Inhaltsverzeichnis

1. Entwurfs-Muster / Design Pattern

1.1. Ziele

☑ Wiederholung: OOP

☐ Vererbung/abstrakte Klassen/Ploymorphismus

☑ Design Pattern kennen und anwenden können

☐ Observer-Pattern

☐ MVC-Pattern

☑ Anwendungen bei Java/Swing und CPP

☑ http://www.modeliosoft.com/modules/pattern-designer.html

1.2. Warum Entwurfs-Muster

Bereitstellung **bewährter, vorgefertigter Lösungsstrukturen** für wiederkehrende

Entwurfsprobleme

1.3. Was muss ich als OOP - ProgrammiererIn wissen

Entwurfsmuster sind Programmcode-Teile, die im Sinne der Vererbung realisiert sind.

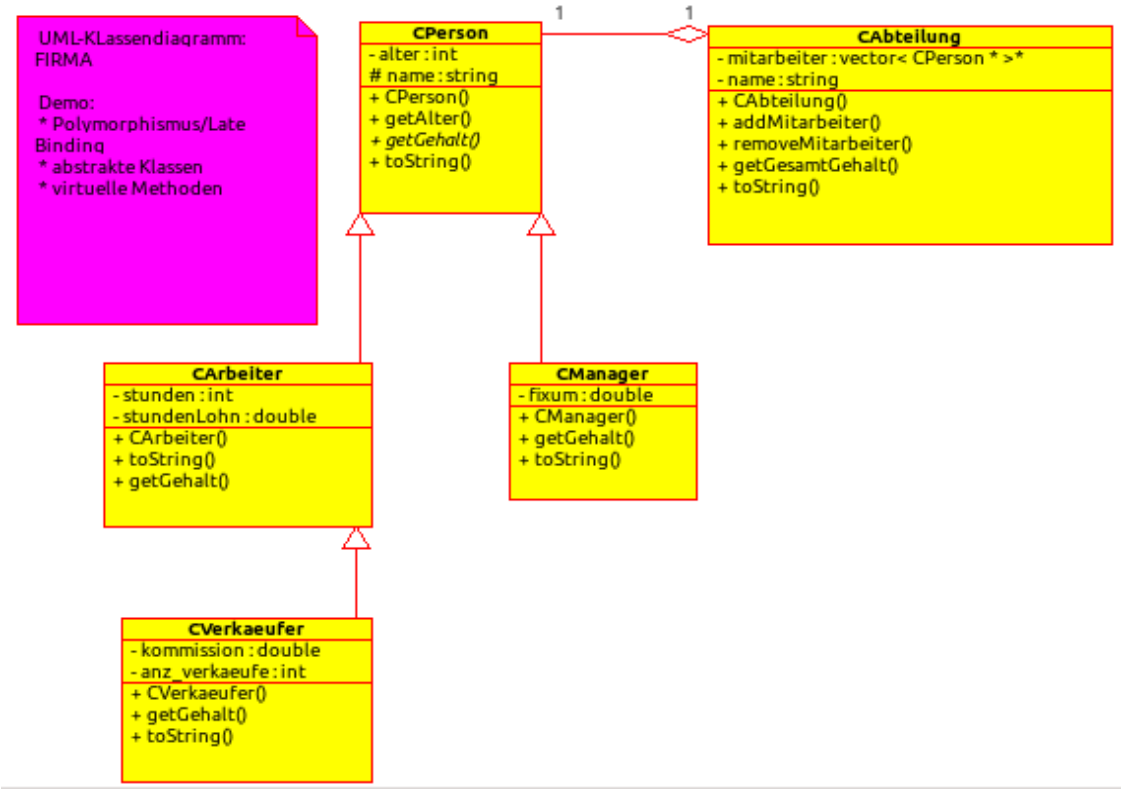
D.h. Die Entwurfsmuster sind meist als Oberklassen definiert und können im eigenen Programm geerbt werden. Dabei ist besonders wichtig, zu wissen, wie dies alles funktioniert.

1.3.1. Aufgabe: CFIRMA: Vererbung, Polymorphismus und late binding (u)

Studieren Sie z.B: im Kurs CPP die Lerneinheit: 02-oop1-klasse-vererbung

und erstellen Sie das darin befindliche Projekt: CFIRMA.

Hinweis: verwenden Sie als Vorlage: 02-ueben/uCFIRMA.zip



1.3.2. Aufgabe: Fragen zur Vererbung, ...

Beantworten Sie folgende Fragen:

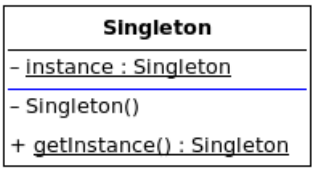
* protected ?
* Welcher Kontruktor wird zuerst fertig ausgeführt:
  der oberklasse
  der unterklasse
* Was ist ein base initializer ? Wozu wird er benötigt ?
* Wozu werden base class pointers verwendet ?
* Warum sollte eine Klasse, die virtuelle Methoden definiert, unbedingt einen virtuellen Destruktor definieren ?
* Worauf ist bei der Verwendung des Schlüsselwortes protected zu achten ?
* Was ist eine abstrakte Klasse?
* Warum gibt es eigentlich Vererbung, base class pointer?

1.4. SINGELTON-MUSTER

http://de.wikipedia.org/wiki/Singleton_%28Entwurfsmuster%29

Das Singleton (auch Einzelstück genannt) ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster (engl. Creational Patterns).

Es verhindert, dass von einer Klasse mehr als ein Objekt erzeugt werden kann. Dieses Einzelstück ist darüber hinaus üblicherweise global verfügbar.



☑ Verwendung wenn

- ☐ nur ein Objekt zu einer Klasse existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird oder
- ☐ das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

☑ Anwendungsbeispiele sind:

- ☐ ein zentrales Protokoll-Objekt, das Ausgaben in eine Datei schreibt.
- ☐ Druckaufträge, die zu einem Drucker gesendet werden, sollen nur in einen einzigen Puffer geschrieben werden.

☑ Vorteile: Das Muster bietet eine Verbesserung gegenüber globalen Variablen:

- ☐ Zugriffskontrolle kann realisiert werden.
- ☐ Das Einzelstück kann durch Unterklassenbildung spezialisiert werden.
- ☐ Welche Unterklasse verwendet werden soll, kann zur Laufzeit entschieden werden.
- ☐ Die Einzelinstanz muss nur erzeugt werden, wenn sie benötigt wird.
- ☐ Sollten später mehrere Objekte benötigt werden, ist eine Änderung leichter möglich als bei globalen Variablen.

Datei: Singleton.java

```
public final class Singleton
{
    /**
     * Privates Klassenattribut,
     * wird beim erstmaligen Gebrauch (nicht beim Laden)
     * der Klasse erzeugt
     */
    private static Singleton instance;

    /** Konstruktor ist privat, Klasse darf nicht von außen
       instanziiert werden. */
    private Singleton() {}

    /**
     * Statische Methode „getInstance()“ liefert die
     * einzige Instanz der Klasse zurück.
     * Ist synchronisiert und somit thread-sicher.
     */
    public synchronized static Singleton getInstance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Hier ein Beispiel in C++:
private sind:

- Default-Konstruktor
- Kopierkonstruktor
- Zuweisungsoperator

```
#include <ctime>
#include <iostream>
using namespace std;
```

```
class Singleton {
private:
     // you cannot create an object
     Singleton() {}

     // you cannot make a copy of an object
     Singleton(const Singleton&) {}

     // you cannot make a copy by assign-operator
     Singleton& operator=(const Singleton&) { return *this; }

     ~Singleton() {}

public:
     static Singleton& getInstance(){
          static Singleton instance;
          return instance;
     }

     // as an example: a Logger Object as a Singleton
     void log(int level, string msg){
          time_t second;
          struct tm *atime;
          char sTime[80];

          time(&second);
          atime= localtime(&second);
          strftime(sTime, 80, "%c", atime);

          cout << sTime << ":" <<level << ":" << msg<<endl;
     }
};



int main() {
     // create the one and only one singleton object.
     // its created within getInstance(), that returns
     // reference to the singleton object

     Singleton& logger= Singleton::getInstance();


     // Addresses are all the same, because of there is
     // only one and only one singleton object

     cout << "\ndemonstration of singleton pattern: " << endl;
     cout << "   3 addresses should have the same value:" <<endl;
     cout << "   "<< hex << &logger << endl;
     cout << "   "<< hex << &Singleton::getInstance() << endl;
     cout << "   "<< hex << &Singleton::getInstance() << endl;

     // you can use/reference the singleton object
```

```
    Singleton::getInstance().log(0, "this is my first log entry");

    logger.log(1, "here is my second log entry");


    return 0;
}
```

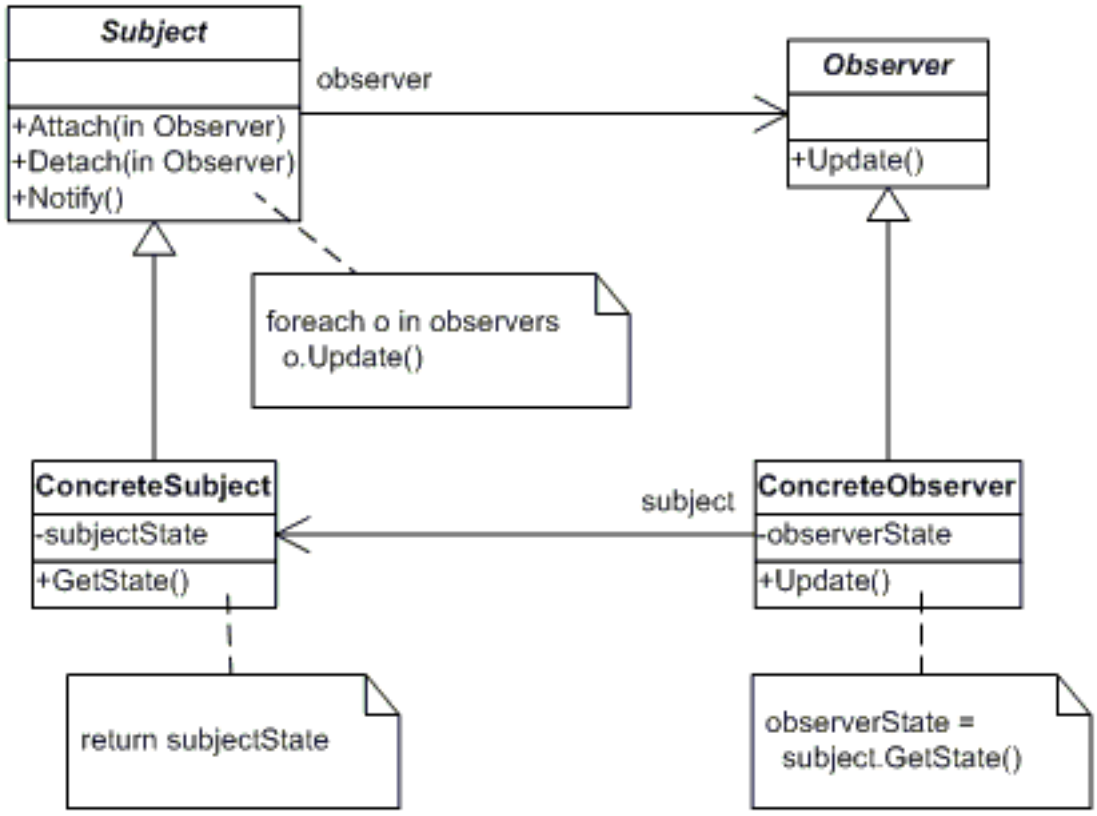
1.5. BEOBACHTER-MUSTER (Observer Pattern)

☑ **Ziel:** Observer (Beobachter, Listener) Pattern

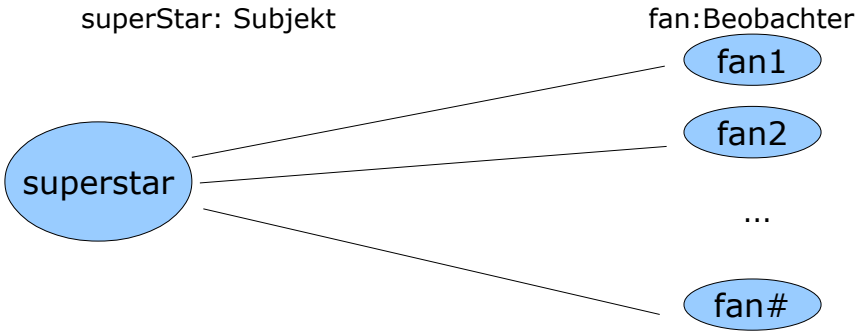
Es dient **zur Weitergabe von Änderungen bei einem Objekt an alle von diesem Objekt abhängige Objekte.**

Dieses Entwurfsmuster ist auch unter dem Namen **publish-subscribe** bekannt, frei übersetzt „veröffentlichen und abonnieren".

1.5.1. UML-Klassendiagramm: Beobachter-Muster



☑ Namenserklärung:



☑ Die Klasse Subjekt
Objekte dieser Klassen werden beobachtet.
In unserem Beispiel, wollen wir dieses Objekt „superstar" nennen.

☑ Die Klasse Beobachter/Observer
Objekte dieser Klasse sollen über Änderungen beim „superstar-Objekt" informiert werden.
In unserem Beispiel, wollen wir diese Objekte „fan" nennen.

Ablauf:

☑ Klasse Subjekt:
Jedes Objekt der Klasse **Subjekt führt eine Liste von Beobachtern**, welche bei Veränderungen im Zustand des Subjekts informiert werden sollen.

☑ Klasse Subjekt:
Die Methode **Attach**() / addObserver() fügt Beobachter in die Liste ein.

Die Methode **Detach**()/deleteObserver() entfernt Beobachter aus der Liste.

☑ Klasse Subjekt:
Die Methode **Notify**()/notifyStateChange()
Nach jeder Veränderung beim superStar-Objekt (Klasse Subjekt) schickt es sich selbst die Botschaft this->Notify().

Diese iteriert die Liste der Beobachter und ruft bei jedem Beobachter die Botschaft **Update**() auf.

Das benachrichtigende Objekt (superStar) wird als Parameter mitgegeben. Ggf. können weitere Informationen (z.B. die Art des eingetretenen Ereignisses) als Parameter mitgegeben werden.

☑ Klasse Beobachter:
Jeder benachrichtigte Beobachter (fan) reagiert, indem er beim benachrichtigenden Gegenstandsobjekt (superStar) mit **getXXX()-Botschaften** die ihn interessierenden Informationen abruft.

1.6. CPP-Projekt: BEOBACHTER-MUSTER in cpp programmieren

Wir wollen das Beobachter-Muster (Design Pattern: Observer/Observable) implementieren.
Dazu gehen wir wie folgt vor:

Namenserklärung (in Anlehnung an Java):
statt dem Klassennamen Subjekt, wollen wir den Namen **Observable** (superstar) verwenden.
statt dem Klassennamen Beobachter, wollen wir den Namen **Observer** (fan) verwenden.

1.6.1. Aufgabe: Die Basisklasse: Observable (u)

Projekt: ws-qt/seng/design-pattern/Beobachter-Muster

observable.h, observable.cpp

Erstellen Sie die Basisklasse: Observable:
☑ member:
vector< Observer* > observers;

☑ methoden:
bool addObserver(Observer* observer);
bool removeObserver(Observer* observer);

void notifyStateChange();
// ruft bei allen eingetragenen Observer-Objekten die Methode update() auf

1.6.2. Aufgabe: Die Basisklasse: Observer (u)

Projekt: ws-qt/seng/design-pattern/Beobachter-Muster

observer.h, observer.cpp

Erstellen Sie die Basisklasse: Observer
☑ methode:
void update();
// wird bei einer Änderung im Observable von dort aus aufgerufen.

main.cpp

Wir wollen nun ein kleines Testprogramm schreiben, um die beiden Klassen zu testen.

```
#include "observer.h"
#include "observable.h"
#include <iostream>
using namespace std;
int main() {
	cout << "Demo: Beobachter-Muster!!! V1" << endl;
```

```
        Observer* fan1= new Observer();
        Observer* fan2= new Observer();
        Observable* bayern= new Observable();

        bayern->addObserver(fan1);
        bayern->addObserver(fan2);

        cout<< "Hey, bei Bayern tut sich was....................."<<endl;
        bayern->notifyStateChange();
        return 0;
}
```

Kontrolle:
Folgender Output sollte erzeugt werden:

```
Demo: Beobachter-Muster!!! V1
Hey, bei Bayern tut sich was.....................
Observer: update() wurde aktiviert ..........
Observer: update() wurde aktiviert ..........
```

1.7. Aufgabe: BEOBACHTER-MUSTER-FUSSBALL (cpp, u)

Projekt: ws-qt/seng/design-pattern/Beobachter-Muster-Fussball

Wir wollen nun das obige Beobachter-Muster anwenden.

Die Situation:

Fußballvereine sollen von Fanclubs bzw. auch von Presseagenturen beobachtet werden.

Wir nutzen das Beobachter-Muster und bilden von den Klassen Observer und Observable, die ja nur das Muster definieren Unterklassen: Fanclub, Presseagentur bzw. Fussballverein

Klasse: Observer

- Unter-Klasse: Fanclub (Objekte: fan1, fan2, fan3)
- Unter-Klasse: Presseagentur: (Objekt: apa)

Klasse: Observable

- Unter-Klasse: Fussballverein: (Objekte: bayern, stuttgart)

Die beiden Vereine werden also von Fans und der APA (AustriaPresseAgentur) beobachtet.

1.7.1. Aufgabe: Unterklasse: Fussballverein erbt von Observable(u)

Projekt: ws-qt/seng/design-pattern/Beobachter-Muster-Fussball

Um die obige Implementierung zu testen schreiben Sie jeweils folgende Unterklasse:

Unterklasse: **Fussballverein** erbt von Observable:
☑ member:
string name;
string message;
☑ methoden:
Konstruktor

void setPresseMitteilung(string m);
// ruft die Methode notifyStateChange() auf

string getPresseMitteilung() const;
// wird von den Observer-Objekten (update()) aufgerufen

1.7.2. Aufgabe: Unterklassen: Fanclub, Presseagentur erben von Observer(u)

Projekt: ws-qt/seng/design-pattern/Beobachter-Muster-Fussball

Unterklasse: **Fanclub, Presseagentur** erbt von Observer:
☑ member:
string name;
☑ methoden:
Konstruktor
überschreibt die geerbte Methode update()

Testen Sie Ihre Implementierung mit folg. Programm:

```
#include <iostream>

#include "fanclub.h"
#include "presseagentur.h"

#include "fussballverein.h"

using namespace std;

int main() {
     cout << "Demo: ANWENDUNG: Beobachter-Muster V2!!!" << endl;

     Fanclub* fan1= new Fanclub("Fanclub Wiesn");
     Fanclub* fan2= new Fanclub("Fanclub Wald und Wiesn");
     Fanclub* fan3= new Fanclub("Fanclub Schwabenwald");
     Presseagentur* apa= new Presseagentur("Austria Presse Agentur");

     Fussballverein* bayern= new Fussballverein("FC Bayern");
```

```
    Fussballverein* stuttgart= new Fussballverein("VfB Stuttgart");

    bayern->addObserver(fan1);
    bayern->addObserver(fan2);
    bayern->addObserver(apa);

    stuttgart->addObserver(fan3);

    cout<< "!!! Hey, bei Bayern tut sich was ............"<<endl;
    bayern->setPresseMitteilung("FC Bayern ist Meister!!!");


    cout<< endl<<"!!! Hey, bei Stuttgart tut sich was ….."<<endl;
    stuttgart->setPresseMitteilung("VfB Stuttgart ist zweiter!!!");

    delete fan1;
    delete fan2;
    delete fan3;
    delete apa;
    delete bayern;
    delete stuttgart;

    return 0;
}
```

Kontrolle:
Folgender Output sollte erzeugt werden.

Demo: ANWENDUNG: Beobachter-Muster!!!

!!! Hey, bei Bayern tut sich was .....
FANCLUB: [Fanclub Wiesn] bin aktiviert worden.
M I T T E I L U N G: <FC Bayern ist Meister!!!>

FANCLUB: [Fanclub Wald und Wiesn] bin aktiviert worden.
M I T T E I L U N G: <FC Bayern ist Meister!!!>

PRESSE: [Austria Presse Agentur] bin aktiviert worden.
P R E S S E M I T T E I L U N G: <FC Bayern ist Meister!!!>

!!! Hey, bei Stuttgart tut sich was .....
FANCLUB: [Fanclub Schwabenwald] bin aktiviert worden.
M I T T E I L U N G: <VfB Stuttgart ist zweiter!!!>

Frage:
Die Klasse Observer soll als abstrakte Klasse implementiert werden. Was ist zu tun?

Antwort:
virtual void update()=**0**;

1.8. Java-Projekt: MUSTER-BEOBACHTER programmieren (java,u)

- Aufgabe: Mehrere Polizisten-Objekte sollen einen Dieb beobachten.
- Projekt: Muster-Beobachter

Die **Klasse Observerable** und das **Interface Observer** sind im Package: **java.util**.

Hier nun die Ausgabe: (ein Auszug)

```
POLIZIST: WachFrau Eva
 meldet folg. Beobachtung: Elster Karl..ich bin gerade hier..Schlossalle 12..tolle Uhren

POLIZIST: WachMann Heinz
 meldet folg. Beobachtung: Elster Karl..ich bin gerade hier..Schlossalle 12..tolle Uhren

POLIZIST: WachFrau Eva
 meldet folg. Beobachtung: Elster Karl..ich bin gerade hier..Einkaufsstrasse..schwere
Glunker

… usw. …
```

1.8.1. Aufgabe: Klasse DemoObserver, Dieb.java, Polizist.java (u)

- Projekt: Muster-Beobachter

Bringen Sie das folgende Programm zum Laufen:

DemoObserver.java

```
public class DemoObserver {
     public static void main(String[] args) {
          // OBSERVER
          Polizist[] polizisten = {
                    new Polizist("WachMann Heinz"),
                    new Polizist("WachFrau Eva")
          };

          //OBSERVABLE
          Dieb dieb = new Dieb("Elster Karl");

          for (int i=0; i < polizisten.length; i++) {
               dieb.addObserver(polizisten[i]);
          }

          dieb.aufGehts("Schlossalle 12 ... tolle Uhren");
          dieb.aufGehts("Einkaufsstrasse ... schwere Glunker");
          dieb.aufGehts("GefÃ¤ngnisstrasse 6 ... bin erwischt worden");
     }
}
```

Dieb.java

```
import java.util.Observable;
import java.util.Observer;

public class Dieb extends Observable {
	private String name;
	private String wo;

	public Dieb(String name) {
		super();
		this.name= name;
	}


	public void aufGehts(String wo) {
	   this.wo = wo;

	   // Markierung, dass sich was geändert hat
	   super.setChanged();

	   // ruft für alle Beobachter die update-Methode auf
	   super.notifyObservers();
	}


	public String toString() {
	   return name + "..ich bin gerade hier.." + wo + "\n";
	}
}
```

Polizist.java

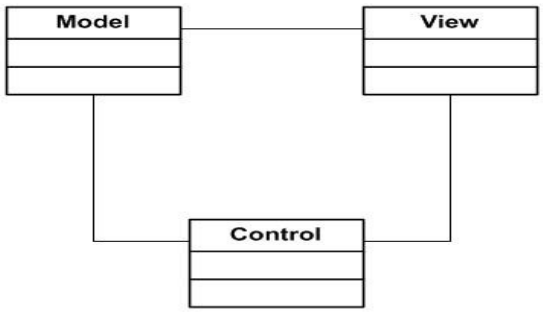
```
import java.util.Observable;
import java.util.Observer;

public class Polizist implements Observer {
	private String name;

	public Polizist(String name) {
	   this.name= name;
	}


	public void update(Observable o, Object arg) {
		Dieb dieb= (Dieb) o;

		System.out.println("POLIZIST: " + name);
		System.out.print(" meldet folg. Beobachtung: " );
		System.out.println(dieb.toString());
	}
}
```

1.9. Das MVC Konzept

Das Kürzel MVC steht für **Model-View-Control**.

Das MVC-Pattern ist eigentlich relativ einfach. Es besagt, dass

EIN PROGRAMM
in
DREI unterschiedliche TEILE aufgeteilt ist.



☑ Der **Model**-Teil
beinhaltet dabei die Daten bzw. den **Datenzugriff**.

Hier könnten sich klassische Objekte eines objektorientierten Programms befinden oder auch ein Datenbankzugriff oder jede andere Art von Datenverwaltung.

☑ Der **View**-Teil.
Hier befinden sich alle **Ansichten (GUI)** des Programms.

Diese zeigen normalerweise die Daten des Models an und sind **komplett unabhängig von den Daten** implementiert.

VORTEILE:
+ Views können einfach hinzugefügt/gelöscht werden.
+ Änderungen im Model haben kaum Auswirkungen auf die View.

☑ Der **Control**-Teil.
Dieser beinhaltet alle Kontrollmöglichkeiten, mit welchen ein **Benutzer das Programm steuern(Reaktion auf Mausklick, ...)** kann.
Auch dieser Teil ist normalerweise komplett **unabhängig** von den anderen Teilen.

☑ +Variante: **Document/View-Modell**
ist ein **M**odel/**V**iew**C**ontroller – Modell

In manchen Programmiersprachen bzw. Anwendungen ist man gezwungen den View-, sowie den Controlteil in einer Einheit unterzubringen. Dieses Pattern wird dann als **Document/View Modell** bezeichnet. (Java/Swing ist ein derartiges System)

Model,View,Controller sind meist nur mittels Referenzen verbunden und greifen über definierte/standardisierte Methoden aufeinander zu.

Wir wollen im nächsten Kapitel ein derartiges Beispiel entwickeln. Es wird ein Programm vorgestellt, das nicht nur ein **MVC Entwurfsmuster** implementiert, sondern gleichzeitig auch noch ein **Observer-Pattern** verwendet.

Diese Vermischung der beiden Entwurfsmuster kommt sehr häufig vor, da sie gut zusammen passen. Dabei sind die beiden so verknüpft, dass es sich bei den

☑ **View**-Klassen um die **Observer**klassen des Observer-Patterns handelt und die

☑ **Model**-Klasse die **Subject**-Klasse des Observer-Patterns repräsentiert.

1.10. Swing und das Model-View-Controller-Prinzip

Anstatt den gesamten Code in eine einzelne Klasse zu packen, werden beim MVC-Konzept drei unterschiedliche Bestandteile eines grafischen Elements nach dem Observer-Pattern unterschieden:

- View: grafische Darstellung der Komponente,
- Model: Daten der Komponente und Verwaltung der Daten,
- Controller: empfängt Tastatur- und Mausereignisse und veranlasst Änderung von Model und View.

1.10.1. So funktioniert es

Die Application enthält in ihrer View Bedienelemente zur Eingabe von Benutzer-Befehlen. Diese melden dem Model: Der Benutzer wünscht die Daten zu manipulieren.

Das Model manipuliert also die Daten und meldet der Application: Daten wurden manipuliert.

Die Application ihrerseits kann jetzt die frischen Daten im Model abrufen, um diese z.B. in der View anzuzeigen.

1.10.2. Warum?

Der Nutzen eines solchen Programm-Designs wird deutlich, wenn man es ausnutzt:

Man kann eine weitere View, die die Model-Daten anders präsentiert auf einfache Weise hinzufügen. Dazu die folgende Aufgabe.

1.11. Java-Projekt: MUSTER-MVC-LOTTO (java,u)

- Projekt: Muster-MVC-Lotto

Wir wollen ein Lotto-Programm nach dem MVC-Muster entwickeln. Swing soll als GUI-Framework Anwendung finden.

1.11.1. Model.java: extends Observable (u)

```
import java.util.Observable;

public class Model extends Observable {
      ...
}
```

☑ damit werden folg. Methoden von der Klasse Observable geerbt:

void addObserver(Observer o)
void deleteObserver(Observer o)
void setChanged()
void notifyObservers(Object arg)

☑ um auf die Daten zuzugreifen bietet die Klasse Model:

void doZiehung()
// führt eine Ziehung durch und speichert diese in den privaten MemberVar.
...
// zusätzlich wird das Beobachter-Muster verwendet,um interessierte Views zu
// benachrichtigen.
this.setChanged();
this.notifyObservers(this);

int[] getData()
// liefert die zuletzt gezogenen Lotto-Zahlen
// wird von den Views verwendet.

☑ Hier nun die Klasse: Model.java

```
// Das Model wird von Views 'beobachtet'
import java.util.Observable;
import java.util.Random;

public class Model extends Observable {
      private int[] zahlen;

      public Model(){
            this.zahlen= new int[6];
```

```
	}

	public void doZiehung(){
		Random zufall= new Random();

		for(int i=0; i <zahlen.length; i++){
			zahlen[i]= zufall.nextInt(45) +1; //1...45

			// gibts die Zahl schon?
			boolean bereitsGezogen=false;
			for (int j = 0; j < i; j++) {
				if (zahlen[i]== zahlen[j]){
					bereitsGezogen=true;
				}
			}
			if(bereitsGezogen==true){
				i--;
			}
		}//for

		//fertig, jetzt noch die Views informieren
		this.setChanged();
		this.notifyObservers(this);

	}

	public int[] getData(){ // wird v. Den Views aufgerufen
		return zahlen;
	}
}
```

1.11.2. EineZiehungView.java: implements Observer (u)

```
public class EineZiehungView extends JFrame implements Observer{
...
```

Die View implementiert das Interface Observer und muss demnach folgendes tun:

☑ die Methode:
public void update(Observable m, Object o){ …} implementieren.

Diese Methode wird automatisch aufgerufen, wenn sich beim Model was verändert.

☑ Die Methode:
public void setModel(Model m)

Diese Methode gibt der View einen Verweis auf das Model-Objekt.

Hier nun die Klasse: EineZiehungView.java

```
import java.awt.BorderLayout;

public class EineZiehungView extends JFrame implements Observer{
	// VIEW-Elemente
	private JPanel contentPane;
	private JButton btnZiehung;
	private JButton btnEnde;
	private JLabel label;

	// MODEL
	Model model;


	public EineZiehungView() {
		….
	}

	public void setModel(Model m){
		// Das Model merken
		model=m;

	}

	private JButton getBtnZiehung() {
		if (btnZiehung == null) {
			btnZiehung = new JButton("Ziehung");
			btnZiehung.addActionListener(new ActionListener() {
				public void actionPerformed(ActionEvent arg0) {
					model.doZiehung();
				}
			});
			btnZiehung.setBounds(12, 92, 93, 25);
		}
		return btnZiehung;
	}

	private JButton getBtnEnde() {
		…
	}

	private JLabel getLabel() {
		…
	}

	@Override
	public void update(Observable o, Object arg) {
		// Daten holen
		Model m= (Model)o;
		int[] zahlen= m.getData();
```

```
            //anzeigen
            String s= "";
            for (int i = 0; i < zahlen.length; i++) {
                  s+= String.valueOf(zahlen[i]) + ", ";
            }

            label.setText(s);
      }
}
```

1.11.3. Main.java: Das Hauptprogramm (u)

Main.java

```
public class Main {
      public static void main(String[] args) {

            // 1. Das Model erzeugen
            Model model= new Model();

            // 2. Die View
            EineZiehungView eineZiehungView = new EineZiehungView();
            eineZiehungView.setVisible(true);

            // 3. Der View das Model mitteilen
            eineZiehungView.setModel(model);

            // 4. Die View beim Model als Observer registrieren
            model.addObserver(eineZiehungView);
      }
}
```

Der einfache Austausch des Model-Objektes ist ebenfalls denkbar, d.h. Die Wiederverwendung von Views erhöht sich.

Diese Austauschbarkeit und Wiederverwendbarkeit ist ein Vorteil von MVC.

1.11.4. Aufgabe: Muster-MVC-Lotto (u)

1. Erstellen Sie das Projekt Muster-MVC-Lotto. Verwenden Sie als Vorlage die Datei Muster-MVC-Lotto-UE.zip.

2. Fügen Sie eine weitere View namens AlleZiehungenView hinzu. Verwenden Sie folgenden Programmteil (aus Main.java)

```
        // ------------------------------------------------
        // -- AUFGABE: Eine weitere View einfügen
        // ------------------------------------------------
        AlleZiehungenView alleZiehungenView= new AlleZiehungenView();
        alleZiehungenView.setVisible(true);

        // ???????????
```

3. Die View sollte alle Ziehungen aufzeigen und verwendet dafür eine Textarea.

```
public class AlleZiehungenView extends JFrame implements Observer{
     //VIEW
     private JPanel contentPane;
     private JButton btnZiehung;
     private JButton btnEnde;
     private JScrollPane scrollPane;
     private JTextArea textArea;
…
```