Lösungen zur Übung Open Closed Principle

Lösung Aufgabe 1

Variante 1 – Just Change It Quickly (not so good example)

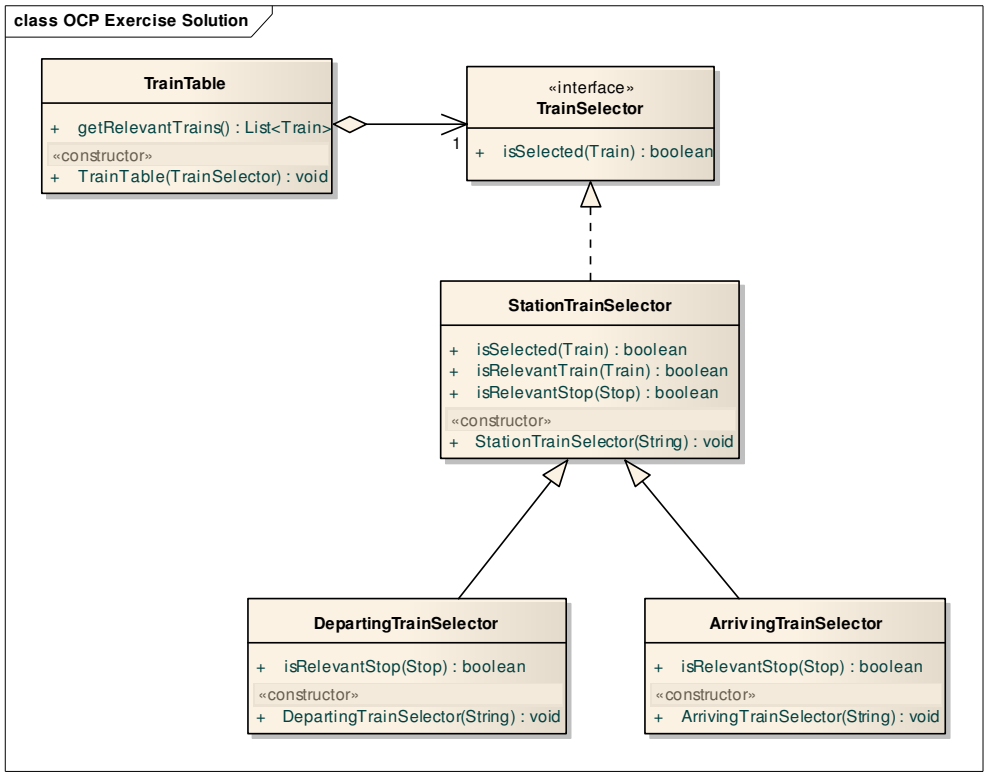
Folgende Lösung erfüllt das Open Closed Principle in keiner Weise. Alle weiteren Erweiterungen werden unweigerlich zu Code-Veränderungen führen und der Code wird mit jeder Erweiterung unübersichtlicher:

```
/**
 * Table of trains to display the relevant trains departing or arriving in a
 * station.
 *
 * Version 2 - with quick and dirty extension to support arrival tables.
 */
public class TrainTable {

      private List<Train> relevantTrains;

      /**
       * Initialize a train table from a list of trains
       * by filtering this list for a station.
       *
       * @param trains the trains to use for the table (not yet filtered)
       * @param stationCode the station code of the station for which to filter
       *            the trains
       * @param arrival should the trains arriving in that station be contained
       * @param departure should the trains departing in that station be con-
tained
       */
      public TrainTable(List<Train> trains, String stationCode,
                              boolean arrival, boolean departure) {

            relevantTrains = new ArrayList<Train>();

            for (Train train : trains) {

                  if (train.isPublic()) {

                        Stop stop = train.getStop(stationCode);

                        if (stop != null) {

                              if ((departure && stop.isStopToGetOn())
                                    || (arrival && stop.isStopToGetOff())) {
```

```
                                relevantTrains.add(train);

                            }

                        }

                    }

                }

        }

        /**
         * @return all the trains that depart from the station to display in
         *         departure table.
         */
        public List<Train> getRelevantTrains() {
                return relevantTrains;
        }

}
```

Variante 2 - Delegation to a Selector-Object

A TrainSelector-Interface is introduced. Several implementations of this interface can be plugged into the TrainTable-Component.



Die ursprüngliche Funktionalität *DepartureTable* kann nun einfach unter Verwendung einer *TrainTable* mit einem *DepartingTrainSelector* implementiert werden.

Lösung Aufgabe 2

Mit Design gemäss Variante 2 zur Aufgabe 1 kann nun unsere Komponente ganz einfach erweitert werden, um auch spezielle Bahnhofs-Übersichts-Pläne zu unterstützen:

```
public class StationOverviewTrainSelector extends StationTrainSelector {

      // ... constructor and java doc is missing in this code listing

      public boolean isRelevantStop(Stop stop) {
            return stop.isStopToGetOn() || stop.isStopToGetOff();
      }
}


// Example usage of train table for Station-Overviews:
TrainTable table = new TrainTable(trains,
                                  new StationOverviewTrainSelector("CHUR"));
List<Train> trains = table.getRelevantTrains();
```

Lösung Aufgabe 3

Mit Design gemäss Variante 2 muss lediglich die Basis-Implementation für TrainSelector angepasst werden, also die Methode *isRelevantTrain(Train)* in der Klasse *StationTrainSelector*. Da alle Train-Selektoren davon ableiten, profitieren sie automatisch von diesem „bug fix".

Lösung Zusatz-Aufgabe

Mögliche Ansätze:

- *TrainSelector* um eine compare-Methode erweitern welche Züge vergleichen kann.
- *TrainSelector* um eine Methode erweitern welche die relevante Zeit für einen Zug zurückgibt, nach welcher sortiert werden soll (z.B. `getRelevantTime(Train train)`)

Bei beiden Varianten ist der *TrainSelector* eigentlich ein wenig mehr als ein „Selektor" geworden. Es wäre in diesem Falle eventuell sinnvoll die Klassen umzubenennen, z.B. in *TrainTableStrategy*.

Es gibt sicherlich noch weitere mögliche Varianten.