

2 Das Projekt Taschenrechner

Bemerkung:

Das Projekt *Taschenrechner* habe ich aus dem Lehrbuch *Java lernen mit BlueJ* von **M. Kölling** und **D. Barnes** übernommen und es an meinen Unterricht angepasst. Die Entwurfsmuster werden beschrieben im Buch *Entwurfsmuster von Kopf bis Fuß* von **Eric Freeman** und **Elisabeth Freeman**. Die Gestaltungsmöglichkeiten einer grafischen Oberfläche mit Swing werden in *The JFC Swing Tutorial, Second Edition* von **Kathy Walrath** u.a. dargestellt.

Dieses Projekt modelliert einen sehr einfachen Taschenrechner, der nur

- a) mit ganzen, maximal zweistelligen Zahlen rechnen kann,
- b) die Rechenoperationen *Addition* und *Subtraktion* beherrscht.

Die grafische Oberfläche, also die Sicht auf den Taschenrechner, besteht aus den beiden Teilen Tastatur und Display. Dahinter steckt natürlich ein Rechnermodell, das diese Sichten steuert, also auf die Benutzereingaben reagiert und die Rechenergebnisse im Display anzeigt.

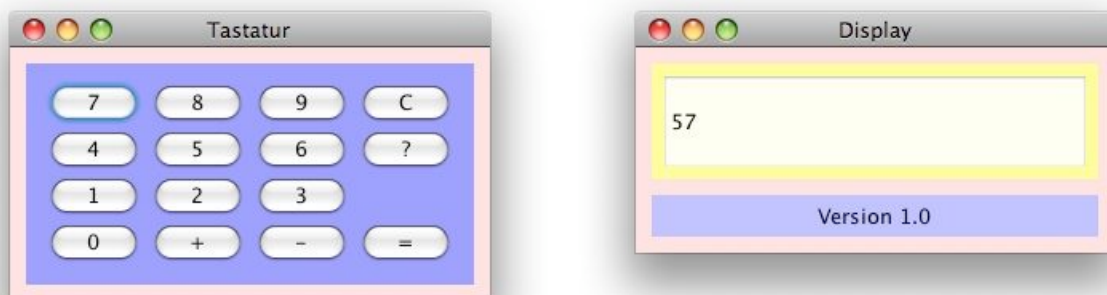


Abbildung 2.1: Elemente des Taschenrechners

Du wirst lernen, wie der Taschenrechner aus den eingegebenen Ziffern die Zahlen aufbaut und mit diesen Zahlen einfache Rechenoperationen (in diesem Projekt nur die Addition und die Subtraktion) durchführt. Dazu müssen die beiden in Abbildung 2.1 dargestellten Sichtweisen mit dem Rechnermodell zusammen arbeiten. Hierzu werden die Entwurfsmuster **Strategy-Muster** und **MVC-Muster** (Model-View-Controller) eingeführt.

2.1 Einführende Überlegungen

Natürlich ist es sinnvoll, zuerst ein Modell des Taschenrechners zu entwickeln. Hierzu solltest du dir überlegen,

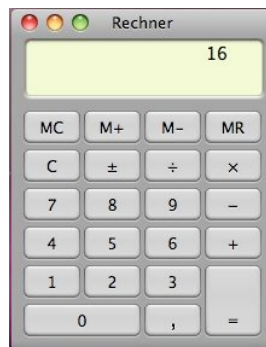
1. wie der Taschenrechner funktionieren soll,
2. ob spätere weitere Funktionen hinzukommen sollen.

Mache dir zuerst Gedanken darüber machen, wie ein Taschenrechner auf Benutzereingaben reagiert. Dieses Verhalten untersuchst du am besten an Hand der folgenden Übung.

Übung 2.1:

Sowohl auf dem Mac als auch auf dem PC existiert ein kleines Programm *Rechner*.

Öffne dieses und untersuche die folgenden Rechenaufgaben:



- a) $5 + 7 = [12]$
- b) $57 + 4 = [61]$
- c) $5 + 7 + [12] 4 = [16]$
- d) $5 + 7 = [12] + 4 = [16]$

Die in den Klammern kursiv dargestellten Zahlen sind die Ergebnisse, die der Rechner liefert. Überlege, welche Rechenoperationen der Taschenrechner durchführt, wenn der Benutzer die oben aufgeführten Aufgaben berechnen will.

Aufgabe a) scheint recht eindeutig zu sein. Die beiden Operanden 5 und 7 bilden mit dem Operator *plus* die Rechenoperation Addition.

Aber schon in Aufgabe b) zeigen sich die ersten Schwierigkeiten. Wenn du 5 eintippst, wird diese Ziffer als Einerstelle erkannt. Tippst du anschließend die Ziffer 7 ein, rückt die 5 auf die Zehnerstelle und die Einerstelle ist nun belegt mit der Ziffer 7. Offensichtlich erkennt der Rechner, dass du erst mit dem Drücken des *Plus*-Operators den Aufbau der Zahl abgeschlossen hast. Der Rechner macht sich nun also bereit für den neuen Operanden 4. Das Tippen auf den *Istgleich*-Operator zeigt wiederum dem Rechner, dass auch der Aufbau des zweiten Operanden nun abgeschlossen ist. Nun kann er die Rechenoperation Addition mit diesen beiden Operanden durchführen. Hoffentlich hat der Rechner sich gemerkt, dass der Benutzer bereits schon den *Plus*-Operator gedrückt hat.

Aufgabe c) zeigt, dass das Tippen des zweiten *Plus*-Operators die gleiche Wirkung hat wie der *IstGleich*-Operator. Der Rechner weiß allerdings, dass die Rechnung noch nicht abgeschlossen ist, sondern dass der Benutzer eine weitere Rechenoperation durchführen will. Das Zwischenergebnis 12 wird also jetzt zum linken Operanden einer weiteren Addition und der Rechner wartet auf den neuen Operanden.

Aufgabe d) zeigt, dass der Benutzer nach einem *IstGleich*-Operator einen zweiten Operator verwenden darf. Der *IstGleich*-Operator beendet also nicht die Berechnung, sondern speichert das Ergebnis im linken Operanden und setzt den Taschenrechner in eine Art Wartezustand für eine eventuelle weitere Rechenoperation.

Das gleiche Verhalten zeigt der Taschenrechner auch bei der Subtraktion. Hier wird der *Plus*-Operator nur ersetzt durch einen *Minus*-Operator.

Es gibt also einiges zu tun.

2.2 Das Strategy-Muster

2.2.1 Einführung in das Strategy-Muster

Zuerst wirst du die beiden Rechenoperationen *Addition* und *Subtraktion* untersuchen. Bei beiden Rechenoperationen erwartet der Taschenrechner zwei Operanden und einen Operator:

	Operanden	Operator
Addition	erster Summand und zweiter Summand	plus
Subtraktion	Minuend und Subtrahend	minus

Es werden also immer zwei Zahlen benötigt und eine anschließende Berechnung ergibt dann das Ergebnis. Nur bei den Rechenarten zeigt der Taschenrechner ein unterschiedliches Verhalten. Bei der Addition werden diese beiden Zahlen addiert, bei der Subtraktion soll eine Zahl von der anderen subtrahiert werden. Und bei einer späteren Multiplikation oder Division zeigt der Taschenrechner zwei weitere Verhaltensweisen. Jede Verhaltensweise, die du später einführst oder veränderst, musst du in all den verschiedenen Methoden, Unterklassen usw. aufspüren und ändern und dabei wahrscheinlich neue Fehler einbauen.

Jedoch gibt es für solche Fälle ein Design-Prinzip:

Identifiziere die Aspekte deiner Anwendung, die sich ändern können, und trenne sie von denen, die konstant bleiben.

Dieses Design-Prinzip bedeutet:

Du musst in deinem Quelltext alle Teile nehmen, die sich ändern können, und sie kapseln, damit du diese veränderlichen Teile später ändern oder erweitern kannst, ohne dass das Auswirkungen auf die Teile hat, die sich nicht ändern. Somit hat eine Quelltextveränderung weniger unvorhergesehene Folgen und die Flexibilität bei Erweiterungen wird erhöht.

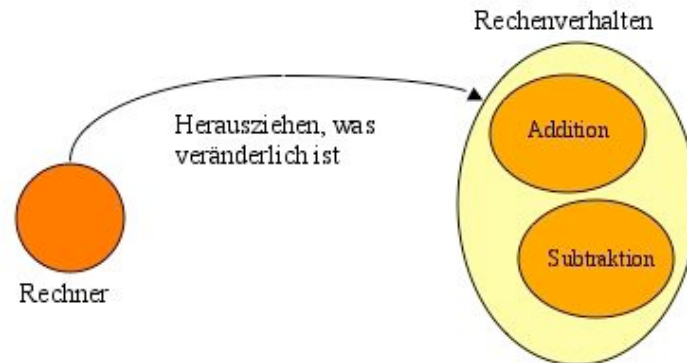


Abbildung 2.2: Das, was veränderlich ist, von dem trennen, was gleich bleibt

Du wirst nun also einen Satz von Klassen erstellen, deren einziger Existenzgrund die Repräsentation eines Verhaltens ist.

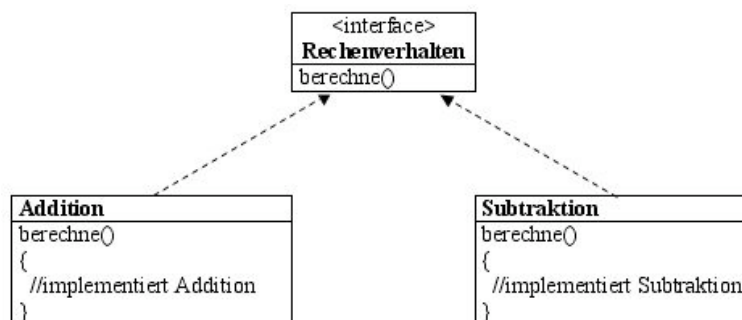


Abbildung 2.3: Das aktuelle Rechenverhalten existiert in einer separaten Klasse

Der Rechner muss nun also nichts mehr über die Details seines eigenen Rechenverhaltens wissen. Er besitzt nur noch ein Datenfeld *rechenart* vom Datentyp *Rechenverhalten*. Dieses Datenfeld erhält dann je nach gewünschter Rechenart eine Referenz auf die Addition oder Subtraktion. Wenn später der Rechner auch noch multiplizieren soll, so wird dieses neue Verhalten einfach hinzugefügt, ohne eine bereits vorhandenen Verhaltensklasse oder gar die Klasse *Rechner* zu ändern.

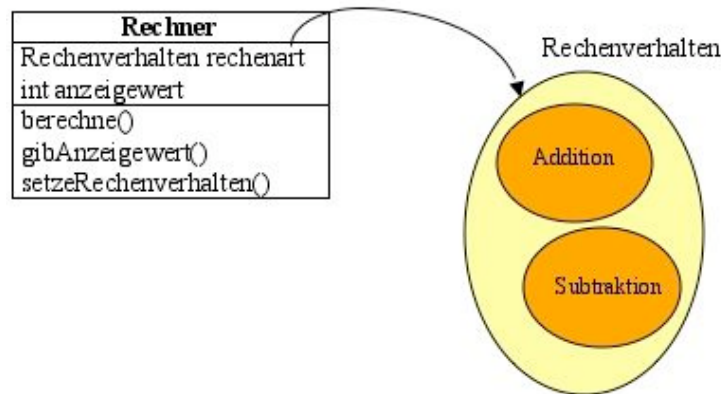


Abbildung 2.4: Datenfeld *rechenart* erhält eine Referenz auf ein bestimmtes Verhalten, das zur Laufzeit verändert werden kann

Bevor du das Strategy-Muster implementierst, solltest du dir die Zusammenhänge an Hand des Klassendiagramms in Abbildung 2.5 klar machen.

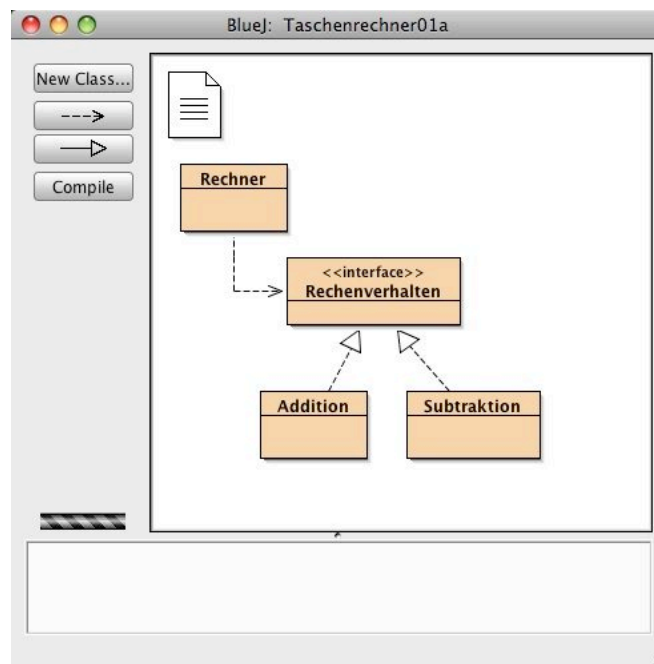


Abbildung 2.5: Klassendiagramm des Projekts *Taschenrechner01a*

Du erkennst die beiden Klassen *Addition* und *Subtraktion*. Da sie zu einer Familie von Algorithmen gehören, die das Rechenverhalten darstellen, implementieren sie das Interface *Rechenverhalten*. Laut Abbildung 2.3 wird somit vertraglich vereinbart, dass diese Klassen die Methode *berechne()* besitzen. Die Klasse *Rechner* besitzt nun ein Datenfeld *rechenart* vom Typ *Rechenverhalten* und sie kann nun auf die Methode *berechne()* zugreifen. Sie benötigt allerdings keine Details über diese Methode. Somit kannst du später

leicht weitere Rechenverhalten hinzufügen oder ändern, ohne den Quelltext der Klasse *Rechner* neu zu programmieren.

2.2.2 Modellierung des Strategy-Musters

Wie Abbildung 2.4 zeigt, benötigst du die beiden Datenfelder *rechenart* für das entsprechende Rechenverhalten und *anzeigewert* für das Ergebnis der Rechenoperation.

```
public class Rechner
{
    private Rechenverhalten rechenart;
    private int anzeigewert;

    public Rechner()
    {
        //leer
    }

    public void berechne(int a, int b)
    {
        anzeigewert = rechenart.berechne(a, b);
    }

    public int gibAnzeigewert()
    {
        return anzeigewert;
    }

    public void setzeRechenverhalten(Rechenverhalten rechenart)
    {
        this.rechenart = rechenart;
    }
}
```

Abbildung 2.6: Quelltext der Klasse *Rechner*

Die Methode *setzeRechenverhalten()* legt fest, welche Rechenoperation der Rechner verwenden soll. Die Methode *berechne()* erwartet als Parameter die beiden Operanden und ruft damit die Methode *berechne()* der Rechenoperation Addition oder Subtraktion auf, je nachdem, welches Verhalten im Datenfeld *rechenart* gespeichert wurde.

Was aber wird nun im Datenfeld *rechenart* gespeichert. Ist es ein Objekt der Klasse *Addition* oder ein Objekt der Klasse *Subtraktion*. Dann sollte der Datentyp *Addition* oder *Subtraktion* sein. Offensichtlich sollen jedoch beide Datentypen in das Datenfeld *rechenart* passen.

Dieses Problem wird mit Hilfe eines Interfaces gelöst. Jede Rechenoperation, die du später benötigst, muss dieses Interface implementieren.

```
public interface Rechenverhalten
{
    public int berechne(int a, int b);
}
```

Abbildung 2.7: Quelltext des Interfaces *Rechenverhalten*

Bemerkung:

Methoden in einem Interface sind öffentlich, *public* ist also nicht notwendig!

Betrachte ein Interface zunächst einmal als eine Vereinbarung, dass alle Klassen, die dieses Interface implementieren, auch die Methode *berechne()* besitzen. Der Rechner kann nun also sicher sein, dass jede Rechenart mit den beiden Parametern *a* und *b* etwas anfangen kann. Er muss nichts darüber wissen, was die einzelnen Rechenarten nun damit anfangen, sie liefern auf jeden Fall ein Ergebnis an den Rechner zurück.

Weiterhin wird hier auch der Polymorphismus genutzt, indem auf ein Supertyp programmiert wird, damit das tatsächliche Laufzeitobjekt nicht im Quelltext festgeschrieben werden muss. Der Rechner verwendet nur noch Supertypen *Rechenverhalten*, er muss also die tatsächlichen Objekttypen (Addition, Subtraktion, usw.) nicht kennen. Somit können an den Rechner alle möglichen Rechenarten angeknüpft werden, wichtig ist nur, dass diese das Interface *Rechenverhalten* implementiert haben. Wird nun eine neue Rechenart hinzugefügt, so muss der Rechner selbst nicht mehr verändert werden.

Rechner und alle Rechenarten sind locker gebunden, sie können miteinander agieren, müssen aber nur wenige Kenntnisse voneinander besitzen.

```
public class Addition implements Rechenverhalten
{
    public Addition()
    {
        //leer
    }

    public int berechne(int summandA, int summandB)
    {
        return summandA + summandB;
    }
}
```

Abbildung 2.8: Quelltext der Klasse *Addition*

Übung 2.2:

Erzeuge in **BlueJ** ein Projekt *Taschenrechner01a*.

- a) Implementiere den Quelltext der Klassen *Rechner* und *Addition* und des Interfaces *Rechenverhalten*.
- b) Teste dein Projekt. Erzeuge ein Objekt der Klasse *Addition* und ein Objekt der Klasse *Rechner*. Überprüfe nun mit Hilfe des Inspektors die Methoden *setzeRechenverhalten()* und *berechne()*.
- c) Implementiere den Quelltext der Klasse *Subtraktion*. Das entsprechende Klassendiagramm zeigt Abbildung 2.5. Teste nun dein Projekt

Die Übung 2.2 c) zeigt, wie einfach es ist, neue Rechenoperationen einzuführen. Und sollte später einmal die Addition neu definiert werden (z.B. $a + b = 2a + b$), stellt auch diese Änderung kein Problem mehr dar.

Außerdem kannst du während der Laufzeit dem Taschenrechner ein neues Verhalten, man sagt auch eine neue Strategie, zuweisen. Du rufst einfach die Methode *setzeRechenverhalten()* auf und änderst somit den Wert des Datenfelds *rechenart*.

Und vielleicht willst du später auf deinem Taschenrechner Lieder abspielen. So implementierst du ein weiteres Interface *Musikverhalten* mit einer Familie von Verhaltensklassen wie *Beat*, *Hip Hop* oder *Reggy*.

Bemerkung:

Zwischen den Klassen besteht eine HAT-EIN-Beziehung. Der Rechner hat ein Rechenverhalten. Werden zwei Klassen auf diese Art zusammengefügt, bezeichnet man dies als **Komposition**. Der Taschenrechner erhält sein Verhalten durch Komposition mit dem entsprechenden Verhaltensobjekt. Solch eine Komposition ermöglicht es, das Verhalten zur Laufzeit zu ändern.

Nun zur Definition des **Strategy-Musters**:

Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen, variieren zu lassen.

Übung 2.3:

Suche im Internet eine Beschreibung für das **Strategy-Muster**. Findest du auch weitere Entwurfsmuster?

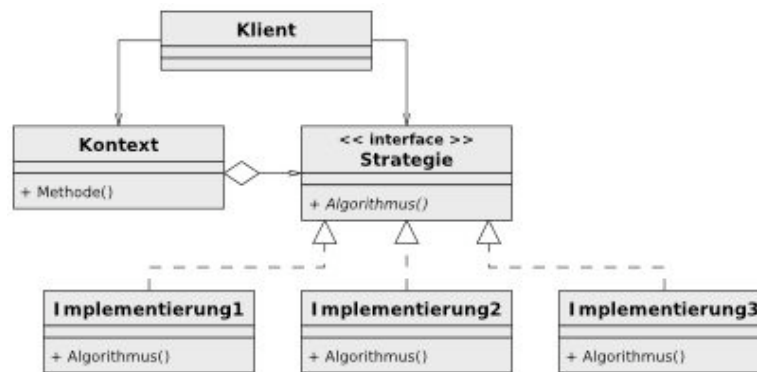


Abbildung 2.9: Das Strategy-Muster

Suchst du nach dem Strategy-Muster, wirst du Abbildungen wie oben gezeigt finden. Hier werden folgende Akteure verwendet:

Abbildung 2.8	Projekt <i>Taschenrechner01</i>
<<interface>> Strategie	Schnittstelle <i>Rechenverhalten</i> legt wie in einem Vertrag die Methode <i>berechne()</i> fest.
Implementierung1 Implementierung2	Konkrete Strategien <i>Addition</i> und <i>Subtraktion</i> implementieren die Schnittstelle <i>Rechenverhalten</i> .
Kontext	<i>Rechner</i> stellt den eigentlichen Kontext dar.
Klient	Klient stellt eine Simulation oder einen Testlauf mit der Methode <i>main()</i> dar.

Bemerkung:

Da in **BlueJ** Objekte direkt erzeugt werden können, brauchte die Klasse *Klient* nicht programmiert werden. Diese könnte folgenden Quelltext enthalten:

```

public class Klient
{
    public static void main(String[] args)
    {
        System.out.println("Testlauf von Taschenrechner01");

        Rechner rechner = new Rechner();
        rechner.setzeRechenverhalten(new Addition());
        rechner.berechne(5, 7);
        System.out.println(rechner.gibAnzeigewert());

        // Wechsel der Strategie
        rechner.setzeRechenverhalten(new Subtraktion());
        rechner.berechne(5, 7);
        System.out.println(rechner.gibAnzeigewert());
    }
}
  
```

}

Abbildung 2.10: Möglicher Quelltext der Klasse *Klient*

In der Klasse *Klient* wurde der Wechsel des Rechenverhaltens durchgeführt mit den Anweisungen

```
rechner.setzeRechenverhalten(new Addition());
rechner.setzeRechenverhalten(new Subtraktion());
```

Hierfür ist es sinnvoll, separate Methoden *plus()* bzw. *minus()* zu verwenden.

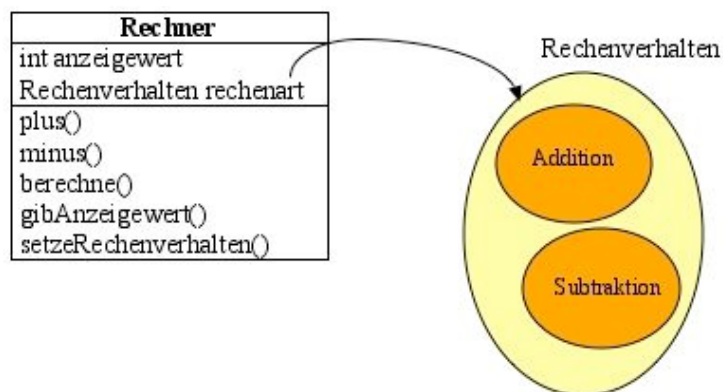
```
public void plus()
{
    setzeRechenverhalten(new Addition());
}
```

Abbildung 2.11: Quelltext der Methode *plus()*

Übung 2.4:

Speichere dein Projekt unter dem Namen *Taschenrechner01b*.

- Implementiere die Methode *plus()* aus Abbildung 2.10 und setze anschließend die Methode *setzeRechenverhalten()* als *private*.
- Teste die Methode *plus()*.
- Implementiere eine geeignete Methode *minus()* und teste diese.

**Abbildung 2.12:** Modell des Rechners im Projekt *Taschenrechner01b*

2.3 Modellierung des Rechners

Nun wirst du das bisherige Modell des Taschenrechners näher an die Realität heranbringen. Hierzu werden die Vorüberlegungen aus Kapitel 2.1 betrachtet:

Der Benutzer will nun die Rechnung $5 + 7 = [12]$ durchführen. Hierzu werden folgende vier Schritte abgearbeitet:

1. Er tippt zuerst die Ziffer 5, die anschließend gleich in der Anzeige erscheint, also im Datenfeld *anzeigewert* gespeichert wird.
2. Nun drückt der Benutzer den *Plus*-Operator. Hierbei erfolgen zwei Dinge:
 - a) Der Anzeigewert 5 wird zum linken Operanden der Rechenoperation. Der Wert 5 wird also in das Datenfeld *linkerOperand* gespeichert und somit ist das Datenfeld *anzeigewert* wieder frei für den nächsten Operanden 7.
 - b) Das Rechenverhalten des Taschenrechners wird auf Addition gesetzt.
3. Nun kann er Benutzer auf die Ziffer 7 drücken. Diese erscheint in der Anzeige und ist somit in *anzeigewert* gespeichert.
4. Zum Abschluss drückt der Benutzer den *IstGleich*-Operator. Der Taschenrechner sucht sich die beiden Operanden, die in den Datenfeldern *linkerOperand* und *anzeigewert* gehalten werden, und benutzt diese als Parameter für die Methode *berechne()*.

Übung 2.5:

- a) Ordne die unten angegebenen Methoden den oben genannten Punkten zu.
- b) Speichere dein Projekt unter dem Namen *Taschenrechner02a*. Ergänze die Klasse *Rechner* mit den fehlenden Methoden und kommentiere diese.
- c) Teste dein Projekt ausführlich.

```
public void plus()
{
    operatorAnwenden();
    setzeRechenverhalten(new
Addition());
}

private void berechne(int a, int b)
{
    anzeigewert =
rechenart.berechne(a, b);
}

public int gibAnzeigewert()
{
    return anzeigewert;
}

public void gleich()
{
    berechne(linkerOperand,
anzeigewert);
}

private void operatorAnwenden()
{
    linkerOperand = anzeigewert;
}

public void minus()
{
    operatorAnwenden();
    setzeRechenverhalten(new
Subtraktion());
}
```

```

private void setzeRechenverhalten(Rechenverhalten rechenart)
{
    this.rechenart = rechenart;
}

public void zifferGetippt(int ziffer)
{
    anzeigewert = ziffer;
}

```

Jetzt wirst du kennen lernen, wie man eine zweistellige Zahl in deinen Taschenrechner eingeben kann.

Der Benutzer will nun die Rechnung $57 + 4 = [61]$ durchführen.

Selbstverständlich kannst du der Methode *zifferGetippt()* den Parameterwert 57 übergeben. Dies ist allerdings nicht realistisch, da der Benutzer eines Taschenrechners zuerst auf die Taste *fünf* drückt, anschließend erscheint in der Anzeige die Zahl 5. Nun drückt der Benutzer auf die Taste *sieben*, nun erscheint allerdings nicht die Zahl 7, sondern die 5 rückt von der Einerstelle auf die Zehnerstelle und die neue Einerstelle ist die 7, so dass in der Anzeige die Zahl 57 dargestellt wird. Aus den beiden Ziffern *fünf* und *sieben* wurde also die Zahl 57 gebildet und in der Anzeige dargestellt.

Vergleiche die obige Rechnung $57 + 4 = [61]$ mit der bereits besprochenen Rechnung $5 + 7 = [12]$. Offensichtlich wird durch das Drücken des *Plus*-Operators der Aufbau der Zahl 57 aus ihren beiden Ziffern abgeschlossen.

Dieses Verhalten führt zu Änderungen in den Methoden *ziffernGetippt()* und *operatorAnwenden()*.

```

public void zifferGetippt(int ziffer)
{
    if (anzeigewertImAufbau) {
        //diese Ziffer in einen bestehenden Operanden einbauen
        anzeigewert = anzeigewert * 10 + ziffer;
    }
    else {
        //mit dieser Ziffer einen neuen Operanden beginnen
        anzeigewert = ziffer;
        anzeigewertImAufbau = true;
    }
}

```

Abbildung 2.13: Quelltext der Methode *zifferGetippt()*

Hier wird ein neues Datenfeld *boolean anzeigewertImAufbau* verwendet, das im Anfangszustand auf *false* gesetzt wurde. Drückt der Benutzer die Ziffer *fünf*, so wird diese in der Anzeige dargestellt und *anzeigewertImAufbau* auf *true* gesetzt.

Drückt nun der Benutzer auf die nächste Ziffer *sieben*, wird hiermit die Zahl 57 aufgebaut.

```
private void operatorAnwenden()
{
    //Aufbau des Operanden ist abgeschlossen
    anzeigewertImAufbau = false;

    linkerOperand = anzeigewert;
}
```

Abbildung 2.14: Quelltext der Methode *operatorAnwenden()*

Drückt allerdings der Benutzer nach Eingabe der Ziffer *fünf* auf den *Plus*-Operator, so wird die Methode *operatorAnwenden()* aufgerufen, die *anzeigewertImAufbau* auf *false* setzt und somit den Aufbau einer Zahl für beendet erklärt.

Nützlich wird es in Zukunft sein, im Konstruktor den Rechner in einen sinnvollen Anfangszustand zu setzten.

```
public Rechner()
{
    setzeAnfangszustand();
}

private void setzeAnfangszustand()
{
    anzeigewertImAufbau = false;
    anzeigewert = 0;
}
```

Abbildung 2.15: Aufruf der Methode *setzeAnfangszustand()* im Konstruktor

Übung 2.6:

- Speichere dein Projekt unter dem Namen *Taschenrechner02b*. Ergänze die Klasse *Rechner* mit den erweiterten Methoden *zifferGetippt()*, *operatorAnwenden()* und *setzeAnfangszustand()*. Ergänze den Konstruktor. Kommentiere deinen Quelltext ausführlich.
- Teste dein Projekt mit Hilfe des Inspektors. Führe beispielsweise die Rechnung $57 + 4 = [61]$ durch. Führe gleich anschließend die weitere Rechnung $[61] + 7 = [68]$ aus.
- Führe nun die Rechnung $57 + 4 + 7 = [68]$. Welches Problem taucht nun auf? Verwende den Debugger, um das Problem zu isolieren!

Du wirst wahrscheinlich festgestellt, dass die Rechnung $57 + 4 + 7$ das Ergebnis 11 liefert. Das Problem liegt in der Methode *operatorAnwenden()*. Der

Taschenrechner vergisst offensichtlich, das Zwischenergebnis 61 zu berechnen. Die beiden Abbildungen 2.16 und 2.17 geben einen tabellarischen Verlauf dieser beiden Rechnungen wider.

	Benutzereingabe	LinkerOperand	Anzeigewert
1.	zifferGetippt(5)	0	5
2.	zifferGetippt(7)	0	57
3.	plus()	57	57
4.	zifferGetippt(4)	57	4
5.	gleich()	57	61
6.	plus()	61	61
7.	zifferGetippt(7)	61	7
8.	gleich()	61	68

Abbildung 2.16: Rechenoperationen $57 + 4 = [61]$ und $[61] + 7 = [68]$

	Benutzereingabe	LinkerOperand	Anzeigewert
1.	zifferGetippt(5)	0	5
2.	zifferGetippt(7)	0	57
3.	plus()	57	57
4.	zifferGetippt(4)	57	4
5.	plus()	4	4
6.	ziffergetippt(7)	4	7
7.	gleich()	4	11

Abbildung 2.17: Rechenoperationen $57 + 4 + 7 = [11]$

Abbildung 2.17 zeigt, dass der Benutzer im 4. Schritt bereits die Zahl 57, den *Plus*-Operator und die Zahl 4 eingegeben hat. Im 5. Schritt drückt nun der Benutzer wiederum den *Plus*-Operator. Die Methode *plus()* ruft die Methode *operatorAnwenden()* auf, die den momentanen *anzeigewert* (hier 4) in *linkerOperand* hineinschreibt. Es kann also nicht mehr der Wert der Summe $57 + 4 = [61]$ berechnet werden. Offensichtlich muss der Taschenrechner nicht nur beim Drücken des *Gleich*-Operators, sondern auch beim Drücken des *Plus*-Operators die Summe berechnen. Somit muss die Methode *operatorAnwenden()* auch die Methode *berechne()* aufrufen, allerdings nur, wenn noch nicht die Methode *gleich()* aufgerufen wurde.

Übung 2.7:

- Speichere dein Projekt unter dem Namen *Taschenrechner02c*. Ergänze in der Klasse *Rechner* die Methoden *operatorAnwenden()* und *gleich()* und

kommentiere diese. Beachte, dass bei der Erzeugung eines Objekts *rechner* der Wert von *operatorGegeben* auf *false* gesetzt werden muss.

```
private void operatorAnwenden()      public void gleich()
{                                     {
    anzeigewertImAufbau = false;      berechne(linkerOperand,
                                     berechne(linkerOperand,
                                     anzeigewert);
    if (operatorGegeben) {           operatorGegeben = false;
        berechne(linkerOperand,      }
        anzeigewert);
    }
    linkerOperand = anzeigewert;
    operatorGegeben = true;
}                                     }
```

- c) Teste dein Projekt ausführlich. Entwerfe eine Tabelle wie in Abbildung 2.16 bzw. 2.17!

Bemerkung:

Drückt der Benutzer nach der *Plus*-Taste sofort die *Gleich*-Taste, dann zeigt der Rechner den doppelten Wert an. Der Benutzer des Taschenrechners sollte mit einer Fehlermeldung auf dieses Fehlverhalten hingewiesen werden.

Zur Verdeutlichung des *Strategy*-Musters sollte jedoch dieser Entwurf genügen.

Zum Abschluss kannst du noch ein kleines Feature einbauen:

```
public String gibVersion()
{
    return "Version 1.0";
}

public String gibAutor()
{
    return "Ralph Henne";
}
```

Abbildung 2.18: Quelltext der Methoden *gibVersion()* und *gibAutor()*

Diese beiden Methoden werden abwechselnd verwendet, wenn der Benutzer auf die *?*-Taste drückt.

Somit ist der Taschenrechner nun modelliert. Jetzt kannst du dich der grafischen Oberfläche widmen. Hierzu wirst du ein weiteres Programmiermuster kennen lernen, das so genannte MVC-Muster.

2.4 Einführung in das MVC-Muster

Model-View-Controller (MVC, wörtlich etwa „Modell-Präsentation-Steuerung“) bezeichnet ein Architekturmuster zur Aufteilung von Softwaresystemen in die drei Einheiten: Datenmodell (engl. *Model*), Präsentation (engl. *View*) und Programmsteuerung (engl. *Controller*).

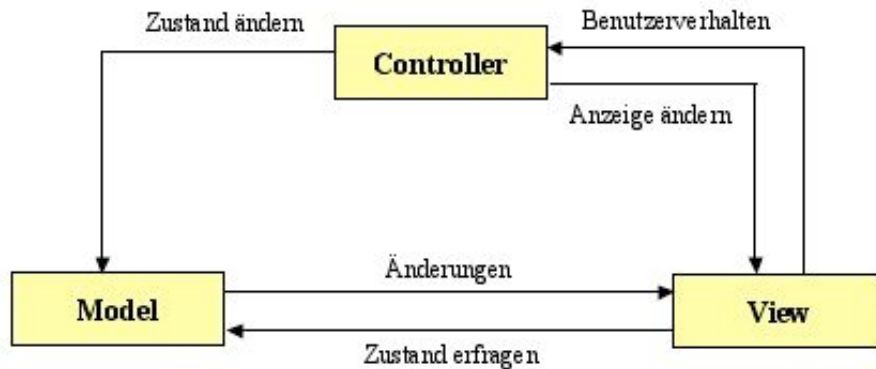


Abbildung 2.19: Das MVC-Muster

Ziel des Musters ist ein flexibles Programmdesign, das eine spätere Änderung oder Erweiterung erleichtern und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglichen soll.

Als **Model** wird die Komponente bezeichnet, die die Datenstruktur der Anwendung definiert. Das Model speichert die Daten und somit den Zustand der Anwendung und stellt die Methoden zur Änderung der Daten zur Verfügung. Das Model kennt weder den View noch den Controller.

Als **View** wird die Komponente bezeichnet, die die Daten des Models auf dem Bildschirm präsentiert, jedoch keine Programmlogik besitzt. Der Benutzer führt auf dem View die Aktionen aus, die durch den Controller an das Model weitergeleitet werden.

Der View kennt das Model, ist dort registriert und kann dessen Zustand erfragen.

Als **Controller** wird die Komponente bezeichnet, die die Ablaufsteuerung darstellt. Der Controller kennt die beiden anderen Komponenten, den View und das Model. Er reagiert auf die Interaktionen der Anwender in dem View und überprüft diese. Anschließend ruft er die jeweilige Methode des Models auf und verändert dessen Zustand.

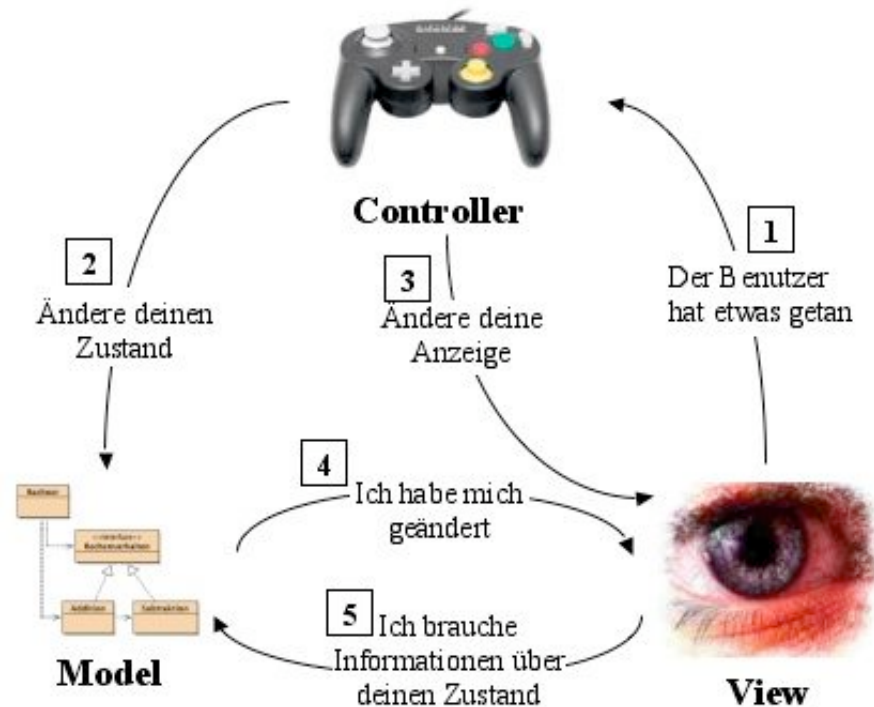


Abbildung 2.20: Beziehungen zwischen Model, View und Controller

1. ***Du als Beobachter interagierst mit dem View.***
Der View ist deine Sicht auf das Model. Wenn du also irgendetwas mit dem View machst, beispielsweise auf die *Plus*-Taste klickst, teilt der View dem Controller mit, was du getan hast. Es ist dann Aufgabe des Controllers, entsprechende Steuerungsmaßnahmen zu ergreifen.
2. ***Der Controller fordert das Model auf, seinen Zustand zu ändern.***
Der Controller nimmt deine Aktionen an und interpretiert sie. Wenn du auf einen Button klickst, muss der Controller herausfinden, was das bedeutet und wie das Model auf Grund dieser Aktion beeinflusst werden muss.
3. ***Der Controller kann auch den View auffordern, seinen Zustand zu ändern.***
Wenn der Controller eine Aktion vom View erhält, muss er den View auf Grund dessen eventuell auffordern, sich zu ändern. Beispielsweise könnte der Controller bestimmte Buttons oder Menüpunkte in der grafischen Benutzeroberfläche aktivieren oder deaktivieren.
4. ***Das Model benachrichtigt den View, wenn sich sein Zustand geändert hat.***
Wenn sich am Model etwas ändert – entweder auf Grund einer Aktion von dir (beispielsweise Eingeben einer Ziffer) oder einer internen Veränderung (beispielsweise Aufbau einer Zahl) -, meldet das Model dem View, dass sich sein Zustand geändert hat.
5. ***Der View fragt das Model nach seinem Zustand.***

Der View erhält den Zustand, den er anzeigt, direkt vom Modell. Wird er beispielsweise vom Modell benachrichtigt, dass er eine Rechenoperation durchgeführt hat, fragt der View das Modell nach dem aktuellen Ergebnis und zeigt dieses als Anzeigewert an. Der View kann das Modell auch nach dessen Zustand fragen, wenn er vom Controller aufgefordert wurde, die Ansicht zu verändern.

Nun wirst du die drei Komponenten im Rechnermodell zusammenbauen.

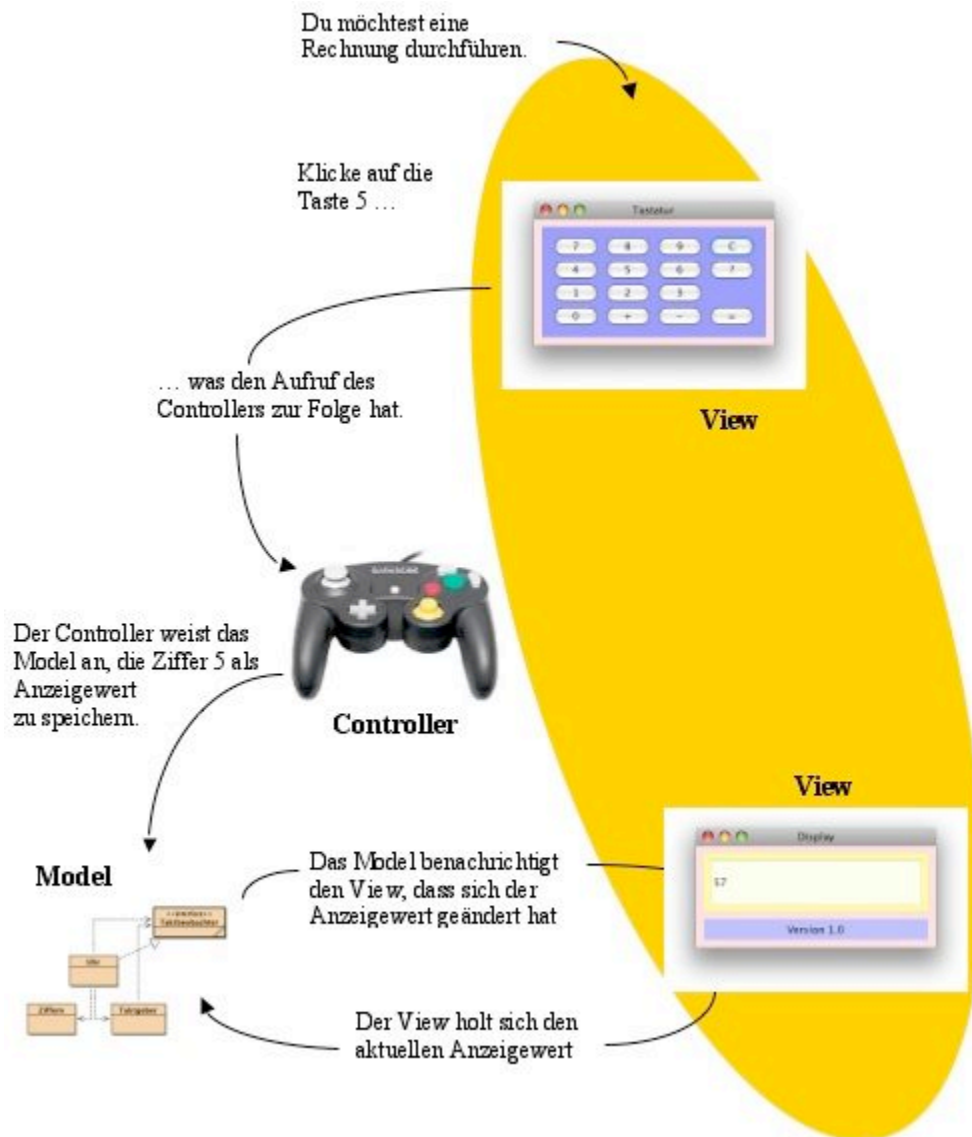


Abbildung 2.21: Der Taschenrechner im MVC-Muster

Wenn du den Text aufmerksam gelesen hast, wirst du bemerkt haben, dass das Model den View benachrichtigt, dass sich der Anzeigewert geändert hat, obwohl das Model weder den View noch den Controller kennt. Trotzdem zeigen alle drei Abbildungen einen Pfeil vom Model zum View.

Der View besteht aus mehreren Fenstern. In Abbildung 2.21 sind die beiden Fenster *TastaturView* und *DisplayView* dargestellt. Du kannst dir noch weitere Darstellungsmöglichkeiten ausdenken.

Um all diese verschiedenen Views zu verwalten, verwendet das Model das Observer-Muster. Es benötigt daher Methoden zur Registrierung von Objekten als Rechnerbeobachter und Methoden zum Verschicken von Nachrichten an diese Rechnerbeobachter. Das Model kennt also nur diese Rechnerbeobachter. Das Observer-Muster verlangt, dass *Rechnerbeobachter* ein Interface ist und dass alle Views, die die Anzeige benötigen, das Interface *Rechnerbeobachter* implementieren. Somit kennt das Model nicht die Komponenten des Views, sondern die Rechnerbeobachter.

2.5 MVC-Muster konkret

2.5.1 Die Klasse Rechner

Der Rechner muss nun mehrere Views überwachen, in diesem Beispiel den DisplayView, den TastaturView und vielleicht später noch weitere Viewkomponenten, beispielsweise Rechner mit Staffelwalze oder Sprossenrad oder was sich die Hersteller von Taschenrechner sonst noch so einfallen lassen.

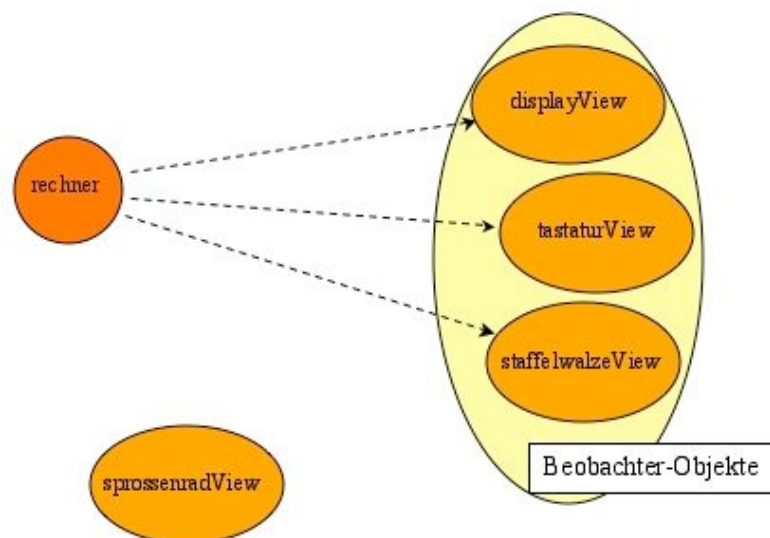


Abbildung 2.22: Das Observer-Muster

Nur der Rechner allein weiß genau, welches Ergebnis die Rechenoperation liefert. Die Beobachter-Objekte, also die *displayView*, *tastaturView* und *staffelwalzeView*, haben ein Abonnement beim Rechner. Das bedeutet, sie haben sich beim Rechner registriert, um Aktualisierungen zu erhalten, wenn also der Rechner ein neues Rechenergebnis ermittelt hat.

Das Objekt *sprossenradView* hat leider Pech gehabt. Da es kein Beobachter ist, wird es auch nicht informiert, wenn sich die Daten des Rechners ändern. Aber *sprossenradView* kann dem Rechner sagen, dass es Beobachter werden möchte. Dazu lässt es sich beim Rechner registrieren und wird somit in die Liste der Beobachter-Objekte aufgenommen. Nun wartet *sprossenradView* zusammen mit den anderen Views auf die nächste Benachrichtigung von Rechner, dass wieder ein neues Rechenergebnis zur Abholung bereit steht.

Genauso gut kann nun *staffelwalzeView* darum bitten, als Beobachter entfernt zu werden. Der Rechner berücksichtigt diese Anfrage und entfernt *staffelwalzeView* aus seiner Liste der Beobachter. Alle Beobachter, außer *staffelwalzeView*, der nun nicht mehr der Beobachtermenge angehört, erhalten weitere Benachrichtigungen vom Rechner.

Das Prinzip des Observer-Muster ist nun erklärt. Jetzt wirst du dieses Muster modellieren. Die Abbildung 2.23 zeigt die beiden Klassen *Rechner* und *DisplayView* mit ihren für das Observer-Muster wichtigen Datenfeldern und Methoden.

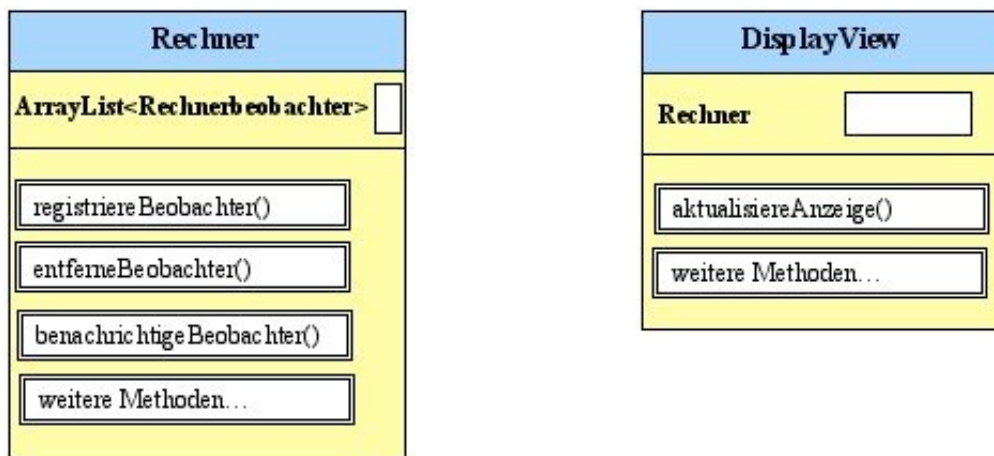


Abbildung 2.23: Die Klassen *Rechner* und *DisplayView*

In der Klasse *Rechner* muss du also eine *rechnerbeobachterliste* erzeugen und die Methoden *registriereBeobachter()*, *entferneBeobachter()* und *benachrichtigeBeobachter()* implementieren.

```

import java.util.ArrayList;

public class Rechner
{
    //Datenfelder
    //bereits vorhanden

    //beobachtende Objekte, die vom Rechner benachrichtigt werden
    private ArrayList<Rechnerbeobachter> rechnerbeobachterliste;
  
```

```
public Rechner()
{
    rechnerbeobachterliste = new ArrayList<Rechnerbeobachter>();

    //bereits vorhanden
}

private void setzeAnfangszustand()
{
    //bereits vorhanden
}

public void zifferGetippt(int ziffer)
{
    //bereits vorhanden
}

public void plus()
{
    //bereits vorhanden
}

public void minus()
{
    //Bereits vorhanden
}

public void gleich()
{
    //bereits vorhanden
}

public void clear()
{
    //bereits vorhanden
}

private void operatorAnwenden()
{
    //bereits vorhanden
}

private void berechne(int a, int b)
{
    //Bereits vorhanden
}

public int gibAnzeigewert()
{
    //bereits vorhanden
}
```

```
    }

    public String gibVersion()
    {
        //Bereits vorhanden
    }

    public String gibAutor()
    {
        //bereits vorhanden
    }

    private void setzeRechenverhalten(Rechenverhalten rechenart)
    {
        //bereits vorhanden
    }

    /** ===== Methoden des Observer-Musters ===== */

    public void registriereBeobachter(Rechnerbeobachter b)
    {
        rechnerbeobachterliste.add(b);
    }

    public void entferneBeobachter(Rechnerbeobachter b)
    {
        int i = rechnerbeobachterliste.indexOf(b);
        if (i >= 0) {
            rechnerbeobachterliste.remove(i);
        }
    }

    public void benachrichtigeRechnerbeobachter()
    {
        for (Rechnerbeobachter beobachter : rechnerbeobachterliste) {
            beobachter.aktualisiereAnzeige();
        }
    }
}
```

Abbildung 2.24: Quelltext der Klasse *Rechner* erweitert um das Observer-Muster

Übung 2.8:

Sichere dein bisheriges Projekt als *Taschenrechner03a*. Implementiere die Klasse *Rechner* aus [Abbildung 2.24](#).

Übung 2.9:

Überlege, in welchen Methoden der Klasse *Rechner* die Methode *benachrichtigeRechnerbeobachter()* aufgerufen werden muss.

Die Methoden *zifferGetippt()*, *gleich()*, *clear()* und *operatorAnwenden()* benötigen die Methode *benachrichtigeRechnerbeobachter()*.

2.5.2 Das Interface Rechnerbeobachter

Ganz so einfach ist diese Sache jedoch nicht. Wie in Abbildung 2.22 dargestellt gibt es verschiedene Views. Alle diese Views sehen ein bisschen anders aus, haben somit verschiedene Methoden und alle diese verschiedenen Views müssen von einer einzigen Beobachterliste verwaltet werden. Das Problem ist nun, dass diese Beobachterliste nur einen einzigen Typ sammeln kann.

Dieses Problem wird mit Hilfe eines Interfaces gelöst. Jeder View, der irgendwie das Rechenergebnis benötigt, muss dieses Interface implementieren. Somit verfügen all diese Views über die Methode *aktualisiereAnzeige()*, und können nun vom Rechner davon unterrichtet werden, dass ein neues Rechenergebnis vorliegt

```
public interface Rechnerbeobachter
{
    public void aktualisiereAnzeige();
}
```

Abbildung 2.25: Quelltext des Interfaces *Rechnerbeobachter*

Bemerkung:

Methoden in einem Interface sind öffentlich, *public* ist also nicht notwendig!

Betrachte ein Interface zunächst einmal als eine Vereinbarung, dass alle Klassen, die dieses Interface implementieren, auch die Methode *aktualisiereAnzeige()* besitzen. Der Rechner kann nun also sicher sein, dass alle seine Beobachter auch seine Benachrichtigung erhalten und damit etwas anfangen können. Er muss nichts darüber wissen, was die einzelnen Beobachter nun damit anfangen. Mal kann der Rechenwert in einem gewöhnlichen Display dargestellt werden, mal als bestimmte Positionen der Kugeln eines Abakus.

Weiterhin wird hier auch der Polymorphismus genutzt, indem auf ein Supertyp programmiert wird, damit das tatsächliche Laufzeitobjekt nicht im Quelltext festgeschrieben werden muss. Der Rechner sammelt also in seiner Beobachterliste nur noch Supertypen *Rechnerbeobachter*, er muss also die tatsächlichen Objekttypen (Display, Abakus) nicht kennen. Somit können an den Rechner alle möglichen Geräte zur Darstellung angeschlossen werden, wichtig ist nur, dass diese das Interface *Rechnerbeobachter* implementiert haben. Wird

nun ein neuer Beobachter hinzugefügt, so muss der Rechner nicht mehr verändert werden.

Rechner und Beobachter sind locker gebunden, sie können miteinander agieren, müssen aber nur wenige Kenntnisse voneinander besitzen.

Übung 2.10:

Implementiere das Interface *Rechnerbeobachter* im Projekt *Taschenrechner003a*.

2.5.3 Die Klasse *DisplayView*

Bemerkung:

Das Programmieren einer grafischen Benutzeroberfläche ist hier nicht das Thema. Die Gestaltungsmöglichkeiten mit Swing werden in *The JFC Swing Tutorial, Second Edition* von Kathy Walrath, in *Handbuch der Java-Programmierung* von Guido Krüger, u. a oder in ähnlichen Büchern beschrieben.



Abbildung 2.26: Grafische Oberfläche des Displays

Übung 2.11:

Hole vom Schulserver (oder von meiner Homepage) das Projekt *DisplayGUI*. Füge mit Hilfe von *Add Class from File...* die Klasse *DisplayView* in dein Projekt *Taschenrechner03a* ein. Studiere den Quelltext der Klasse *DisplayView*. Suche die Komponenten *infoL* und *anzeigeTF*. Überlege, wie diese Komponenten in das Fenster eingebaut wurden.

Wie in Abbildung 1.23 bereits dargestellt, besitzt die Klasse *DisplayView* also ein Rechner-Objekt. Somit kann das *DisplayView*-Objekt nun den Rechner bitten, dass Rechner dieses *DisplayView*-Objekt registriert und somit in seine Beobachterliste aufnimmt. Dies erfolgt im Konstruktor.

```
public DisplayView(Rechner rechner)
{
    this.rechner = rechner;
    rechner.registriereBeobachter((Rechnerbeobachter)this);

    versionAnzeigen = true;

    erzeugeDisplayView();
}
```

Abbildung 2.27: Quelltext des Konstruktors der Klasse *DisplayView*

Falls sich der Zustand vom Rechner geändert hat, beispielsweise eine weitere Rechenoperation liefert ein neues Ergebnis, so durchläuft der Rechner nun seine Beobachterliste, um alle registrierten Beobachter zu benachrichtigen, dass ein neues Rechenergebnis vorliegt. Allen Beobachter wird also gesagt, dass sie ihre Anzeige aktualisieren müssen. Und diese Beobachter können nun mit dieser Nachricht machen, was sie wollen, hier beispielsweise das neue Ergebnis in das *anzeigeTF* schreiben.

Übung 2.13:

Die Klasse *DisplayView* soll nun das Interface *Rechnerbeobachter* implementieren und ein Datenfeld *rechner* erhalten. Im Konstruktor soll *DisplayView* sich beim Rechner als *Rechnerbeobachter* registrieren. Zuletzt muss *DisplayView* noch die im Interface *Rechnerbeobachter* vertraglich festgelegte (noch leere) Methode *aktualisiereAnzeige()* erhalten.

Vergleiche hierzu auch die [Abbildung 2.23](#).

Hat sich also nun der Zustand des Rechners (des Models) geändert, so durchläuft der Rechner seine Liste aller *Rechnerbeobachter* (und somit auch alle Views) und benachrichtigt diese, dass ein neues Rechenergebnis vorliegt. Die Views fragen nun beim Model nach diesem aktuellen Rechenergebnis.

Übung 2.14:

Betrachte die [Abbildungen 2.20](#) und [2.21](#) und suche die oben beschriebenen Datenflüsse.

Der Rechner liefert das Rechenergebnis, das in das entsprechende Textfeld geschrieben.

```
public void aktualisiereAnzeige()
{
    String anzeige = erstelleAnzeige();
    anzeigeTF.setText(anzeige);
}
```

```
}  
  
private String erstelleAnzeige()  
{  
    String ergebnisanzeige;  
  
    int ergebnis = rechner.gibAnzeigewert();  
    ergebnisanzeige = "" + ergebnis;  
    return ergebnisanzeige;  
}
```

Übung 2.15:

Implementiere die beiden Methoden *aktualisiereAnzeige()* und *erstelleAnzeige()* in der Klasse *DisplayView*.

Teste nun dein Projekt *Taschenrechner03a*. Erzeuge zuerst einen Rechner und dann *DisplayView*. Führe anschließend folgende Rechenaufgaben durch:

- $5 + 7 = [12]$
- $57 + 4 = [61]$
- $5 + 7 + [12] 4 = [16]$
- $5 + 7 = [12] + 4 = [16]$

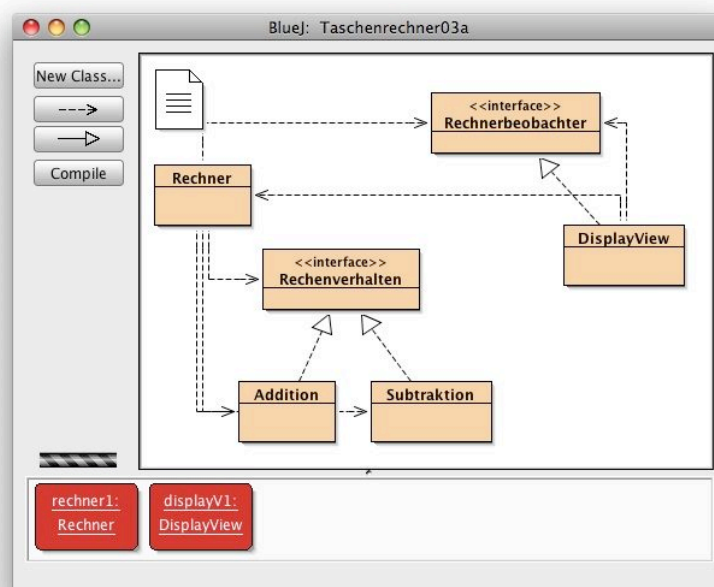


Abbildung 2.28: Klassendiagramm des Projekt *Taschenrechner03a*

In der Übung 2.15 wirst du festgestellt haben, dass du zuerst ein Rechner-Objekt erzeugen musst. Damit sich der *DisplayView* als Beobachter bei beim Rechner

registrieren lassen kann, muss das `DisplayView`-Objekt den Rechner als Parameter übergeben werden.

2.5.4 Die Klasse `TastaturView`

Die grafische Oberfläche zeigt zwar nun die Rechenergebnisse, aber ist noch nicht für Benutzereingaben vorbereitet. Hierzu wird der View erweitert durch ein neues Fenster, dem `TastaturView`.

Mit Hilfe der Zifferntasten kann der Benutzer die Zahlen eingeben. Die Rechenoperationen werden mit der *Plus*-, *Minus*- bzw. *Gleich*-Taste durchgeführt. Die *Clear*-taste setzt die Anzeige wieder zurück auf 0 und die Versionsnummer bzw. der Autor wird mit der *?*-Taste gezeigt.



Abbildung 2.29: Die grafische Oberfläche zur Steuerung des Rechners

Übung 2.16:

Sichere dein bisheriges Projekt als *Taschenrechner03b*.

Hole vom Schulserver (oder von meiner Homepage) das Projekt *TastaturGUI*. Füge mit Hilfe von *Add Class from File...* die Klasse *TastaturView* in dein Projekt *Taschenrechner03b* ein. Studiere den Quelltext der Klasse *TastaturView*.

- Wie wird eine einzelne Taste erzeugt?
- Überlege, wie der leere Bereich zwischen der *?*-Taste und der *Gleich*-Taste entsteht.

Der Benutzer hat also nun eine Zifferntaste gedrückt. Der `TastaturView` delegiert diese Benutzeraktion an den Controller. Der Controller ist das einzige

Objekt, das weiß, wie man mit den Aktionen des Benutzers umgeht. Der Controller entscheidet,

- a) ob er dem Model mitteilen soll, dass es seinen Zustand ändern soll,
- b) ob er dem View mitteilen soll, dass er seine Anzeige ändern soll.

Übung 2.17:

Betrachte die Abbildungen 2.20 und 2.21 und suche die oben beschriebenen Datenflüsse.

Die Benutzeraktionen werden vom TastaturView direkt an den Controller weitergeleitet. Die Klasse *TastaturView* muss deshalb ein Datenfeld *controller* erhalten. Der Konstruktor erhält als Parameter somit ein Controller-Objekt, das in dem Datenfeld *controller* gehalten wird

```
private Controller controller;  
  
public TastaturView(Controller controller)  
{  
    this.controller = controller;  
  
    erzeugeTastaturView();  
}
```

Abbildung 2.30: Datenfeld und Konstruktor der Klasse *TastaturView*

Klickt der Benutzer nun auf eine beliebige Taste, so wird dies dem Controller mitgeteilt.

```
private void tasteAction(ActionEvent event)  
{  
    int ziffer = 0;  
  
    String taste = event.getActionCommand();  
    if(taste.equals("0") ||  
       taste.equals("1") ||  
       taste.equals("2") ||  
       taste.equals("3") ||  
       taste.equals("4") ||  
       taste.equals("5") ||  
       taste.equals("6") ||  
       taste.equals("7") ||  
       taste.equals("8") ||  
       taste.equals("9")) {  
        ziffer = Integer.parseInt(taste);  
        controller.behandleZiffer(ziffer);  
    }  
    else if (taste.equals("+")) {
```

```
        controller.addiere();
    }
    else if (taste.equals("-")) {
        controller.subtrahiere();
    }
    else if (taste.equals("=")) {
        controller.istgleich();
    }
    else if (taste.equals("C")) {
        controller.reset();
    }
    else if (taste.equals("?")) {
        controller.informiere();
    }
}
```

Abbildung 2.31: Mitteilungen von Benutzereingaben an den Controller in der Klasse *TastaturView*

Übung 2.18:

Implementiere den in den Abbildungen 2.30 und 2.31 dargestellten Quelltext. Suche in den Abbildungen 2.20 und 2.21 den hier programmierten Schritt.

2.5.5 Die Kasse Controller

Jetzt musst du dich noch um den Controller kümmern.

Eine der Aufgabe des Controllers ist, auf Grund der Benutzeraktion im View zu entscheiden, welche Daten im Model geändert werden müssen. (Später wird der Controller auch noch den View ändern.) Der Controller darf jedoch nicht die Daten selbst manipulieren, dies muss das Model machen.

```
import java.awt.*;
import java.awt.event.*;

public class Controller
{
    private Rechner rechner;
    private TastaturView tastaturView;
    private DisplayView displayView;

    public Controller(Rechner rechner)
    {
        this.rechner = rechner;

        tastaturView = new TastaturView(this);
        displayView = new DisplayView(rechner);
    }
}
```

```
    }

    public void behandleZiffer(int ziffer)
    {
        rechner.zifferGetippt(ziffer);
    }

    public void addiere()
    {
        //leer
    }

    public void subtrahiere()
    {
        //leer
    }

    public void istgleich()
    {
        //leer
    }

    public void reset()
    {
        //leer
    }

    public void informiere()
    {
        //leer
    }
}
```

Wie bereits in der Einführung in das MVC-Muster beschrieben, kennt der Controller die beiden anderen Komponenten, den View und das Model. Er ist also dasjenige Objekt, das alles zusammenhält. Deswegen erhält der Konstruktor das Model (den Rechner) als Argument und erzeugt den View. Der View besteht jetzt bereits schon aus zwei Teilen,

1. dem TastaturView, der den Controller benötigt, der ja die Benutzereingaben delegieren muss,
2. dem DisplayView, der das Model (den Rechner) benötigt, um dessen Zustand abzufragen.

Hat der Anwender in der grafischen Benutzeroberfläche beispielsweise auf eine Zifferntaste gedrückt, meldet der *TastaturView* dies dem *Controller* mit der Methode *controller.behandleZiffer()*. Der *Controller* teilt nun dem *Rechner* mit, dass eine Ziffer getippt wurde. Der *Rechner* weiß dann schon, was er mit dieser Ziffer anfangen soll und benachrichtigt anschließend den *DisplayView*

(eigentlich alle Rechnerbeobachter), dass sich sein Zustand (neues Rechenergebnis) geändert hat. Der *DisplayView* fragt nun beim *Rechner* nach Rechenergebnis, erstellt daraus eine Anzeige und schreibt diese in das entsprechende Textfeld.

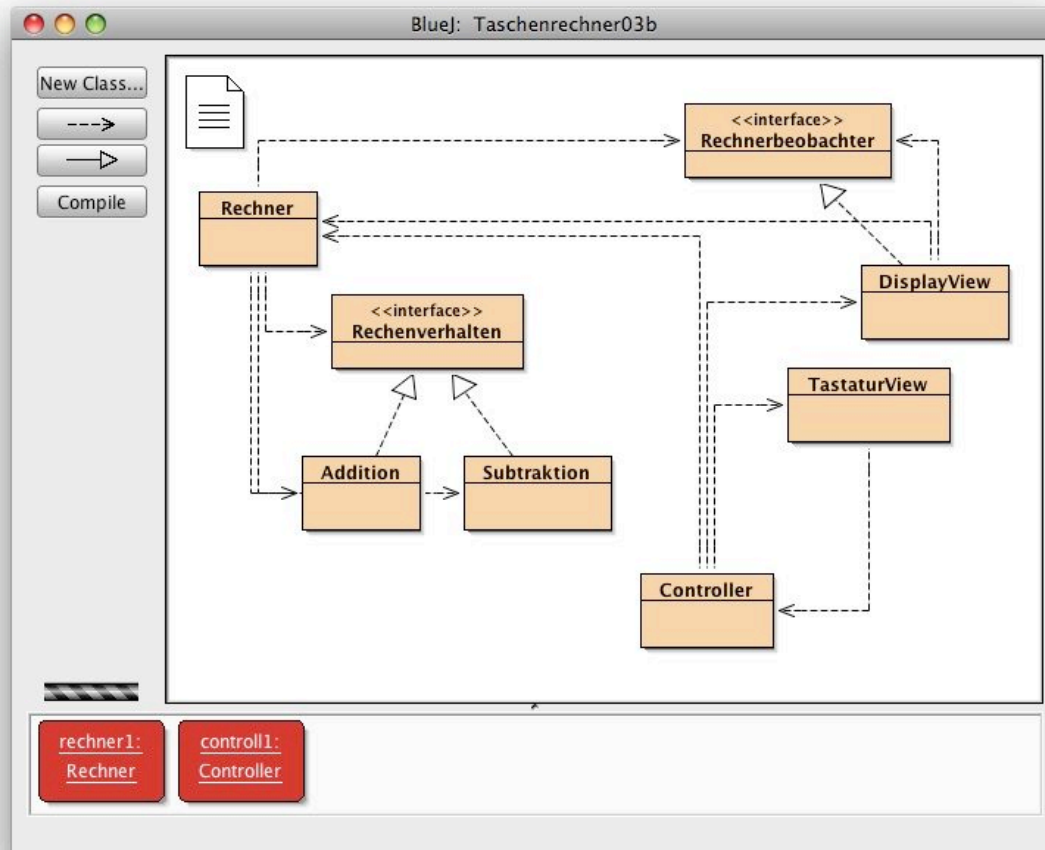


Abbildung 2.32: Klassendiagramm des Projekts *Taschenrechner03b*

Übung 2.19:

Versuche die oben beschriebenen Datenflüsse im Klassendiagramm des Projekt *Taschenrechner03b* nachzuvollziehen. Vergleiche deine Gedanken auch mit den Beziehungen zwischen Model, View und Controller in den Abbildungen 2.20 und 2.21.

Übung 2.20:

Implementiere die Klasse *Controller* in deinem Projekt *Taschenrechner03b*. Vervollständige nun auch die Methoden *addiere()*, *subtrahiere()*, *istgleich()* und *reset()*. Teste dein Projekt, indem du nacheinander ein *Rechner*-Objekt und ein

Controller-Objekt erzeugt. Führe anschließend folgende Rechenaufgaben durch:

- a) $5 + 7 = [12]$
- b) $57 + 4 = [61]$
- c) $5 + 7 + [12] 4 = [16]$
- d) $5 + 7 = [12] + 4 = [16]$

2.5.6 Die ?-Taste

Übung 2.21:

Betrachte das MVC-Muster in Abbildung 2.20. Überlege, welche der dort dargestellten Datenflüsse im bisherigen Projekt *Taschenrechner03b* noch nicht berücksichtigt worden ist?

Klickt der Benutzer auf die ?-Taste, so erhält der *Controller* vom *TastaturView* die Aufforderung, den *DisplayView* zu ändern. Im *DisplayView* muss das *infoL* wechseln zwischen den Texten *Version 1.0* und *Ralph Henne*. Der Controller erhält eine Aktion vom View (hier vom *TastaturView*) und fordert anschließend nun direkt den View (hier den *DisplayView*) auf, sich zu verändern. Der Zustand des Models ändert sich dabei. Es wird also die in der Abbildung 2.33 rot dargestellte Aktion durchgeführt.

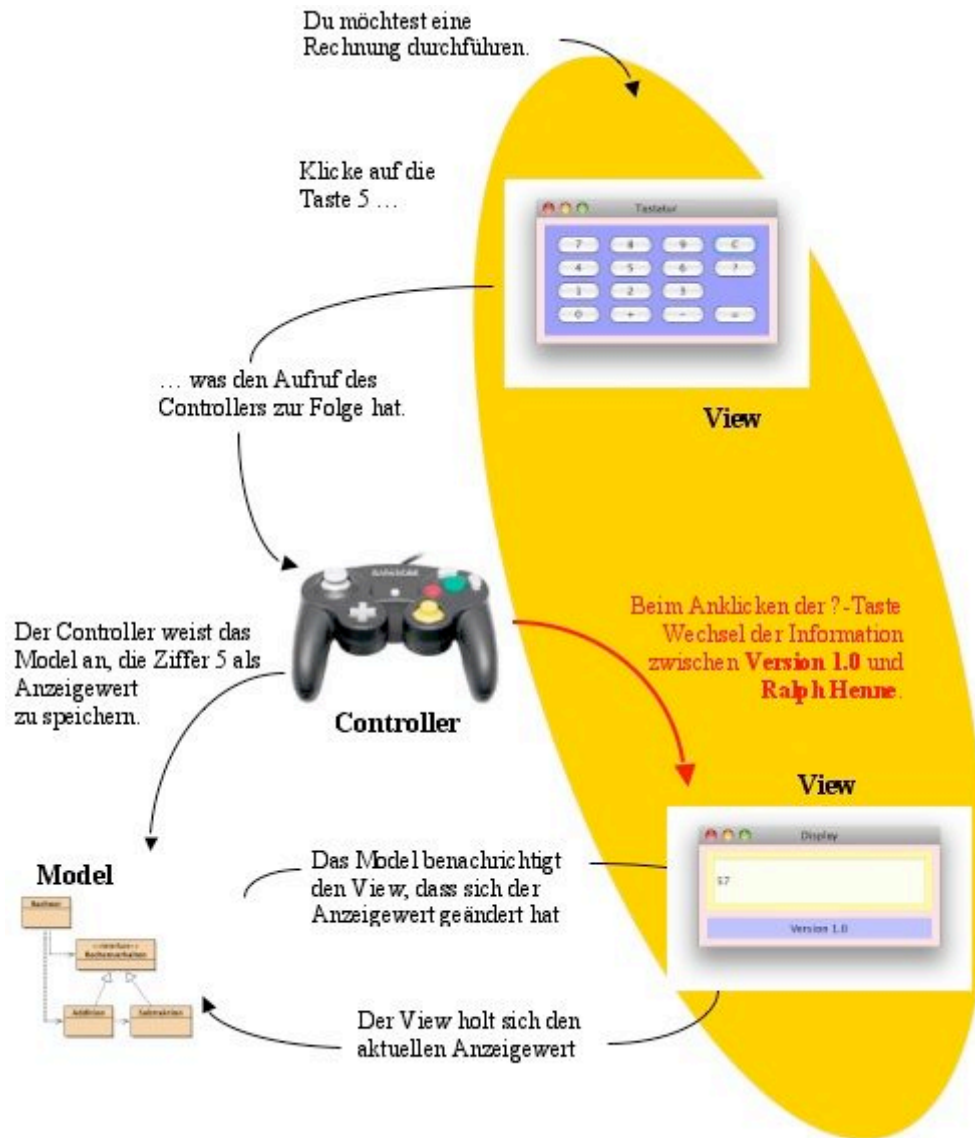


Abbildung 2.33: Der Controller weist den View an, sich zu ändern

Je nach Wert des Boolean *versionAnzeigen* holt sich der *DisplayView* vom *Rechner* den Autor bzw. die Version.

```
public void infoZeigen()
{
    if (versionAnzeigen) {
        infoL.setText(rechner.gibAutor());
    }
    else {
        infoL.setText(rechner.gibVersion());
    }
    versionAnzeigen = !versionAnzeigen;
}
```

Abbildung 2.34: Quelltext der Methode *infoZeigen()* in der Klasse *DisplayView*

Drückt also der Benutzer im *TastaturView* die ?-Taste, so wird die Methode *controller.informiere()* aufgerufen. Der Controller befiehlt mit der Methode *displayView.infoZeigen()* direkt dem *DisplayView*, seinen Schriftzug zu wechseln

Übung 2.22:

Implementiere die Methode *infoZeigen()* in der Klasse *DisplayView* und vervollständige die Methode *informiere()* in der Klasse *Controller*.

Sorge auch dafür, dass gleich zu Beginn im *infoL* die Versionsnummer angezeigt wird. Teste dein Projekt durch mehrmaliges Drücken der ?-Taste.

Bemerkung:

Der View und der Controller stellen das Strategy-Muster dar. Der View ist nur für die Sicht zuständig, die Entscheidungen über sein Verhalten delegiert er an den Controller. Der Controller ist somit das Verhalten (die Strategie) des Views. Wird für den View ein anderes Verhalten gewünscht, muss der Controller ausgetauscht werden.

Außerdem bleibt durch die Verwendung des Strategy-Musters der View entkoppelt vom Model. Nur der Controller weiß, wie man mit Benutzeraktionen umgeht.

Streng genommen verlangt das Strategy-Muster ein Interface, z.B.

ControllerInterface, an das alle möglichen Controller angeschlossen werden können. So ähnlich wie beim Rechenverhalten oder bei den Beobachtern müssten alle Controller dieses Interface implementieren. Ich glaube, da in diesem Beispiel nur ein einziger Controller verwendet wird, dass man auf ein *ControllerInterface* verzichten kann.

Genauerer kann man jedoch im eingangs erwähnten Buch *Entwurfsmuster von Kopf bis Fuß* nachlesen.

2.6 Zusammenfassung des Views

Die beiden Teile des Views – die Sicht auf das Model (*DisplayView*) und die Steuerungselemente für die Benutzerschnittstelle (*TastaturView*) werden in zwei getrennten Fenstern angezeigt. Deswegen wurden diese beiden Teile in zwei verschiedene Klassen gelegt. Das Klassendiagramm vereinfacht sich jedoch erheblich, wenn diese in einer einzigen Klasse *RechnerView* stecken. Dies wird nun aber kein großes Problem mehr darstellen, weil die beiden Klassen *DisplayView* und *TastaturView* für die geplante Zusammenfassung schon weit gehend vorbereitet sind.

Bemerkung:

Im Vergleich zum Projekt *Uhr* aus Kapitel 1 werden hier die beiden Klassen für die grafische Benutzeroberfläche nicht direkt von *JFrame* abgeleitet. Vielmehr wird im Konstruktor der entsprechenden View-Klassen ein *JFrame*-Objekt erzeugt. Diese Aufgabe wird dann später die Klasse *Controller* übernehmen, die ja u. A. für die Erzeugung des Views zuständig ist.

Die Klasse *RechnerView* implementiert das Interface *Rechnerbeobachter* und übernimmt alle Datenfelder, die in den beiden Klassen *DisplayView* und *TastaturView* bereits eingeführt wurden. Der Konstruktor erhält als Parameter sowohl ein Controller-Objekt als auch ein Rechner-Objekt.

```
public RechnerView(Controller controller, Rechner rechner)
{
    this.controller = controller;
    this.rechner = rechner;
    rechner.registriereBeobachter((Rechnerbeobachter)this);

    versionAnzeigen = true;
}
```

Abbildung 2.35: Konstruktor der Klasse *RechnerView*

Die Erzeugung der entsprechenden Frames geschieht nicht mehr hier im Konstruktor, diese Aufgabe übernimmt jetzt direkt der Controller.

Übung 2.23:

Sichere dein bisheriges Projekt als *Taschenrechner04a*. Du wirst nun aus den beiden Klassen *DisplayView* und *TastaturView* eine neue Klasse *RechnerView* zusammenstellen.

- Erzeuge eine neue Klasse *RechnerView*. Implementiere alle Datenfelder, die auch in den Klassen *DisplayView* und *TastaturView* verwendet wurden.
- Implementiere den Konstruktor aus Abbildung 2.35 in der Klasse *RechnerView*.
- Implementiere (aus der Klasse *DisplayView*) die Methoden *erzeugeDisplayView()*, *anzeigePERstellen()* und *informationPERstellen()* in der Klasse *RechnerView*.
- Implementiere die sonstigen wichtigen Methoden im DisplayFrame *erstelleAnzeige()* und *infoZeigen()* in der Klasse *RechnerView*.
- Implementiere (aus der Klasse *TastaturView*) die Methoden

erzeugeTastaturView(), *tastenPErstellen()* und *tasteHinzufuegen()* in der Klasse *RechnerView*.

- f) Implementiere die Aktions- Methode im TastaturFrame *tasteAction()* in der Klasse *RechnerView*.
- g) Entferne nun die beiden Klassen *DisplayView* und *TastaturView* aus dem Projekt *Taschenrechner04a*.

Nun musst du, wie bereits oben angedeutet, einige Veränderungen in der Klasse *Controller* durchführen.

```
private Rechner rechner;  
private RechnerView rechnerView;  
  
public Controller(Rechner rechner)  
{  
    this.rechner = rechner;  
    rechnerView = new RechnerView(this, rechner);  
  
    rechnerView.erzeugeDisplayView();  
    rechnerView.erzeugeTastaturView();  
}
```

Abbildung 2.36: Neue Datenfelder und Konstruktor in der Klasse *Controller*

Der Konstruktor der Klasse *Controller* erzeugt nun ein *RechnerView*-Objekt, das seinerseits den *DisplayView* und *TastaturView* erzeugt.

Übung 2.24:

- a) Ersetze die beiden Datenfelder *diplayView* und *tastaturView* durch das Datenfeld *rechnerView* in der Klasse *Controller*.
- b) Implementiere den Konstruktor aus Abbildung 2.36 in der Klasse *Controller*.
- c) Verbessere die Methode *informiere()* in der Klasse *Controller*.

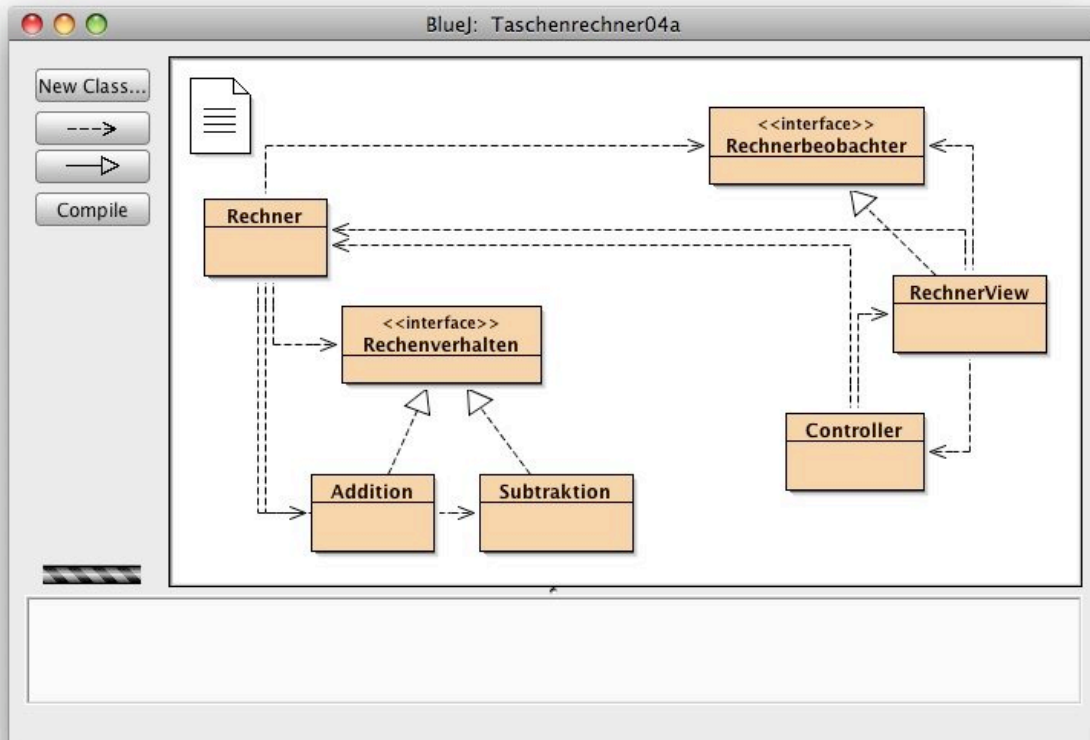


Abbildung 2.37: Klassendiagramm des Projekts *Taschenrechner04a*

2.7 Abschlussarbeiten

Du hast nun alles, was du brauchst: ein Model, einen View und einen Controller. Und all dies hast du in **BlueJ** manuell zu einem MVC-Muster zusammengesetzt, so dass sie gut miteinander arbeiten.

Zum Schluss fehlt noch eine Klasse *Simulation*, die dies automatisch für dich erledigt. Für eine stand-alone-Applikation benötigst du zudem noch die Methode *main()*.

```
public class Simulation
{
    public static void main (String[] args)
    {
        Rechner rechner = new Rechner();
        Controller controller = new Controller(rechner);
    }
}
```

Abbildung 2.38: Die Klasse *Simulation*

Übung 1.27:

Sichere dein bisheriges Projekt als *Taschenrechner04b*. Implementiere die Klasse *Simulation*.

2.8 Ausblick

Du wirst festgestellt haben, dass dieses Projekt noch einige Mängel aufweist:

1. Drücke auf die 5-Taste, dann auf die *Plus*-Taste und anschließend auf die *IstGleich*-Taste.
2. Bei Fehlbedienung des Benutzer sollte ein entsprechender Hinweis erzeugt werden

Hier gibt es noch einige Möglichkeiten, das Projekt zu verbessern.

Man könnte auch die Funktionalität des Taschenrechners vergrößern. Folgende Beispiele möchte ich nennen:

1. Du könntest weitere Rechenoperationen mit zwei Parametern hinzufügen, beispielsweise die Multiplikation oder Division.
2. Du könntest weitere Rechenoperationen mit einem Parametern hinzufügen, beispielsweise Quadrieren oder Wurzelziehen.
3. Du könntest das Rechnen mit negativen Zahlen oder Dezimalzahlen ermöglichen.