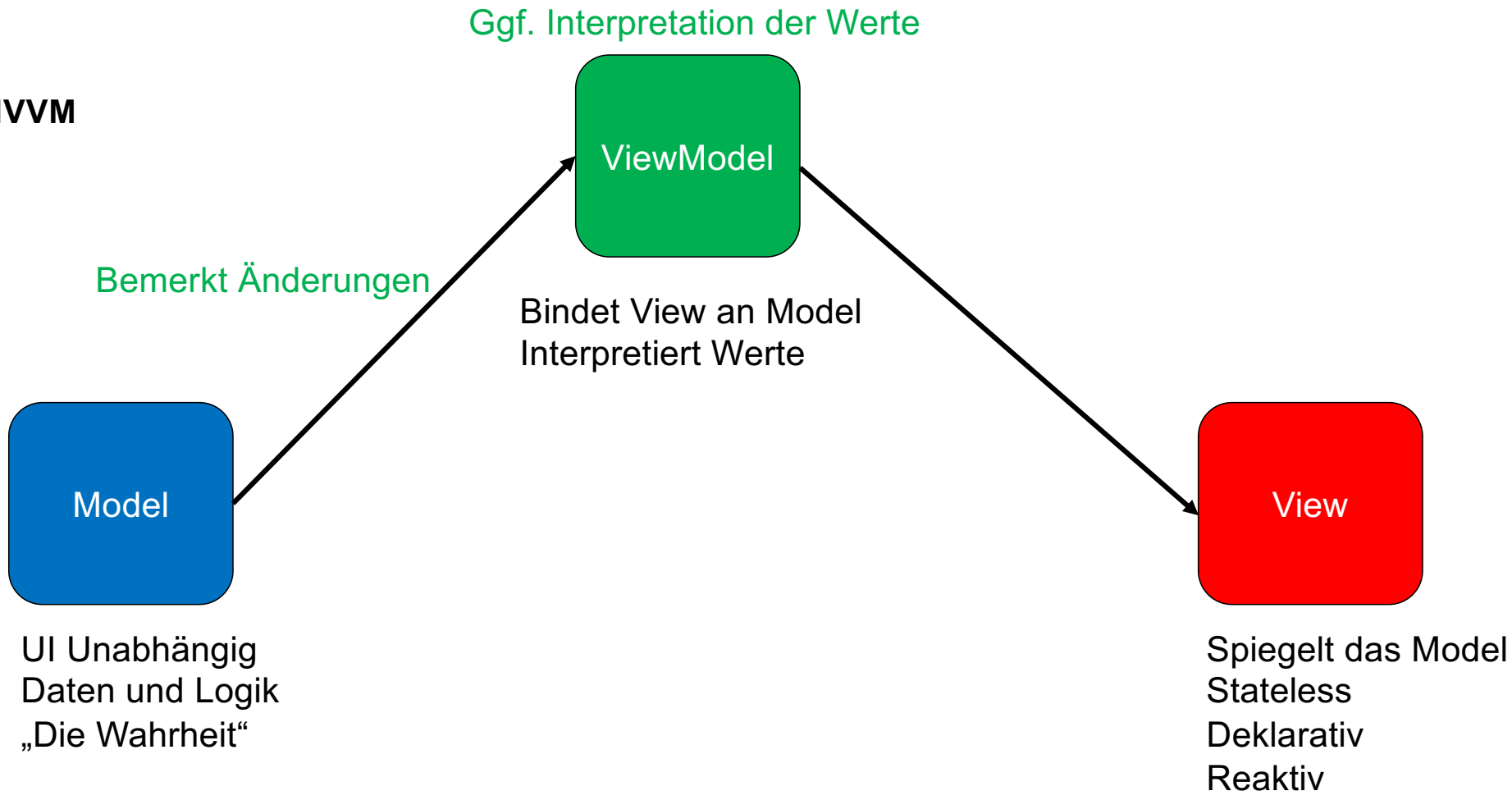
Richtung: Model → View

MVVM



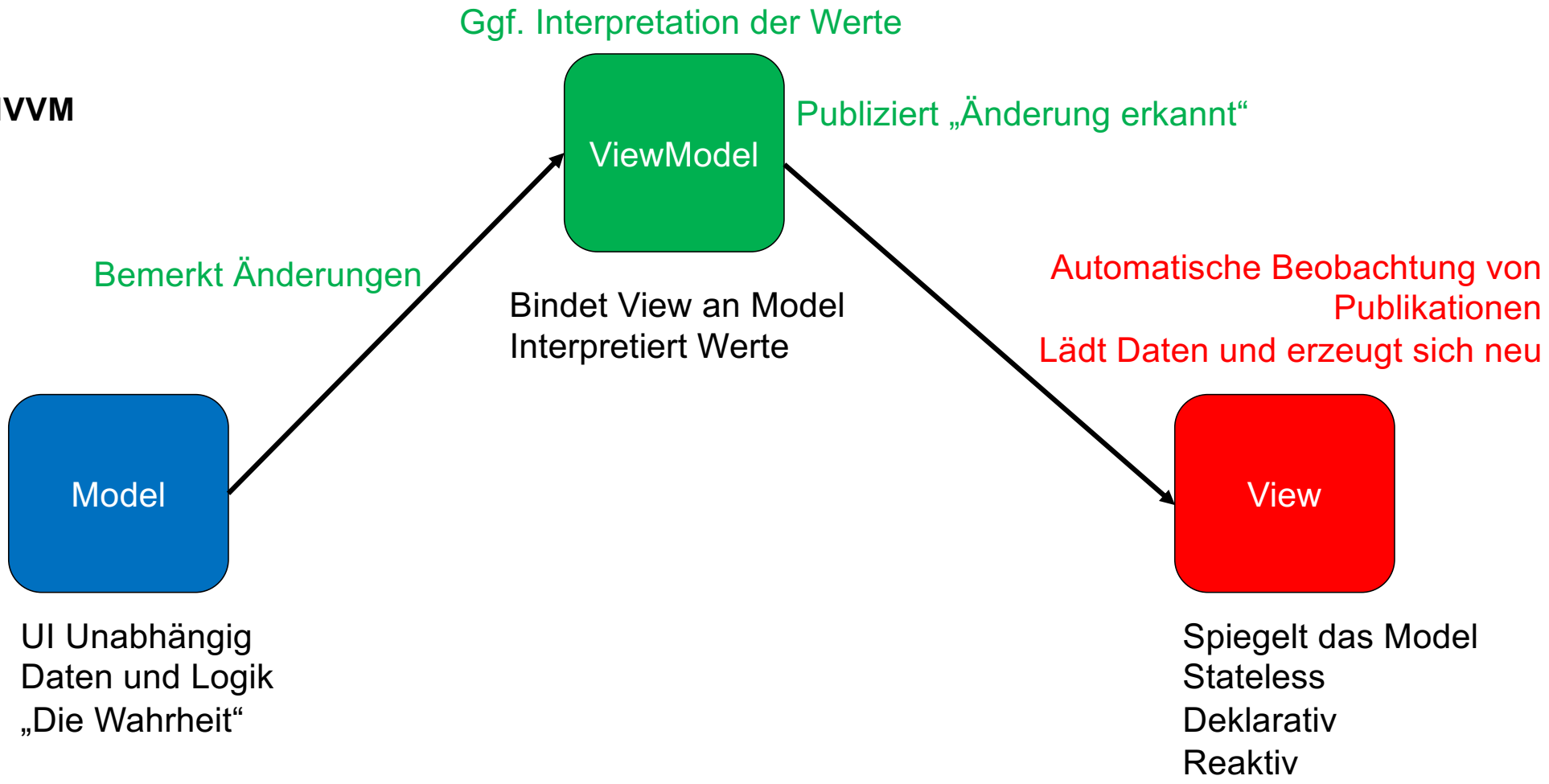
Richtung: Model → View

MVVM



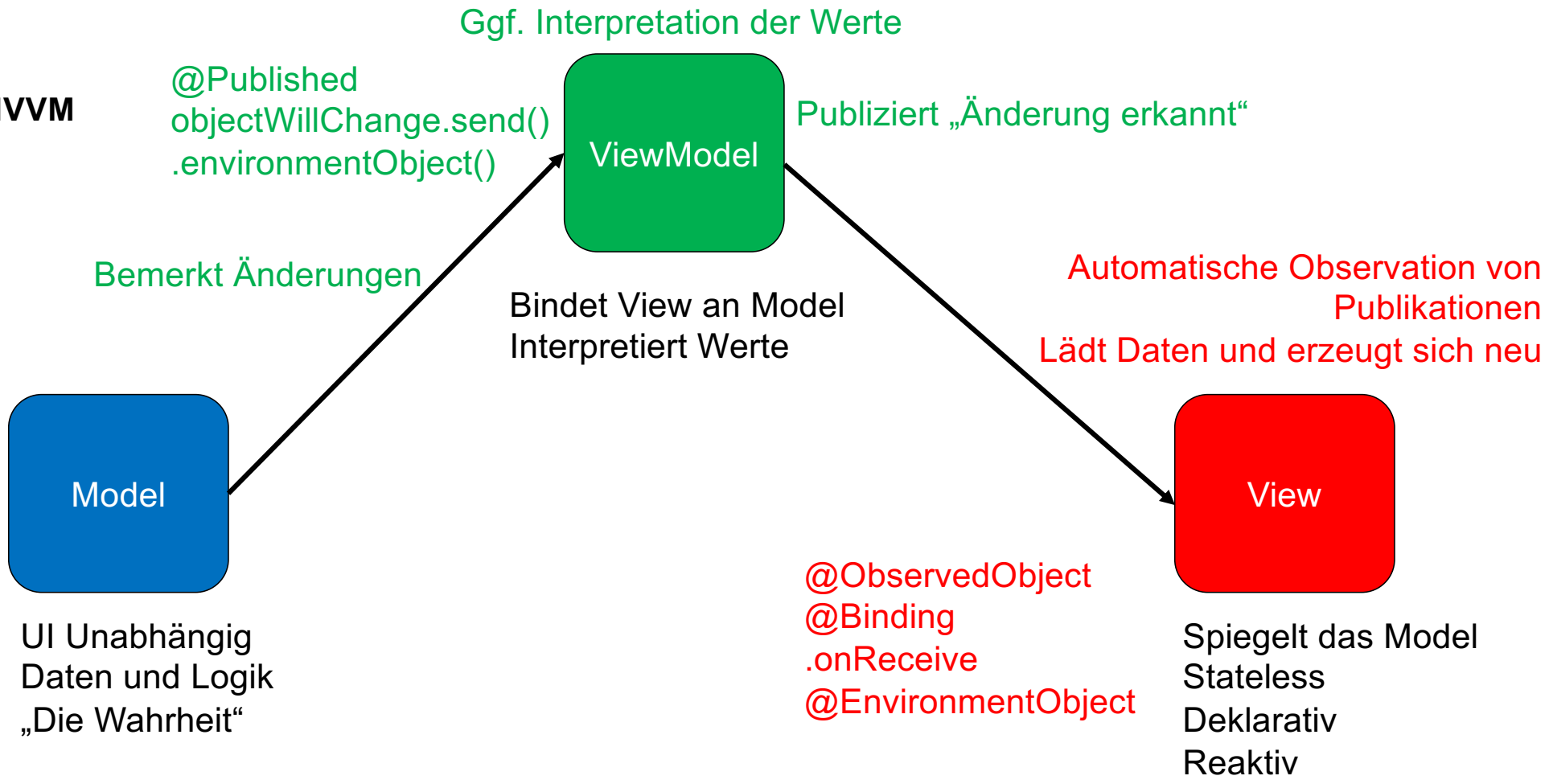
Richtung: Model → View

MVVM



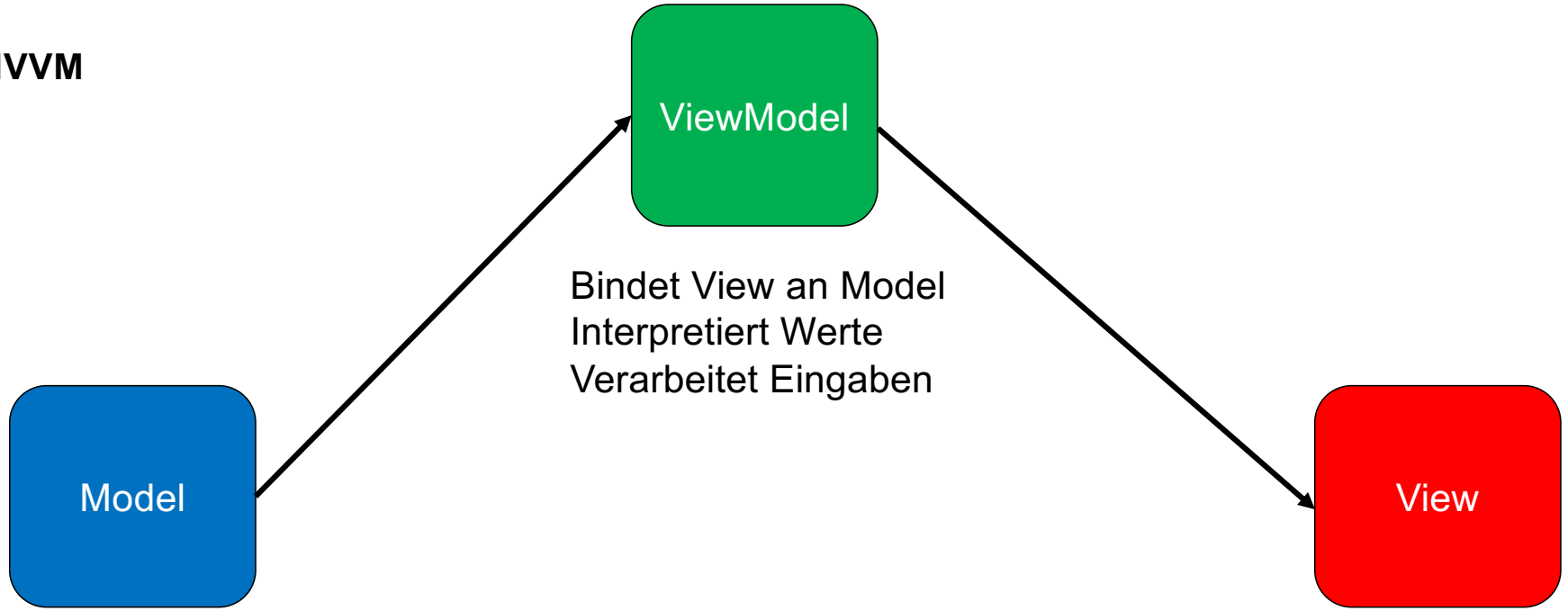
Richtung: Model → View

MVVM



Richtung: View → Model

MVVM



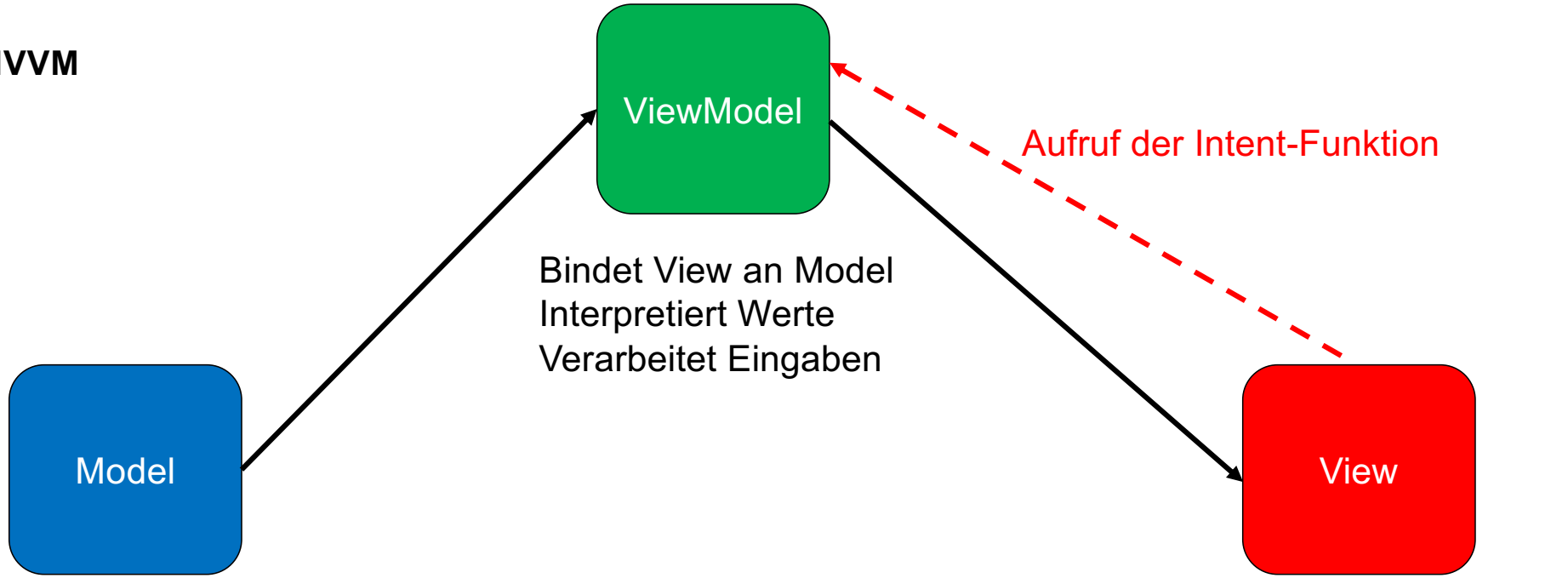
Bindet View an Model
Interpretiert Werte
Verarbeitet Eingaben

UI Unabhängig
Daten und Logik
„Die Wahrheit“

Spiegelt das Model
Stateless
Deklarativ
Reaktiv

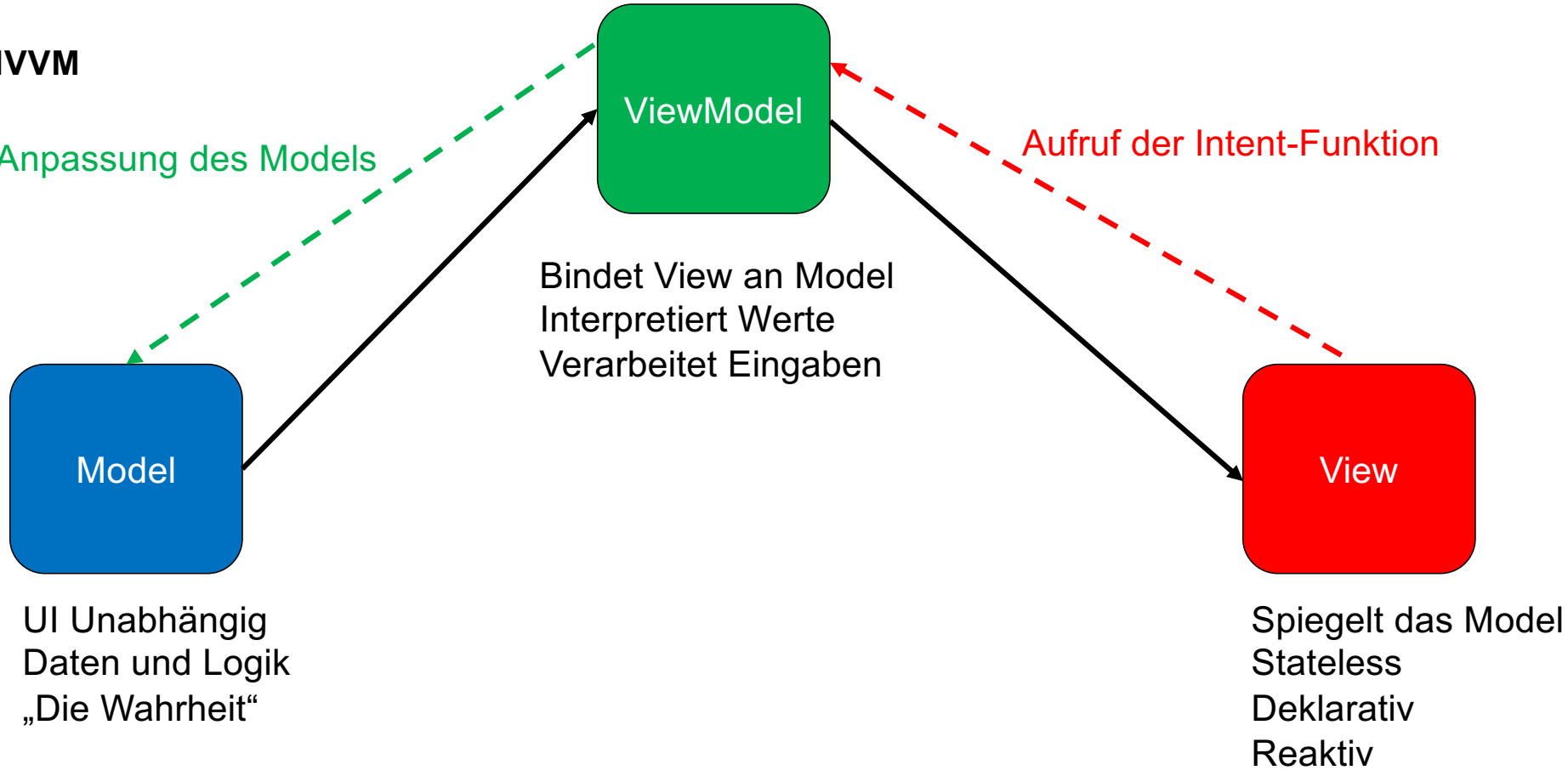
Richtung: View → Model

MVVM



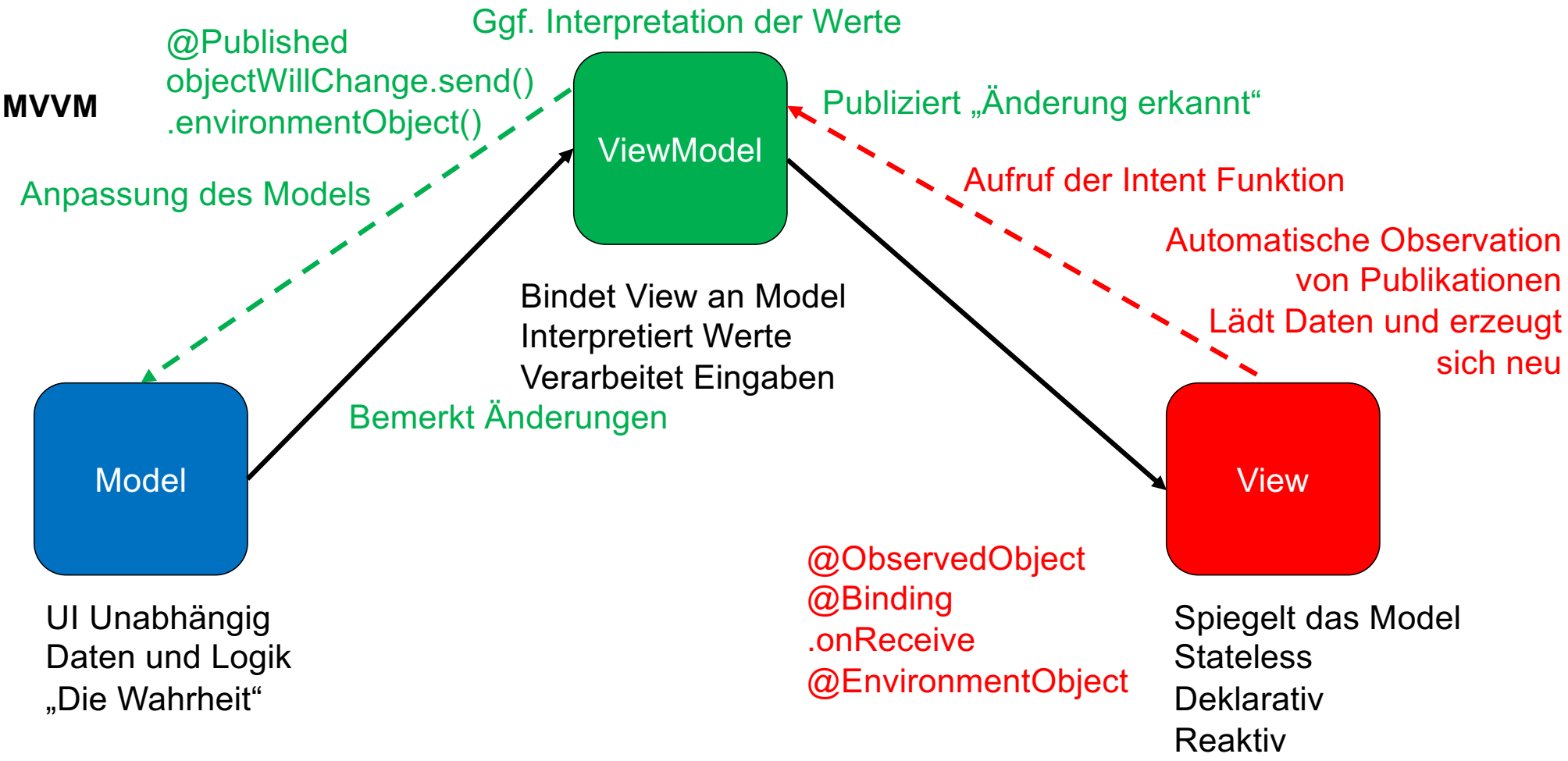
Richtung: View → Model

MVVM



Richtung: Model → View

MVVM



Typensystem von Swift

- struct und class haben
  - Fast die genau gleiche Syntax
  - Gespeicherte Variablen, d.h. im Memory gespeicherte Werte

```
var isFaceUp: Bool
```

Typensystem von Swift

- struct und class haben
  - Fast die genau gleiche Syntax
  - Gespeicherte Variablen, d.h. im Memory gespeicherte Werte
  - Berechnete Variablen, d.h. Werte die durch Codeauswertung berechnet wurden

```
var body: some View{
      return Text(„Hello World“)
}
```

Typensystem von Swift

- struct und class haben
  - Fast die genau gleiche Syntax
  - Gespeicherte Variablen, d.h. im Memory gespeicherte Werte
  - Berechnete Variablen, d.h. Werte die durch Codeauswertung berechnet wurden
  - Konstanten, d.h. Variablen, deren Wert sich nicht ändert

```
let defaultColor = Color.orange
…
CardView().forgroundColor(defaultColor)
```

Typensystem von Swift

- struct und class haben
  - Fast die genau gleiche Syntax
  - Gespeicherte Variablen, d.h. im Memory gespeicherte Werte
  - Berechnete Variablen, d.h. Werte die durch Codeauswertung berechnet wurden
  - Konstanten, d.h. Variablen, deren Wert sich nicht ändert
  - Funktionen

```
func multiply(operand: int, by: Int)->Int{
      return operand * by
}

multiply(operand:5, by: 6)
```

Typensystem von Swift

- struct und class haben
  - Fast die genau gleiche Syntax
  - Gespeicherte Variablen, d.h. im Memory gespeicherte Werte
  - Berechnete Variablen, d.h. Werte die durch Codeauswertung berechnet wurden
  - Konstanten, d.h. Variablen, deren Wert sich nicht ändert
  - Konstruktoren, d.h. Funktionen, die beim Erzeugen des Objektes ausgeführt werden

```
struct MemoryGame{
      init(numberOfPairsOfCards: Int){
            //Erzeugung MemoryGame mit Anzahl von Paaren
      }
}
```

Unterschiede zwischen struct und class

struct	class
Value type	Reference type
Bei Zuweisung oder Weitergabe kopiert	Verwendung von Pointern
Copy on write	Automatisches Reference-Counting
Keine Vererbung	(Einfache) Vererbung
Standard-Intitialisierer intitialisert alle Variablen	Standard-Intitialisierer intitialisert keine Variable
Mutabilität muss explizit ausgewiesen werden	Immer mutierbar
Standard Datenstruktur	Verwendung in spezifischen Fällen
Der Grossteil der Objekte in SwiftUI-Projekten sind Structs	Das ViewModel ist immer eine Klasse

Generics

- Manchmal spielt der Datentyp keine Rolle „Don‘t Care“
  - D.h. der Typ einer Datenstruktur ist unbekannt und ist auch egal
  - Jedoch ist Swift stark typisiert, d.h. es gibt keine Variablen ohne zugeordneten Datentyp
  - Problem: Wie spezifiziert man den Typ von etwas, das unbekannt und auch egal ist?
  - Lösung: Durch Verwendung des „Don‘t Care“-Datentyps: Generics

Generics: Beispiel Array

- Array ist ein gutes Beispiel für Generics:
  - Ein Array enthält Elemente und Funktionen, wobei der konkrete Datentyp keine Rolle spielt
  - In der Implementierung von Array braucht es jedoch Variaben, mit denen Operationen durchgeführt werden, die durch Swift typisiert sein müssen.
  - Auch als Schnittstelle nach Aussen müssen Typen angegeben werden, zum Beispiel beim Hinzufügen von Elementen

```
struct Array<Element>{
      …
      func append(_ element: Element){…}
}
```

Generics: Beispiel Array

```
struct Array<Element>{
      …
      func append(_ element: Element){…}
}
```

- Der Typ des Arguments ist Element, ein Generic
- Die Implementierung von Array weiss nichts über dieses Argument und es spielt auch keine Rolle.
- Element ist ein Platzhalter und kann ein struct, class oder protocol sein
- Array wird wie folgt verwendet. Es ist zu sehen, wie der Typ festgelegt wird:

```
var a = Array<Int>()
a.append(5)
```

Generics: Beispiel Array

```
struct Array<Element>{
      …
      func append(_ element: Element){…}
}
```

- Der Typ des Arrays wird durch die < > Schreibweise festgelegt. D.h. im Beispiele unten akzeptiert der Array nur Integer-Werte

```
var a = Array<Int>()
```

- Es ist möglich unterschiedliche Generics paralell zu verwenden, z.B.

```
 <Element, Foo, Bar >
```

- Andere Sprachen wie Java kennen ebenfalls Generics. In Swift werden diese durch `protocols` deutlich mächtiger

Funktionen als Typ

- Funktionen sind ebenfalls Typen
  - Variablen lassen sich vom Typ `function` deklarieren
  - Die Syntax enthält die Typen der Argumente und den Rückgabetyp

```
(Int, Int) -> Bool // Nimmt 2 Int retourniert Bool
(Double) -> Void // nimmt Double, retourniert nichts
() -> Array<String> // Nimmt keine Argumente, gibt String-Array zurück
()-> Void // Hat keine Argumente und gibt nichts zurück
var foo: (Double) -> Void //Typ von foo: Funktion, die Double nimmt und
                          nichts retourniert
func doSomething(what: () -> Bool) //Typ von what: Funktion die kein
                                   Argument nimmt und Bool retourniert
```

Funktionen als Typ

```
var operation: (Double)->Double //Typ von operation: Funktion die
                                  Double nimmt und Double retourniert
func square(operand: Double)>Double{
      return operand * operand
}
operation = square //Zuweisung eines Wertes an Variable operation
let result1 = opertion(4) // result1 hat Wert 16
operation = sqrt // sqrt ist eine interne Funktion mit passender Signatur
let result2 = operation(4) //result2 hat Wert 2
```

- Wenn Funktionstypen ausgeführt werden, werden keine Bezeichner für das Argument verwendet
- Weitere Beispiele werden in der Demo gezeigt

Funktionen als Typ

- Closures
  - Häufig werden Funktionen herumgereicht, die eingereiht („inlining“) werden
  - Solche eingereihten Funktionen werden als Closure bezeichnet und werden in der Demo gezeigt
  - SwiftUI basiert stark auf Funktionaler Programmierung und Funktionen als Typen relativ oft verwendet.

MVVM und Typen in der Demo

- Inhalte der Demo:
  - MVVM Pattern
  - Eigene init-Funktion
  - Generics in der Model-Implementierung
  - Funktion als Typ im Model
  - Class im ViewModel
  - Rückkanal „Intent“ im MVVM
  - Reaktives UI durch das MVVM-Entwurfsmuster