Entwurfsmuster, Bibliotheken und grafische Oberfläche

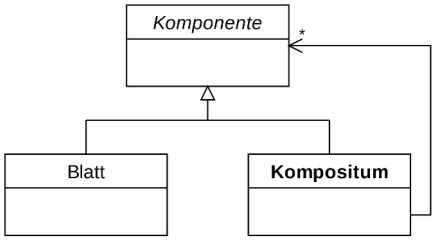
Entwurfsmuster

Entwurfsmuster sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in der Softwarearchitektur und stellen damit eine wiederverwendbare Vorlage zur Problemlösung dar. Neben den ***Entwurfsmuster*** gibt es auch ***Algorithmenmustern***, die Standardstrukturen für algorithmische Probleme darstellen.

Entwurfsmuster Kompositum

Das Kompositum ist bereits von den rekursiven Datenstrukturen bekannt.

Das Kompositum wird angewendet um eine Teil-Ganzes-Hierarchie zu repräsentieren. Man verbirgt hierbei die Unterschiede zwischen einzelnen Objekten und zusammengesetzten Objekten (Kompositionen).



Bibliotheken

Java stellt eine breite Palette von Funktionalitäten und Diensten in Form von Klassen bereit, die Entwickler nutzen können, ohne den Code von Grund auf neu schreiben zu müssen. Diese Klassen werden logisch sinnvoll in Bibliotheken zusammengefasst.

https://docs.oracle.com/en/java/javase/21/docs/api/

Du kennst bereits die Klassen **String** und **Thread** aus dem Paket ***java.lang***. Das Paket java.lang muss nicht importiert werden. Diese Klassen stehen immer zur Verfügung.

Alle anderen Klassen aus der Java-Bibliothek (***Java-API***) müssen erst ***importiert*** werden.

Hier kennst du schon die Klasse **Random** aus dem Paket ***java.util*** und ggf. die Klasse **AtomicInteger** aus dem Paket ***java.util.concurrent.atomic*** kennengelernt.

java.util.ArrayList<E>

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html

Die Klasse ArrayList<E> von Java ist eine Implementierung eines in der Größe veränderbares Array, welches die gängigen Listenmethoden implementiert.

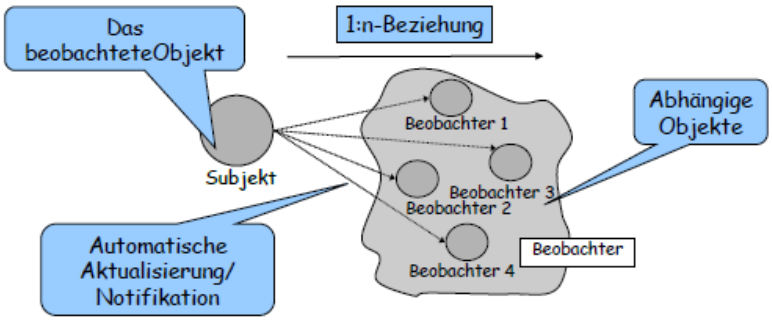
Entwurfsmuster Observer

Aufgabe 1:

Implementiere ein Entwurfsmuster Observer anhand der Präsentation zu Observer.

Dieses Prinzip beruht darauf, dass ein Objekt sich bei einem anderen Objekt (***Subjekt***) als „Beobachter“ (***Observer***) anmeldet und von nun an von diesem über gewisse Zustandsänderungen informiert wird. Es wird eine ***Eins-zu-viele-Abhängigkeit*** umgesetzt.

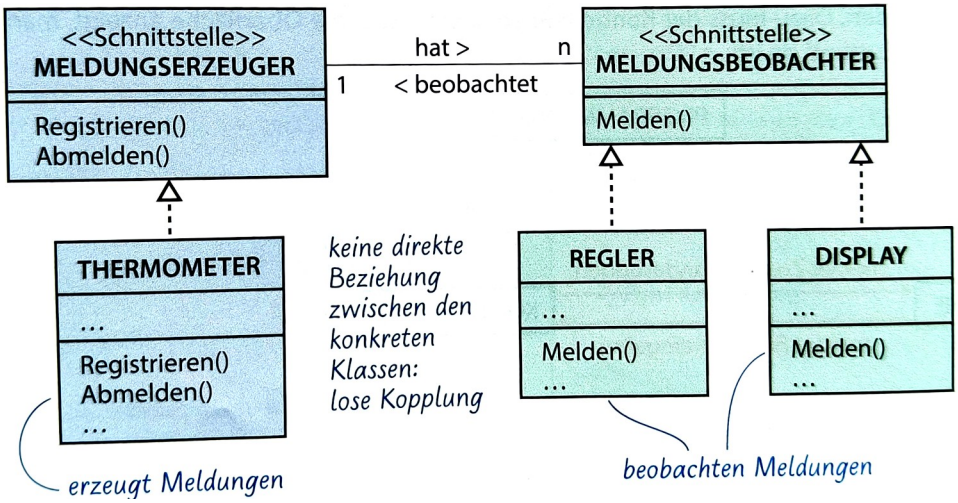
Beispiele: ***Blogs***, ***Newsletter***, ***Push-Nachrichten auf dem Smartphone***, …



Um das zu realisieren verwendet man in der Regel zwei Interfaces:

1. ***Interface Beobachtbar***
   Es bietet eine Möglichkeit, dass sich Beobachter anmelden()/abmelden() können. Die implementierende Klasse (=Subjekt) muss die Beobachter verwalten und kann über die Methode aktualisieren()/melden() die angemeldeten Beobachtbar über eine Änderung informieren.
2. ***Interface Informierbar***
   Enthält eine Methode aktualisieren()/melden(). Da ein Subjekt seine Beobachter verwaltet, kann das Subjekt über diese Methode die Änderungen an jeden Beobachter senden.

Beispiel aus dem Buch S. 176



GUI-Programmierung (grafische Oberfläche)

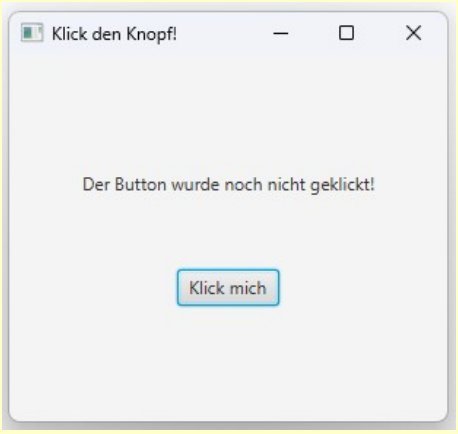
Zur Programmierung einer ***GUI*** (**G**raphical **U**ser **I**nterface) in Java gibt zwei Möglichkeiten:

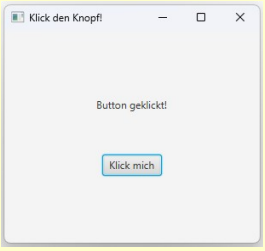
- Das sind die Bibliotheken ***AWT*** (Abstract Window Toolkit) und ***Swing***.
- Oder man nutzt die modernere ***JavaFX*** Bibliothek.

Vergleich: https://joachimhofmann.org/Klasse12/5%20Projekt/JavaFX%20vs.%20AWT%20+%20Swing.pdf

Code-Beispiel:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.event.*;
import javafx.geometry.*;

public class GUI extends Application {

   public void start(Stage stage) {

      //# (1) Komponenten erzeugen
      Label l = new Label("Der Button wurde noch nicht geklickt!");
      Button b = new Button("Klick mich");

      b.setOnAction(new EventHandler<ActionEvent>() {     //Observer welcher bei einem ActionEvent
            @Override                                     //(also einen Klick) benachrichtigt wird
            public void handle(ActionEvent event) {
               l.setText("Button geklickt!");             //Das passiert bei einem Klick
            }
      });
         ODER
      b.setOnAction(e ->  {                               //Kurzform über einen Lambda-Ausdruck
               l.setText("Button geklickt!");
      });

      //# (2) Layouts erzeugen und Komponenten einsetzen
      VBox v = new VBox(l,b);                  // Beide Elemente sind in einer vertikalen Box
      v.setAlignment(Pos.CENTER);              // Elemente mittig in der Box
      v.setPadding(new Insets(10));            // Abstand zum Rand: 10 px
      v.setSpacing(50);                        // Abstand zwischen den Elementen: 50px

      BorderPane pane = new BorderPane();      // Layout-Klasse
      pane.setCenter(v);                       // VBox in die Mitte des Layouts einsetzen

      //# (3) Szene erzeugen und ins Fenster setzen
      Scene scene = new Scene(pane, 300, 250);          // neue Scene mit dem Layout erzeugen

      //# (4) Fenster konfigurieren und anzeigen
      stage.setTitle("Klick den Knopf!");      // Fenstertitel setzen
      stage.setScene(scene);                   // Scene im Fenster anzeigen
      stage.show();                            // Fenster anzeigen
   }

   public static void main(String[] args) {
      launch();
   }
}
```





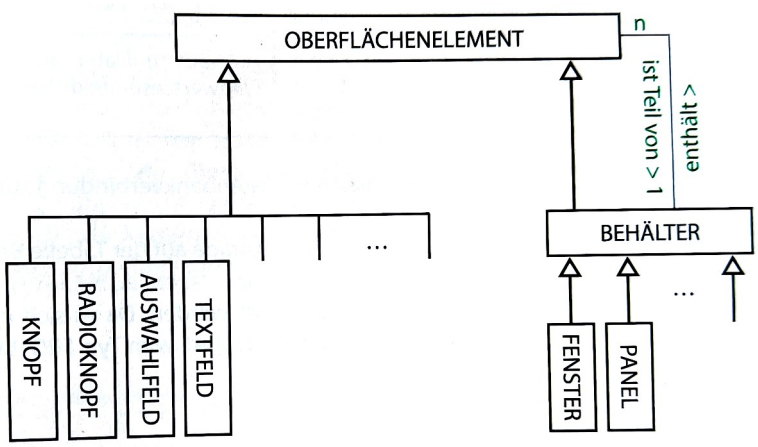
Weiterführende Erklärungen zu

Lambda-Ausdrücke:
https://javabeginners.de/Klassen_und_Interfaces/Lambda_Ausdruecke.php

Funktionsweise von JavaFX und verfügbare Komponenten/Kontrollelemente
https://www.dbs.ifi.lmu.de/Lehre/SEP/WS1718/t03_javafx.pdf

Layouts in JavaFX
https://michaelkipp.de/java/index.html

Entwurfsmuster Kompositum bei den Oberflächenelementen

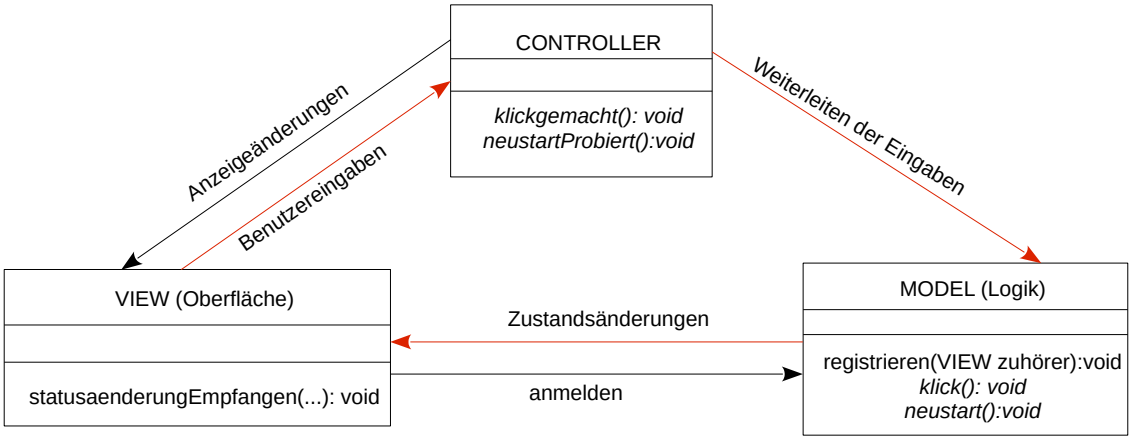


Enwurfsmuster Model-View-Controller (MVC)

Aufgabe 2:

Implementiere ein Entwurfsmuster MVC anhand der Präsentation zu MVC.

Das Entwurfsmuster MVC bietet mit einer Model – View – Controller Struktur ein Softwaredesign und dient als Grundlage vieler Softwareprojekte mit grafischer Aus- und Eingabe. Es verwendet dabei das Observer- und evlt. das Strategie-Muster, um die einzelnen, unabhängigen Komponenten voneinander zu trennen.



MODEL

Das MODEL entspricht der ***Logik*** der Software. Sie verwaltet die Software in ihrem Inneren und nimmt Berechnungen vor:

z.B.: **klick(), neustart()**,...

Treten Änderungen auf, so muss das Model diese ***Zustandsänderungen*** an seine Grafikschnittstelle weitergeben (siehe VIEW)

VIEW

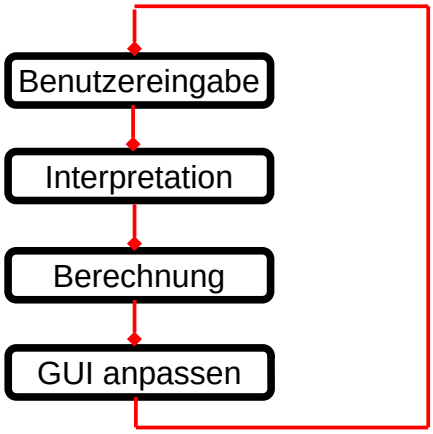
Die VIEW entspricht der ***graphischen Oberfläche (GUI)*** der Software. Sie nimmt Eingaben des Benutzers entgegen und leitet sie erst an den CONTROLLER weiter.

Außerdem muss sie sich beim MODELL (als Beobachter) ***anmelden***, um über Änderungen der inneren Struktur informiert zu werden. Darauf reagiert sie entsprechend mit graphischen Veränderungen.

CONTROLLER

Der CONTROLLER ist das ***Bindeglied*** zwischen VIEW und MODEL. Er interpretiert die Eingaben des Benutzers für eine VIEW und leitet sie weiter an das MODEL. Eventuelle Grafikeinstellungen, die keiner Berechnung bedürfen, nimmt er selber vor.

Es ergibt sich im Wesentlichen folgender Zyklus (im Diagramm rot dargestellt):



main-Methode

Bisher haben wir in BlueJ das Programm über das Erzeugen eines Objekts der „Hauptklasse“ gestartet. Dies ist bei anderen Programmen nicht der Fall. Hier kann man die Datei mit Doppelklick starten.

Die main-Methode ist die Methode, die beim Starten eines Programms (Doppelklick auf die Datei) ausgeführt wird. Sie ***muss*** folgende Signatur besitzen:

public static void main(String[ ] args)

- Sie muss ***public*** sein, ansonsten kann sie der Nutzer, der das Programm starten will, nicht ausführen.
- Die Methode muss ***statisch*** sein, da sie als allererstes aufgerufen wird, bevor ein Objekt erzeugt wurde.
- Sie besitzt ***keinen Rückgabewert***.
- Sie bekommt ein ***String-Array übergeben***, das die Kommandozeilenparameter enthalten kann (Das Array kann auch leer sein). Beispiel:

  shutdown.exe -s -f -t 60

  -s, -f, -t und 60 sind die Kommandozeilenparameter, die das Programm mit gewünschten Werten des Nutzer starten: -s steht für Shutdown also Herunterfahren, -f steht für force also das Herunterfahren erzwingen, -t steht für Timer also eine Angabe, nach wie viel Sekunden heruntergefahren werden soll und 60 steht für die Anzahl der Sekunden.

Einschub: ausführbares Programm mit BlueJ erstellen

- Schreibe eine main-Methode (am besten in der Hauptklasse), die das Programm passend startet.
- Projekt → Als jar-Archiv speichern… → Gib die Klasse mit der main-Methode an → Gib der Datei einen Namen
- Die .jar-Datei lässt sich nun mit Doppelklick ausführen, sofern eine passende Java-Version installiert ist.
  https://www.oracle.com/java/technologies/javase-downloads.html

Für eine JavaFX-Application ist es etwas komplizierter:

- Professionelle Entwicklungsumgebung (z. B. IntelliJ) mit gleichem JDK wie in BlueJ
- JavaFX Projekt anlegen (Default-Einstellungen)
- Java-Dateien aus BlueJ in den src-Ordner
- Extra Main-Klasse „App“ zum Starten anlegen

```
public class App {
    public static void main(String[] args) {
        GUI.main(args);
    }
}
```

- Project Structure → Artifact → + → JAR → neue Klasse als Main-Klasse nutzen
- Build → Build Artifacts

Aufgabe 3:

Exportiere deinen Klickzähler als .jar-Datei.

Daten persistent in Dateien speichern

Es gibt viele Möglichkeiten in Java wie man Daten in einer Datei speichern kann. Die folgende Variante ist kurz und ermöglicht den Zustand (=Attributwerte des Objekts) eines ganzen Objekts z. B. einer ArrayList zu speichern:

Objekte, die gespeichert werden sollen, müssen bei dieser Variante das Interface Serializable implementieren. Dabei muss aber keine Methode implementiert werden.

```
import java.io.Serializable;
public class ERGEBNIS implements Serializable{
   String name;
   int punkte;

   ...

}
```

```
import java.util.ArrayList;
import java.io.*;

public class HIGHSCORE{

   ArrayList<ERGEBNIS> a;

   public HIGHSCORE(){
      a = new ArrayList<ERGEBNIS>();
      //a.add(new ERGEBNIS("a",1));
      //...
   }

   public void save() {
      try{
         FileOutputStream fos = new FileOutputStream("t.txt");          // legt die Datei t.txt an und ermöglicht das Schreiben in die Datei
         ObjectOutputStream oos = new ObjectOutputStream(fos);          // kann Objekten in den FileOutputStream schreiben
         oos.writeObject(a);                                            // konkreter Schreibvorgang in die Datei
         oos.close();                                                   // Schließen der Streams
         fos.close();                                                   // = "Aufräumen"
      }catch(Exception e){}
   }

   public void load() {
      try{
         FileInputStream fis = new FileInputStream("t.txt");            // öffnet die Datei t.txt und ermöglicht das Lesen aus der Datei
         ObjectInputStream ois = new ObjectInputStream(fis);            // kann Objekte aus dem FileInputStream lesen
         a = (ArrayList<ERGEBNIS>) ois.readObject();                    // konkreter Lesevorgang aus der Datei
         ois.close();                                                   // Schließen der Streams
         fis.close();                                                   // = "Aufräumen"
      }catch(Exception e){}
   }

   ...

}
```

Aufgabe 4:

Erstelle eine Liste mit Ergebnissen beim Klickzähler. Ein Ergebnis ist der Name des Spieler (z. B. über ein TextField in der GUI) und die benötigte Zeit. Sobald 50 Klicks erreicht werden, nimm das Ergebnis in der Liste auf und speichere es sofort als Datei.

Beim Starten des Programm soll das Programm diese Datei einlesen.

Hinweise: Falls keine Datei gefunden werden kann, wird eine Exception geworfen. Diese wird aber durch das try-catch abgefangen und das Programm stürzt nicht ab. Überprüfe anschließend, ob das Objekt initialisiert wurde.

Man kann auch überprüfen, ob die Datei existiert:

```
File f = new File(filePathString);
if(f.exists() && !f.isDirectory()) {
    // do something
}
```

Aufgabe 5: (optional)

Wie bei einem Highscore soll die Liste immer nach der Zeit passend sortiert bleiben.

Tipp: Interface Comparable und Collections.sort() ggf. mit Lambda-Ausdruck, damit man nicht alles selbst implementieren muss.