Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

Design Patterns

Tobias Schulz

Eingereicht am: 12.11.2012

Inhaltsverzeichnis

1 Observer

1.1 Kurzbeschreibung

Das Observer-Pattern ermöglicht eine Broadcast-Kommunikation zwischen Objekten und verringert dabei die Abhängigkeiten zwischen den beteiligten Klassen.

1.2 Szenario

Bei der objektorientierten Analyse und dem objektorientierten Design kommt es oft vor, dass man eine gefundene Klasse nochmals in weitere Klassen unterteilt. Ein typischer Fall ist die Trennung von Daten und deren Darstellung.

In Abbildung 1.1 wird schematisch der Klassenentwurf eines Formulars gezeigt. In der Klasse *Form* werden zwei verschiedene Aspekte ausgemacht, nämlich die eigentlichen Formulardaten und ihre Darstellung. Außerdem werden beide Klassen in separate Schichten der Anwendungsarchitektur verteilt.

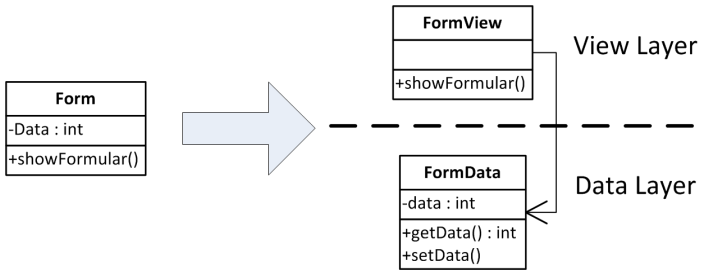

Abb. 1.1: Aufteilen der Klasse nach verschiedenen Aspekten

In diesem Fall ist die Architektur gewahrt. Die Klasse *FormData* der tieferen Schicht *DataLayer* benötigt keine Kenntnisse über Klassen der *ViewLayer* Schicht. Außerdem erhöht es die Wiederverwendbarkeit der *FormData* Klasse.

Will man nun die selben Daten zeitgleich in einem weiteren Formular auf eine andere Weise darstellen, bietet sich die Einführung einer alternativen Formularklasse an. Beispielsweise könnte man die Daten im ersten Formular numerisch, im zweiten grafisch anzeigen lassen. Abbildung 1.2 zeigt das Aufteilen der Formularklasse.

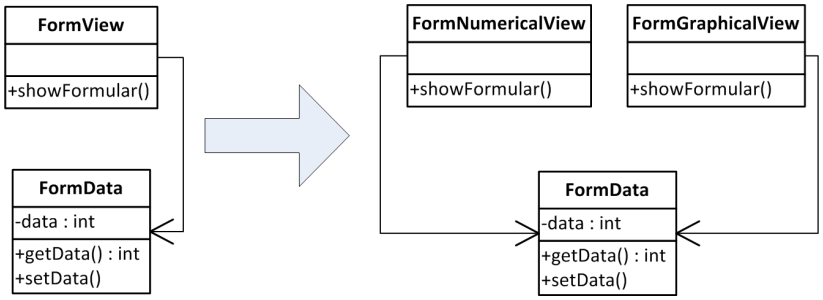

Abb. 1.2: Einführen einer neuen Darstellungsklasse des Formulars

Bei dem Entwurf in Abbildung 1.2 ergibt sich durch das Einführen einer neuen Darstellungsklasse jedoch ein Problem. Die dargestellten Daten in beiden Formularen sollen konsistent gehalten werden. Wird *FormData* durch eine der Anzeigeklassen geändert, muss die andere Anzeigeklasse über diese Änderung informiert werden, um ihre dargestellten Daten zu aktualisieren.

Abbildung 1.3 zeigt einen Entwurf, in dem *FormData* Referenzen auf die Anzeigeklassen erhält. Nach Ausführen von *setData()* werden alle Anzeigenklassen über die *update()* Methode benachrichtigt.

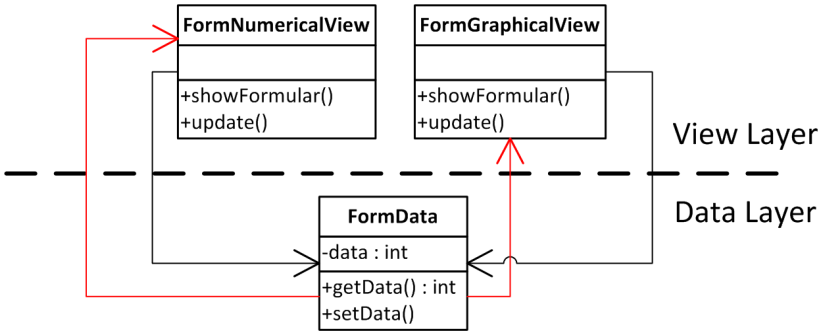

Abb. 1.3: Verletzung des Schichtenmodells (rote Assoziationen)

Durch den Entwurf in Abbildung 1.3 ergeben sich mehrere Probleme. Die roten Assoziationen in der Abbildung zeigen die Verletzung des Schichtenmodells. Die Klasse *FormData* aus dem tieferen *DataLayer* benötigt nun Informationen über Klassen aus dem höheren *ViewLayer* und stellt somit einen Verstoß gegen das Schichtenmodell dar.

Dadurch wird auch stark die Wiederverwendbarkeit von *FormData* eingeschränkt. Die Klasse kann so nicht unverändert ohne die zwei Darstellungsklassen an anderer Stelle verwendet werden.

Ein weiteres Problem ergibt sich, wenn weitere Darstellungsklassen eingeführt werden sollen, da dies Änderungen an *FormData* zur Folge hat.

1.3 Lösung

Das Observer Pattern bietet eine Möglichkeit, beliebig viele Darstellungsklassen über Änderungen zu informieren, ohne dass *FormData* Informationen über diese besitzen muss. Abbildung 1.4 zeigt einen Lösungsentwurf mit dem Observer Pattern.

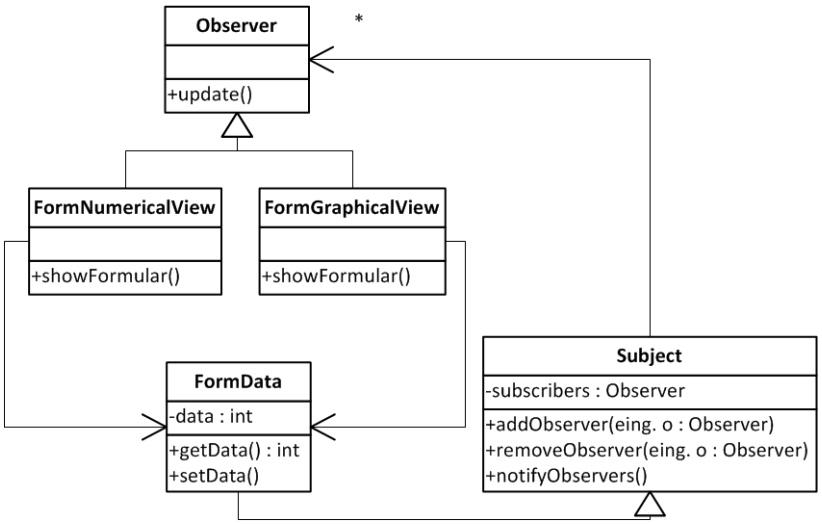

Abb. 1.4: Lösung mit dem Observer Pattern

Alle Darstellungsklassen, die über Änderungen von *FormData* informiert werden sollen, melden sich mit der geerbten Methode *addObserver(this)* an. Die Oberklasse *Subject* von *FormData* stellt ebenfalls eine Methode *removeObserver()* bereit, um die Anmeldung wieder aufzuheben. Die Referenzen auf die Observer-Objekte werden im der *subscribers* Membervariable gespeichert.

Abbildung 1.5 zeigt die Struktur des Observer Patterns.

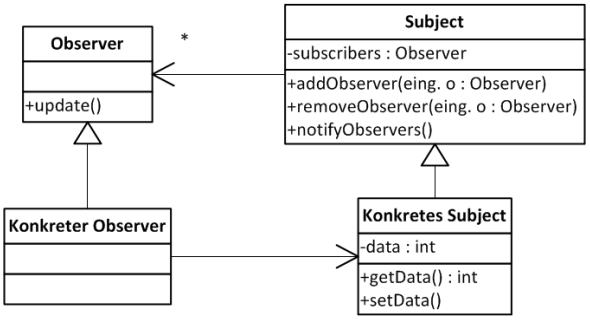

Abb. 1.5: Struktur des Observer Patterns nach [Gam11, S. 289]

Werden die Daten eines *FormData* Objekts jetzt von einem Darstellungsobjekt geändert, informiert *FormData* in *notifyObservers()* über die Referenzen in *subscribers* alle angemeldeten Objekte der Darstellungsklassen. Abbildung 1.6 zeigt die Interaktion der Objekte.

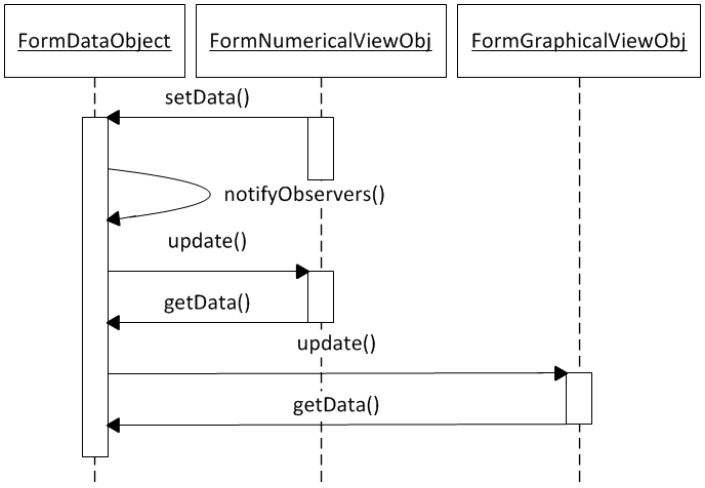

Abb. 1.6: Ablauf bei Änderung der gemeinsamen Formulardaten

Jede Änderung eines *FormData* Objekts zieht nun Aktualisierungen aller angemeldeten Objekte der Darstellungsklassen nach sich. Das Schichtenmodell wird nicht mehr verletzt. *FormData* Objekte halten nur noch Referenzen auf abstrakte *Observer* Objekte.

1.4 Implementierung

Das Beispiel in Abschnitt 1.3 ist sehr einfach gehalten, kann aber so implementiert werden. In diesem Abschnitt wird auf einige Details der Implementierung und mögliche Optimierungen eingegangen.

1.4.1 Erweitern der Benachrichtigungs-Schnittstelle

In dem einfachen Beispiel in Abschnitt 1.3 müssen die Darstellungsobjekte lediglich ein einziges Datum des *FormData* Objekts auslesen, um sich zu aktualisieren. Würde die Darstellungsklasse mehrere *Subject*-Referenzen besitzen, wäre beim Aufruf der *update()* Methode nicht klar, von welchem *Subject*-Objekt der Aufruf stammt. Die Darstellungsobjekte müssten die Daten aller referenzierten *Subject*-Objekte aktualisieren (siehe [Gam11, S. 292]).

Eine Möglichkeit zur Reduzierung dieses Aufwandes wäre eine Erweiterung der Benachrichtigungs-Schnittstelle. Bei Aufruf von *update()* kann das *Subjekt* sich selbst als Parameter mitliefern (Aufruf von *update(this)*). Der Empfänger muss dann nur noch die *getData()* Methode des mitgelieferten Objekts aufrufen. Abbildung 1.7 stellt diesen Ablauf dar.

Der Laufzeit-Aufwand bei der Aktualisierung könnte noch weiter reduziert werden, wenn das *Subject*-Objekt weitere Informationen über seine Änderung übermittelt. Dazu benötigt es aber wiederum Informationen über die Bedürfnisse der Darstellungsklasse, was die Kopplung zwischen beiden Klassen erhöhen und die Wiederverwendbarkeit reduzieren würde.

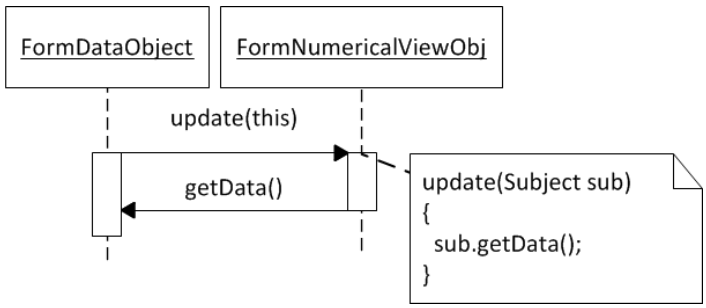

Abb. 1.7: Das *FormNumericalViewObj* ruft nur die *getData()* Methode des mitgelieferten Objektes auf.

1.4.2 Reduzierung der Aktualisierungen

Bei der Lösung in Abschnitt 1.4 veranlasst jede Änderung des *FormData*-Objekts durch die Darstellungsobjekte eine Kaskade von Aktualisierungen. In Abbildung 1.8 nimmt *FormNumericalViewObj* zwei Änderungen an dem *FormData* Objekt vor, mit *setData1()* und *setData2*. Jede Änderung führt zur Benachrichtigung aller angemeldeten Observer (siehe [Gam11, S. 292 f.]).

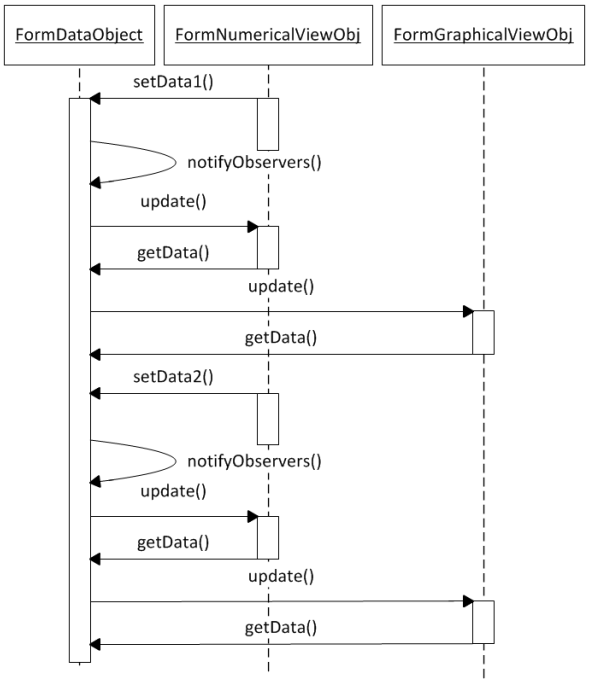

Abb. 1.8: Mehrmaliges Ändern des *FormData*-Objekts veranlasst mehrmalige *update()* Aufrufe

In dem Beispiel aus Abbildung 1.8 wäre es sinnvoll, wenn die Aktualisierung erst erfolgt, wenn alle Änderungen an dem *FormData*-Objekt durch *FormNumericalView* abgeschlossen sind.

Eine Lösung dieser Anforderung zeigt Abbildung 1.9.

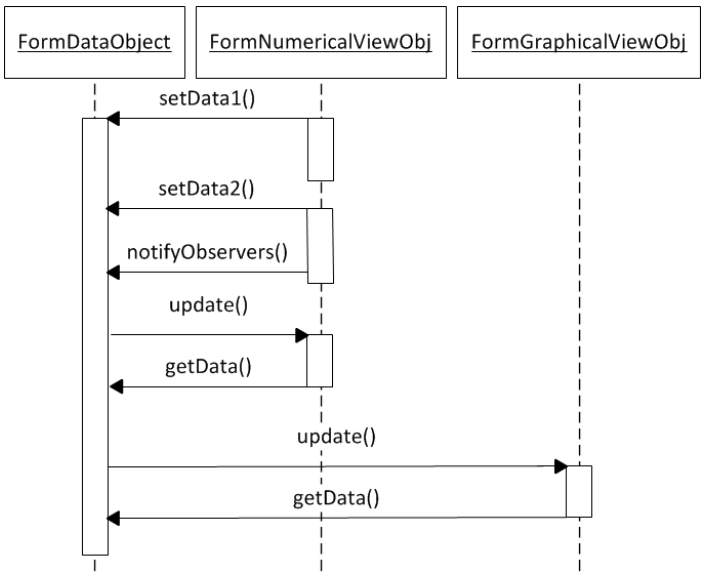

Abb. 1.9: Das FormNumericalView Objekt veranlasst selbst die Benachrichtigung aller anderen Observer

Statt des *FormData*-Objekts nach jeder Datenänderung, ruft *FormNumericalView* die Methode *notifyObservers()* auf. Das *FormNumericalView*-Objekt entscheidet so, zu welchem Zeitpunkt eine Aktualisierung aller Observer sinnvoll ist. Diese Information besitzt das *FormData*-Objekt nicht. Ein Nachteil dieser Lösung ist, dass jede *Observer* Implementierung selbst die Verantwortung trägt, die *update()*s zu veranlassen. Vergessene *notifyObservers()* Aufrufe führen zu schwer zu findenden Fehlern.

Eine weitere Möglichkeit die Anzahl der Aktualisierungen zu reduzieren ist, *Subject*s mit weiteren Informationen zu versorgen. Dazu wird die Anmeldemethode des *Subject*s erweitert. Mit *addObserver(Observer, Aspect)* wird dem *Subject* über *Aspect* mitgeteilt, welche Änderungen den *Observer* interessieren. So wird der *Observer* nur bei bestimmten Ereignissen informiert.

1.4.3 Kapselung der Aktualisierungs-Logik

Bei komplexen *Observer-Subject*-Beziehungen kann es vorkommen, dass *Observer* bei einer Operation mehrfach aktualisiert werden. Dies ist der Fall, wenn ein *Observer* mehrere *Subject*s besitzt. Diese redundanten Aktualisierungen können vermieden werden, wenn die *Observer* erst nach Abschluss dieser Operation benachrichtigt werden (siehe [Gam11, S. 295]).

Abbildung 1.10 zeigt dazu eine Lösung mit einem *Mediator*. Das *Mediator* ist ein weiteres Design Pattern. Es kapselt Informationen, die nötig sind, um die Benachrichtigungen an die *Observer* gering zu halten.

Die *Subject*s melden sich nun am *Mediator* an und halten selbst keine Referenzen mehr auf *Observer*-Objekte. Der *Mediator* besitzt eine Datenstruktur, die die *Subjekt-Observer*-

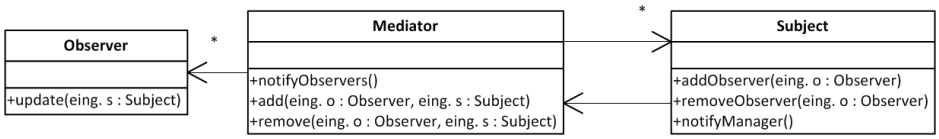

Abb. 1.10: Der *Mediator* kapselt komplexe Beziehungen und Interaktionen zwischen *Observer* und *Subject*

Beziehungen enthält. Wird nun eine Operation durchgeführt, bei der mehrere *Subject*-Objekte geändert werden, werden nicht sofort alle *Observer* informiert. Erst nach Abschluss der Operation werden die betroffenen *Observer* informiert.

1.4.4 Vermeiden von fehlerhaften Referenzen

Wenn ein *Subject*-Objekt gelöscht wird, müssen dessen Beobachter darüber informiert werden. Da dem *Subject*-Objekt alle *Observer* bekannt sind, kann es sie informieren, falls es gelöscht wird (siehe [Gam11, S. 293]).

In C++ könnte beispielsweise das *Subjekt* in seinem Destruktor über eine Methode *notifyDeletion()* alle seine *Observer* über seine Löschung informieren.

In Sprachen wie C# oder Java, die ein automatisches Speichermanagement besitzen, muss der Programmierer sich nicht um diese fehlerhafte Referenzen kümmern.

1.5 Vorteile

Die Kopplung zwischen *Subject* und *Observer* ist lose. Es wird erst zur Laufzeit bestimmt welche *Subjects* welche *Observer* benachrichtigen. So können im Programmverlauf dynamisch Beziehungen auf- und abgebaut werden (siehe [Gam11, S. 291]).

Da die Beziehungen zwischen den konkreten *Subject* und *Observer* Klassen im Voraus nicht bekannt sein muss, erleichtert das deren Entwicklungsprozess. Es können leicht neue konkrete *Observer* hinzugefügt werden (siehe [Eil10, S. 62]).

1.6 Nachteile

Konkrete *Observer* wissen nichts voneinander. Eine einfache Änderung des *Subject*s kann hohe Laufzeitkosten verursachen, wenn viele *Observer* daran angemeldet sind (siehe [Eil10, S. 63]).

Es ist schwer in Komplexen Systemen den Überblick über die Beziehungen zwischen *Observer*- und *Subject*-Objekten zu behalten. Dies erschwert die Fehlersuche, da die Beziehungen erst zur Laufzeit etabliert werden. Es kann zu zirkulären Benachrichtigungen kommen (siehe [Eil10, S. 63]).

Literaturverzeichnis

[Eil10] Eilebrecht, K.; Starke, G.: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*, Spektrum, Akad. Verl., Heidelberg, 3. Ausg., 2010.

[Gam11] Gamma, E.; Riehle, D.; Helm, R.; Johnson, R.; Vlissides, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley, München and Boston and Mass. [u.a.], 6. Ausg., 2011.