Universität Augsburg

Fakultät für Angewandte Informatik
Lehrprofessur für Informatik
Prof. Dr. Robert Lorenz

SS 2012

13.06.2012

Betreutes Programmieren Vorlesung Informatik II, Blatt 8

Musterlösung: ca. 137 Codezeilen*

Beschreibung

Am Bildschirm werden Farben durch die additive Mischung von rot, grün und blau dargestellt. Alle drei Farben zusammen ergeben weiß. Um nachzuvollziehen, wie alle anderen Farben gemischt werden, hilft das *Farbenbeobachterspiel*. Es erlaubt, die drei Farben einzeln zu beeinflussen und das Ergebnis in verschiedenen Repräsentationen am Bildschirm zu sehen (s. Abbildung 1).

Aufgabenstellung

Erstellen Sie eine Klasse **`Farbe`**, die eine konkrete Farbe durch die Werte für rot, grün und blau – jeweils eine ganze Zahl im Bereich zwischen 0 und 255 – repräsentiert und von verschiedenen Fenstern beobachtet werden kann. Das Hauptfenster der Anwendung enthält drei Schieberegler (**`Scrollbar`**), mit denen die Farbwerte verändert werden können. Die beobachtenden Fenster stellen die aktuelle Farbe auf verschiedene Weisen dar:

- Ein Fenster hat die aktuelle Farbe als Hintergrundfarbe.
- Ein Fenster hat das Komplement der aktuellen Farbe als Hintergrundfarbe.
- Ein Fenster zeigt die Zahlenwerte für rot, grün und blau der aktuellen Farbe an.
- Ein Fenster zeigt die Zahlenwerte für rot, grün und blau des Komplements der aktuellen Farbe an.

Abbildung 1: Alle Fenster des Farbenbeobachterspiels.

Die Vorgabe zur Implementierung können Sie dem UML Klassendiagramm aus Abbildung 2 entnehmen. Dabei ist **`ControllerWindow`** das Hauptfenster der Anwendung, **`ColorWindow`** entspricht den Fenstern mit der

*__Nicht gezählt__: Leerzeilen, Kommentare, `import`s. __In eigenen Zeilen__: `{` und `}` (ca. 40x), jede Anweisung, außer bei Deklarationen gleichen Typs (z. B. `int a, b, c = 0`, ...) oder bei einer einzelnen Anweisung nach einem `if` oder einem `else`. __Nicht benutzt__: `?:`-Operator.

Hintergrundfarbe und **`StringWindow`** den beiden anderen Fenstern. Ein privates Attribut **`komplement`** zeigt jeweils an, ob das Fenster die Farbe oder deren Komplement darstellt.

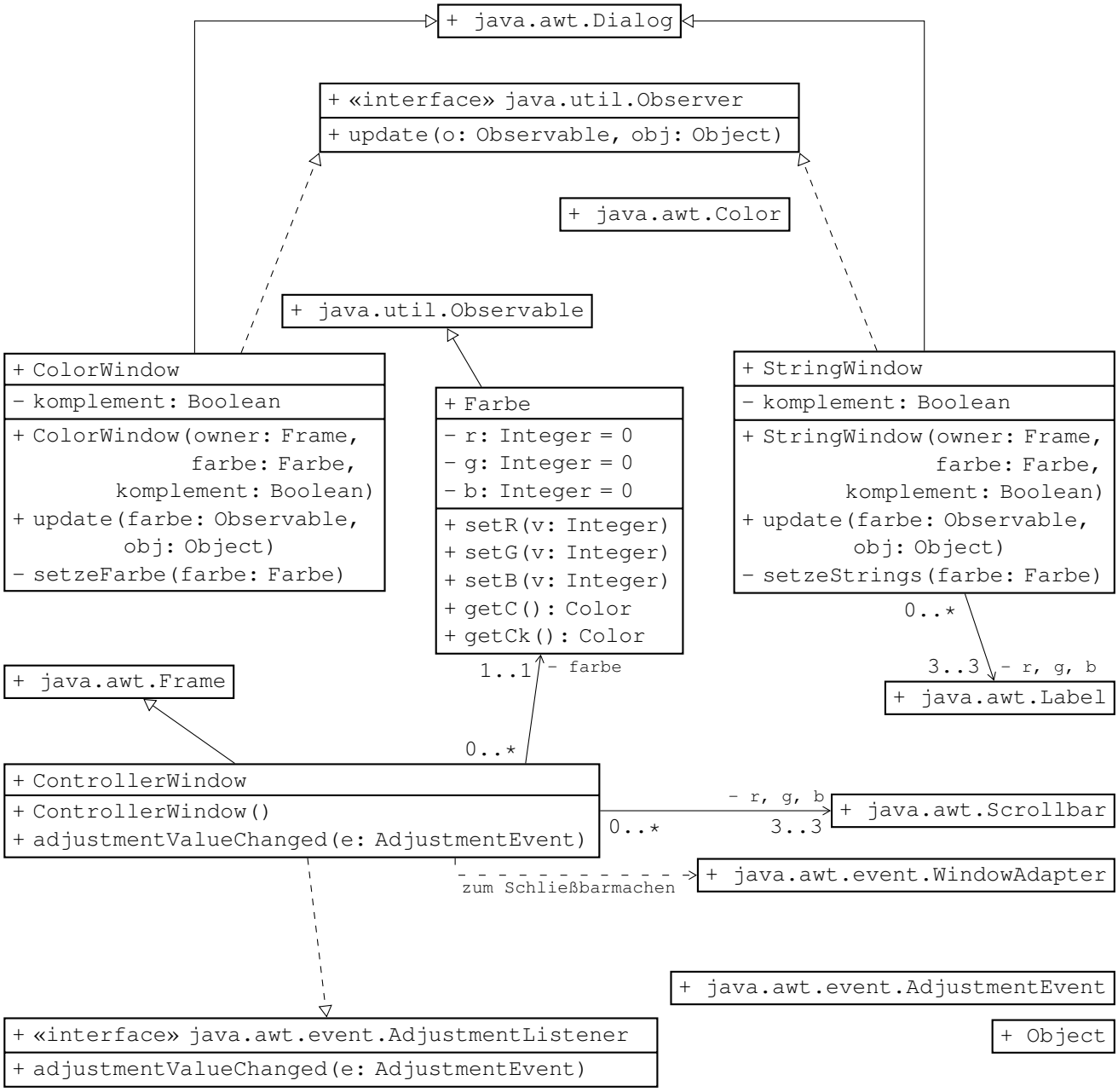


Abbildung 2: UML Klassendiagramm des Farbenbeobachterspiels.

Die Methode **`getC()`** der Klasse **`Farbe`** gibt ein Objekt der Klasse **`Color`** zurück, das die Farbe repräsentiert, die aus den einzelnen Farbwerten des **`Farbe`**-Objektes resultiert. Der Getter, der mit einem **`k`** endet, gibt das Komplement der Farbe zurück.

Die Methoden **`setzeFarbe()`** und **`setzeStrings()`** übernehmen die eigentliche Darstellung der jeweiligen Farbe in Abhängigkeit des Attributes **`komplement`** und werden vom jeweiligen Konstruktor und der jeweiligen **`update()`**-Methode aufgerufen.

Als weitere Vorgabe erhalten Sie das Sequenzdiagramm des Konstruktors der Klasse **`ControllerWindow`** in Abbildung 3 auf der nächsten Seite. Die genaue Implementierung des ⟨*überschriebenen WindowAdapter*⟩ fehlt im Diagramm, kann aber in der Angabe zu Blatt 5 des Betreuten Programmieren nachgelesen werden.

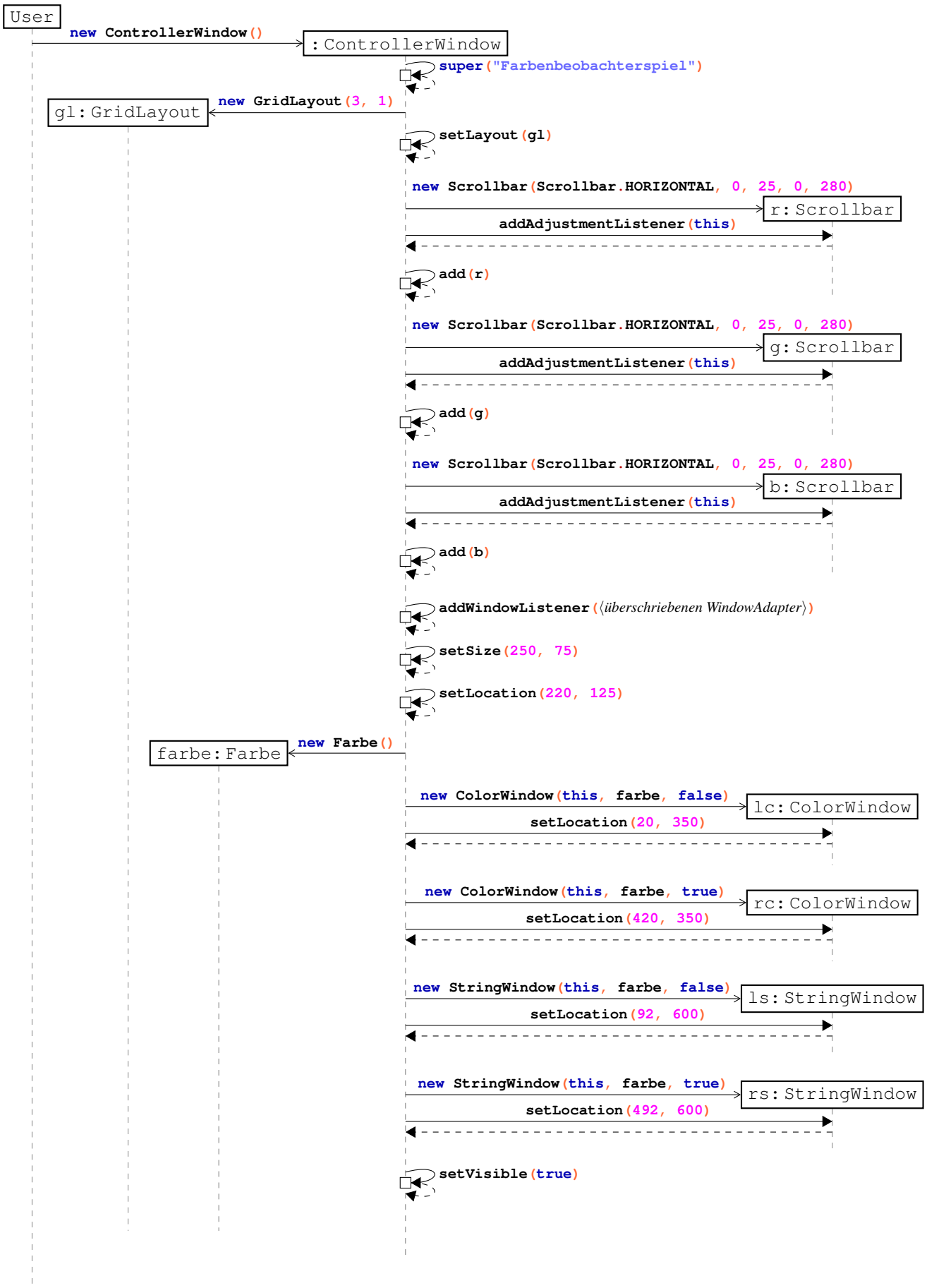


Abbildung 3: UML Sequenzdiagramm von **`ControllerWindow()`**.

Hinweise

Implementieren Sie die Klassen **`Farbe`**, **`ControllerWindow`**, **`ColorWindow`** und **`StringWindow`** und fügen Sie der Klasse **`ControllerWindow`** eine **`main()`**-Funktion hinzu. Beachten Sie, dass **`Farbe`** die zu beobachtende Datenklasse ist, und **`ColorWindow`** und **`StringWindow`** Beobachterklassen darstellen.

Entnehmen Sie die Grundgerüste der Klassen dem UML Klassendiagramm aus Abbildung 2 auf Seite 2, sowie die Implementierung des Konstruktors **`ControllerWindow()`** dem Sequenzdiagramm aus Abbildung 3 auf der vorherigen Seite.

Für weitergehende Erklärungen zum Beobachter-Muster oder zur Farbdarstellung am Bildschirm lesen Sie den Anhang ab Seite 5. Benötigen Sie weitergehende Erklärungen zu Schnittstellen, Vererbung, Konstruktoren, Polymorphismus oder der Implementierung von Fenstern und Dialogen, dann sehen Sie sich die Musterlösungen des Betreuten Programmieren Blatt 3, 4 und 5 an.

Weitergehende Erklärungen

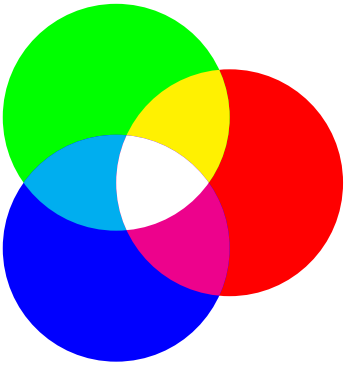
C.1 Farbdarstellung am Bildschirm

Die Darstellung von Inhalten am Bildschirm wird über sog. *Pixel* bewerkstelligt. Ein Pixel ist dabei ein Bildpunkt des Bildschirms. Die *Auflösung* gibt an, wieviele Punkte dargestellt werden. Eine typische Auflösung bei Laptops ist bspw. 1280 auf 800, was bedeutet, dass 1280 Pixel in der Breite und 800 Pixel in der Höhe den Bildschirminhalt auf die Mattscheibe zaubern. Das sind insgesamt $1280 \cdot 800 = 1\,024\,000$ Pixel (1 Megapixel).

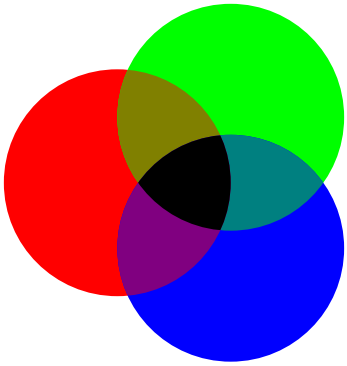
Ein einzelnes Pixel ensteht aus der Mischung der drei Farben Rot, Grün und Blau. Dabei handelt es sich um eine *additive* Mischung im Gegensatz zur *subtraktiven* Mischung, wie man sie von Wasserfarben her kennt. Rot und grün bei Wasserfarben ergibt braun, am Bildschirm jedoch gelb, weil sich das rote Licht und das grüne Licht zusammentun und dann in der gelben Wellenlänge strahlen.

Bei roter Wasserfarbe wird nur das rote Licht reflektiert und alle anderen Wellenlängen werden absorbiert und bei grüner Wasserfarbe entsprechend das grüne Licht reflektiert und alle anderen Wellenlängen absorbiert. Werden beide Farben gemischt, dann wird zwar rotes und grünes Licht reflektiert, aber eben auch ein bestimmter Anteil der beiden Wellenlängen absorbiert, sodass die entstandene Mischfarbe zu braun wird. Braun erhält man auch am Bildschirm aus rot und grün, wenn man deren Intensität jeweils halbiert und damit die Absorption simuliert.

In Abbildung 4 sind die Unterschiede in den beiden Mischarten dargestellt. In den mittleren Bereichen kann man schön erkennen, wie sich in Abbildung 4a alle drei Farben zu weiß addieren und in Abbildung 4b alle drei Farben zu schwarz subtrahieren.



(a) additiv



(b) subtraktiv

Abbildung 4: Die beiden Farbmischungen.

Typischerweise wird nun jedes Pixel am Bildschirm durch drei Bytes kodiert, wobei je ein Byte für rot, grün und blau bereitsteht. Damit sind für jeden Farbwert jeweils Werte zwischen 0 und 255 möglich und so insgesamt $256^3 = 16\,777\,216$ Farben darstellbar.

Java stellt im `awt`-Paket eine Klasse **`Color`** bereit, die z. B. mit **`Color.BLACK`** initialisiert werden kann. Die Klasse stellt Methoden **`getRed()`**, **`getGreen()`** und **`getBlue()`** zur Verfügung, mit denen die einzelnen Farbwerte abgefragt werden können. Leider existieren keine entsprechenden Setter, sodass nur der Ausweg der Neuerzeugung über einen Konstruktor, der drei Farbwerte erwartet, bleibt, falls man einen Wert gezielt ändern will.

```
Color gelb = new Color(255, 255, 0);
gelb = new Color(gelb.getRed() - 55, gelb.getGreen(), gelb.getBlue());
```

C.2 Beobachter-Muster

Um dieses Muster zu verstehen, muss man sich zunächst einmal klarmachen, was das Problem ist, das davon gelöst wird. Dieses Problem tritt v. a. in einer Mehrschichten-Architektur auf, wenn die Datenschicht klar von der Darstellung und der Datenhaltung getrennt ist. Wenn nun mehrere voneinander unabhängige Darstellungen der gleichen Datenschicht existieren, dann müssten alle diese Darstellungen immer wieder die Datenschicht abfragen, um die aktuellen Werte zu besitzen; dieses Verhalten wird *Polling* genannt. Oder derjenige, der die Daten ändert, müsste alle anderen benachrichtigen, dass sich die Daten geändert haben, wozu er wissen müsste, wen er alles zu benachrichtigen hat.

Im ersten Fall würde die Datenschicht ständig von u. U. sehr vielen Objekten abgefragt werden, was unnötig Rechenzeit verschwenden würde und im zweiten Fall müsste jeder, der die Daten ändert, darüber Bescheid wissen, wer alles über diese Änderung informiert werden sollte, was zu redundanten Listen führen würde.

Das Beobachter-Muster geht genau diese beiden Probleme an und liefert eine Lösung, die eine weiterhin saubere Trennung der Datenschicht von den übrigen Schichten zulässt. Die Erklärung beinhaltet auch schon die Implementierung in Java.

Die Bestandteile dieses Muster kann man Abbildung 5 entnehmen. Es gibt ein Objekt einer Datenklasse (abgeleitet von **`Observable`** aus dem **`util`**-Paket), das von mehreren anderen Objekten von Beobachterklassen (implementieren **`Observer`** aus dem **`util`**-Paket) beobachtet wird. „Beobachten“ bedeutet hierbei, über Veränderungen benachrichtigt zu werden, d. h. das beobachtete Objekt ruft bei allen Beobachtern eine bestimmte Methode auf (**`update()`**), wenn sich sein Zustand ändert. Dazu verwaltet es eine Liste aller Beobachter, die sich bei ihm angemeldet haben.

Der Vorteil liegt darin, dass Objekte nicht mehr aktiv nach Veränderungen fragen müssen, und dass es nur noch eine Liste gibt, in der alle zu benachrichtigenden Objekte gespeichert sind.

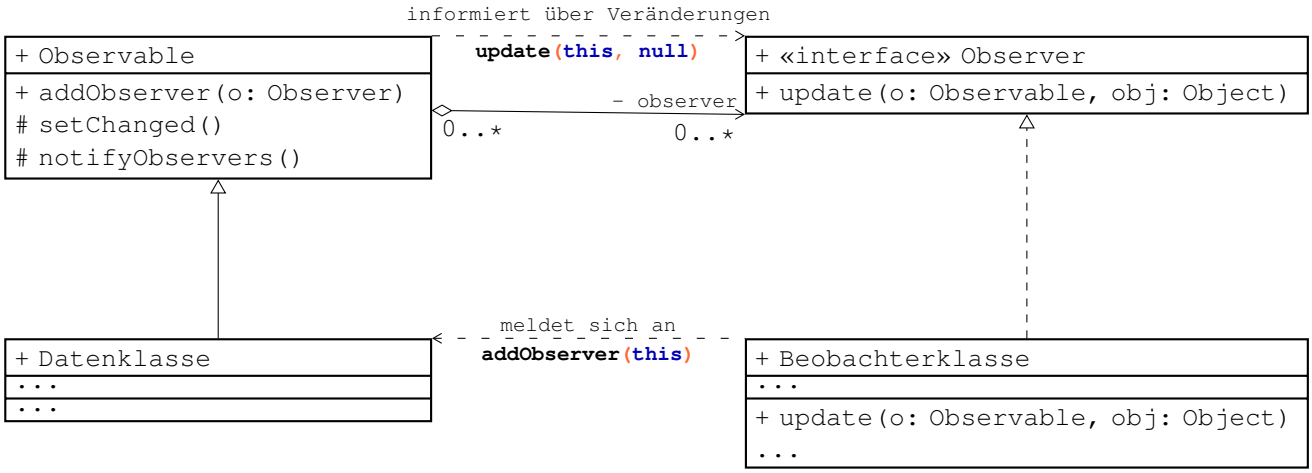


Abbildung 5: Beobachter-Muster in UML.

Die beiden Methoden von **`Observable`**, die **`protected`** deklariert sind, werden von einem Objekt der Datenklasse dazu benutzt, alle Beobachter zu informieren. Die Methode **`setChanged()`** setzt dabei den internen Zustand von **`Observable`** so, dass bei einem nachfolgenden Aufruf von **`notifyObservers()`** bei allen Beobachtern deren **`update()`**-Methode aufgerufen wird, wobei der zweite Parameter **`null`** ist [1]. Ein Aufruf von **`notifyObservers()`** ohne vorherige Ausführung von **`setChanged()`** hat keine Informierung der Beobachter zur Folge.

Wenn Sie beim Beobachter-Muster an den **`ActionListener`** oder den **`WindowListener`** oder auch den **`AdjustmentListener`** denken, dann sind Sie auf den richtigen Gedanken gekommen, denn das Muster ist lediglich eine Verallgemeinerung des Eventlistener-Konzepts.

[1] Es gibt auch eine **`notifyObservers()`**-Methode, die einen Parameter vom Typ **`Object`** erwartet. Dieser Parameter wird dann bei allen Aufrufen von **`update()`** als zweiter Parameter eingesetzt.