Exercise `01_report`

Dr. Günter Kolousek

22. September 2018

Dieses Beispiel soll ein Programm implementieren, das einen Report (Bericht) aus einer Menge von Umsätzen erstellt.

1 Allgemeine Instruktionen

- Don't print this document. It's not worth the paper it is written…
- Creating the excercises environment consists of the steps:
  1. Choose a location on your filesystem where you would like to store the excercises. But you should find it later on… Change into this particular directory.
  2. There, **create** a directory named `<lastname_studentnumber>` E.g.: it could be named `mustermann_i99001`.
  3. Change into the newly created directory. I.e., `cd mustermann_i99001` would do the trick.
  4. Next, create a new mercurial repository **inside** the currently created directory using the command `hg init`! Now!!
  5. Lastly, create an appropriate `.hgignore` file inside the newly created mercurial repository using your favorite text editor. It should consists of *all* filename patterns that should not go into the mercurial store. I.e., it should consists at least of:

```
syntax: glob

bin
build
out
```

You should add relevant patterns later on (if necessary)! Generated files, backup files, or other meaningless files must not go into the store!

6. Setup some procedure to backup the excercises directory. This is **important**! Without this, setting up your environment is not yet completed. Creating an archive regularly and storing it in a safe location would be possible but mounting it onto a Dropbox is probably better but storing the whole repository on bitbucket is definitely the best option (of the presented choices).

- Each exercise has to be stored in a **subdirectory** of the freshly created directory and has to be named `<excercise>`. For this exercise it would have to be named `01_reports`. Do not create this now. This will be left for later.

- **Stick to the coding conventions**. You will find the relevant document on the `ifssh.htlwrn.ac.at`.

  – In particular, you are free to use the C# naming convention (i.e. *pascal case*) but you are not encouraged to do it this way. Otherwise, use the *snake case* or the *camel case* notation. It's your choice but you have to stick with it!!! That's one of the few moments in life where you reached a point of no return `;-)`.

- Additionally, you are free to choose the IDE (like Visual Studio) or text editor (e.g. Emacs `;-)`) you are in love with… But the examinations will be performed using the Linux environment, so be prepared!

  In case you prefer Visual Studio Code, I may provide you with the following link `https://code.visualstudio.com/docs/other/dotnet` which describes the using of .NET Core with Visual Studio Code. Another documentation could be found at `https://docs.microsoft.com/en-us/dotnet/core/tutorials/with-visual-studio-code`. And maybe there are tons of articles to be found using your favourite search engine.

- You have to be capable of using the terminal. This is a **strict** requirement!

- Please bear in mind that the purpose of these exercises is to empower you to master the concepts laid down in the syllabus of this course. So, it's your duty to exercise this in-depth. Then, you will be able to manage the examinations. The relevant subject matter is summarized at the end of the instructions.

- And now to something completely different… No, just kidding! Anyway, it is really important: You have to maintain your repository in a consistent and timely manner. I.e., you have to commit often and regularly in such a way that each implemented meaningful functionality deserves an own commit! It's **your** responsibility! Save your excuses… It's pointless.

2 Überblick und prinzipielle Funktionsweise

Wir gehen davon aus, dass wir eine Datei mit lauter Umsätzen von Verkäufern je Produkt und Monat als auch den erzielten Umsatz (Verkaufspreis und verkaufte Anzahl) in einer Datei vorliegend haben.

Solch eine Menge von Datensätzen (z.B. aus dem ersten Quartal eines Jahres) könnte folgendermaßen aussehen:

Produkt	Verkäufer	Preis	Anzahl	Monat
Audi A6	Maier	55000	1	1
Audi Q5	Maier	60000	1	2
VW Golf	Maier	18000	2	1
VW Golf	Maier	18000	2	2
VW Golf	Maier	18000	1	3
VW Golf	Huber	18000	3	2
VW Golf	Müller	18000	2	2
VW Golf	Müller	18000	2	1
BMW X6	Kaiser	120000	1	2
BMW 5GT	König	80000	2	3
BMW 5GT	König	80000	2	3
Dodge Challenger	Sandmann	35000	1	1
Dodge Challenger	Sandmann	35000	1	1
Honda Civic	Sandmann	28000	2	3
Honda Civic	Dorfer	28000	2	1
Honda Civic	Dorfer	28000	1	3

In weiterer Folge soll diese Tabelle so sortiert werden, dass das Hauptkriterium die Artikel und innerhalb von gleichen Artikeln nach dem Verkäufer sortiert wird. Damit sieht die Tabelle folgendermaßen aus:

Produkt	Verkäufer	Preis	Anzahl	Monat
Audi A6	Maier	55000	1	1
Audi Q5	Maier	60000	1	2
BMW 5GT	König	80000	2	3
BMW 5GT	König	80000	2	3
BMW X6	Kaiser	120000	1	2
Dodge Challenger	Sandmann	35000	1	1
Dodge Challenger	Sandmann	35000	1	1
Honda Civic	Dorfer	28000	2	1
Honda Civic	Dorfer	28000	1	3
Honda Civic	Sandmann	28000	2	3
VW Golf	Huber	18000	3	2
VW Golf	Maier	18000	2	1
VW Golf	Maier	18000	2	2
VW Golf	Maier	18000	1	3
VW Golf	Müller	18000	2	2
VW Golf	Müller	18000	2	1

Es soll jetzt ein Report auf der Konsole ausgegeben werden, der für jeden Verkäufer und jedes Produkt die Umsatzsumme und für jedes Produkt die Gesamtumsatzsumme als auch den Gesamtumsatz ausgibt. Das soll so aussehen:

```
### Umsatzstatistik nach Produkten und Verkäufern ###

Produkt              Verkäufer        Umsatzsumme

Audi A6              Maier             55000 *
Audi A6                                55000 **

Audi Q5              Maier             60000 *
Audi Q5                                60000 **

BMW 5GT              König            320000 *
BMW 5GT                               320000 **

BMW X6               Kaiser           120000 *
BMW X6                                120000 **

Dodge Challenger     Sandmann          70000 *
Dodge Challenger                       70000 **

Honda Civic          Dorfer            84000 *
Honda Civic          Sandmann          56000 *
```

```
Honda Civic                          140000 **

VW Golf              Huber            54000 *
VW Golf              Maier            90000 *
VW Golf              Müller           72000 *
VW Golf                              216000 **

Gesamtumsatz                         981000 ***
```

3 Begriffe

- Unter einer Batchverarbeitung (Stapelverarbeitung, engl. batch processing) versteht man in der Informatik eine sequentielle Verarbeitung einer festgelegten Menge von Daten (z.B. Erstellen aller Lohnabrechnungen für alle Mitarbeiter für ein bestimmtes Monat).

- Ein Datensatz (engl. data record) besteht aus einer Folge von Datenfeldern. Jedes Datenfeld ist durch einen Namen und einem Typ gekennzeichnet. In unserem Beispiel haben wir die Datenfelder *Artikel*, *Verkäufer*, *Preis* (*Verkaufspreis*, VK-Preis), Anzahl und *Monat*.

  Im relationalem Modell, das die Basis von relationalen Datenbanken bildet, entspricht jedes Datenfeld einem Attribut. Die Anordnung der Attribute entspricht dort einem Tupel.

- Ein Gruppenbegriff oder Ordnungsbegriff (engl. key) ist ein Datenfeld nach dem die Datensätze geordnet werden können.

- Alle Datensätze mit gleichem Ordnungsbegriff bilden eine Gruppe.

- Ein Gruppenwechsel (engl. control break) liegt vor, wenn sich der Gruppenbegriff durch Verarbeiten eines neuen Satzes ändert, und somit eine neue Gruppe verarbeitet werden soll.

  Existiert nur ein Gruppenbegriff spricht man von einem einfachen Gruppenwechsel, bei zwei Gruppenbegriffen von einem zweifachen Gruppenwechsel, bei n Gruppenbegriffen von einem n-fachen Gruppenwechsel.

4 Aufsetzen des Projektes mit .NET Core

Alle hier angeführten Anweisungen sind für ein Unix-artiges Betriebssystem angeführt. D.h. die Befehle und die Dateinamenkonventionen sind so angegeben, dass diese unter einem Unix-artigen Betriebssystem (z.B. Linux oder macOS) so verwendet werden können.

Allerdings ist es *nicht* zwingend notwendig Linux zu verwenden, da die angeführten Anweisungen auch unter Windows verwendet werden können, soferne diese entsprechend den jeweiligen Konventionen und Syntaxregeln angepasst werden.

Die einzige Voraussetzung ist, dass die Projekte **ohne** Änderungen auch unter Linux zu übersetzen und auszuführen sind!!!

Beachte jedoch, dass unter Linux zwischen Groß- und Kleinschreibung unterschieden wird.

So ist ein .NET Core Projekt zu handeln:

1. Ok, um es klarzustellen: .NET Core ist super (FWIW), aber muss ich alle meine Informationen freiwillig an Microsoft schicken, auch wenn diese "anonymisisert" sind? Ich verstehe, dass es für Microsoft wirklich interessante Daten sind, aber…

   Wenn du willst, dann setze die Umgebungsvariable `DOTNET_CLI_TELEMETRY_OPTOUT` auf 1 und `dotnet` sendet (angeblich) keine Daten mehr nach Hause. Wer das mit einem Netzwerkscanner wie z.B. `wireshark` nachprüfen kann bitte bei mir melden und vorzeigen. Bitte. Danke. Mitarbeit…

   Prinzipiell gehe ich davon aus, dass du jetzt in Linux **und** Windows Umgebungsvariablen setzen kannst!

2. Ein neues Verzeichnis für ein Beispiel anlegen, also für dieses erste Beispiel wäre dies `01_report`. Das Anlegen eine .NET Core Projektes für Konsolenanwendungen geht mit folgendem Befehl:

   ```
   dotnet new console
   ```

   Dann wird allerdings in dem aktuellen Verzeichnis ein .NET Core Projekt mit dem Namen des aktuellen Verzeichnisses angelegt. Das geht prinzipiell schon, allerdings lautet das ausführbare Programm dann später wie das Verzeichnis und das wäre in unserem Fall `01_report` und das ist nicht gewünscht. Das ausführbare Programm soll `report` heißen.

   Deshalb gehen wir folgendermaßen vor (in unserem Verzeichnis `<lastname_studentnumber>`):

```
dotnet new console --name report --output 01_report
```

Mit diesem Befehl wird ein *neues* Verzeichnis `01_report` angelegt, das ausführbare Programm wird jedoch `report` heißen.

3. In dieses Verzeichnis wechseln: `cd 01_report`

4. In diesem neuem Verzeichnis wurde auch ein "Hello-World" Programm in der Datei `Program.cs` mitangelegt.

   Mittels `dotnet new --help` wird eine Hilfe angezeigt, die alle möglichen und notwendigen Optionen anzeigt. Hier sieht man auch welche Arten von Anwendungen von .NET Core aktuell möglich sind.

5. Mittels `dotnet build` kann das Projekt übersetzt werden.

6. Und mittels `dotnet run` wird es ausgeführt.

   Damit sollte ein nettes "Hello World!" zu sehen sein. Das ist zwar nicht viel, aber immerhin besser als nichts.

7. Allerdings ist es von der Strukturierung der Sourcecode-Dateien besser, wenn diese in einem eigenem Verzeichnis liegen. Deshalb ist jetzt ein Verzeichnis `src` anzulegen und die Datei `Program.cs` in dieses Verzeichnis zu verschieben:

   ```
   mkdir src
   mv Program.cs src
   ```

   Danach wird das Programm mittels `dotnet run` sowohl *übersetzt* als auch ausgeführt werden.

8. Andererseits dauert es doch etwas lange, um die Applikation mittels `dotnet run` zu starten. Ein "richtiges" Programm, also ein Programm, das direkt ausführbar ist, ist unter gewissen Umständen die bessere Wahl.

   Will man ein fertiges *ausführbares* Programm samt allen notwendigen DLLs erzeugen, dann geht dies mit: `dotnet publish --runtime linux-x64 -o build`. Danach kann das ausführbare Programm mittels `build/report` gestartet werden.

   Auch hier hilft ein beherztes `dotnet publish --help,` um zu ein wenig Hilfe zu kommen. Wichtig in diesem Zusammenhang ist der sogenannte Runtime-Identifier. Diese sind im RID-Katalog unter `https://docs.microsoft.com/de-de/dotnet/core/rid-catalog` zu finden.

   Die wichtigsten für uns sind:

- `win10-x64`
- `linux-x64`
- `osx-x64`

9. Eine entsprechend aktuelle .NET Core Implementierung in der Version 2.1 vorausgesetzt, kann man die C# Sprachversion von der default-mäßigen auf die C# Sprachversion 7.3 einstellen, indem man folgendes Codestück in die `.csproj` Datei hinzufügt:

```
<PropertyGroup>
    <LangVersion>7.3</LangVersion>
</PropertyGroup>
```

   Das Element `LangVersion` kann direkt zum bestehenden Element `PropertyGroup` hinzugefügt werden.

   Zwar brauchen wir die Features von C# 7.3 nicht direkt, aber was man hat, das hat man, nicht wahr?

5 Nun zum Programmieren!

1. Die zu schreibenden Klassen sollen sich in einem Namespace lautend auf deine Matrikelnummer befinden. D.h. ändere die Datei `Program.cs` entsprechend ab.

2. Schreibe eine `usage` Methode der Klasse `Program` in einem Programm `report`, die einfach nur den folgenden Text genau in dieser Form ausgibt und danach das Programm mit dem Exitcode 1 ($\rightarrow$ `Environment.Exit(1)`) beendet:

```
usage: report [--help|-h|-s] [FILE]
Print a sales statistics report ordered by product and salesclerk.

  --help|-h ... Help!
  -s ... sort it before producing the report
  FILE ... file name or - (stdin). If FILE is missing read from stdin
```

   Erweitere das Hauptprogramm, sodass du die Funktion auch testen kannst.

3. So jetzt ist noch eine Funktion `parse_argv` in *gewohnter* Weise zu implementieren! Die Funktion erwartet sich die Optionen und Parameter der Kommandozeile (Programmname ist unter .NET nicht dabei) als Sequenz und soll ein `struct` (selber

deklarieren) zurückliefern, das den Namen der Datei und einem boolschen Wert (`true` … sortieren!) enthält.

Wird kein Dateiname angegeben oder nur der Bindestrich, dann soll `"-"` für den Namen der Datei zurückgeliefert werden.

Diese Kommandozeilenverarbeitung soll so programmiert werden, dass die Option *vor* dem Parameter kommen muss. Eine falsche Option, eine Option *nach* dem Parameter oder mehrere Optionen oder Parameter werden als "falsch" zurückgewiesen:

```
$ report -t sales_statistics.csv
usage: report [--help|-h|-s] [FILE]
Print a sales statistics report ordered by product and salesclerk.

  --help|-h ... Help and exit!
  -s ... sort it before producing the report
  FILE ... file name or - (stdin). If FILE is missing read from stdin

Unknown option: '-t'
$ report -s sales_statistics.csv xxx
usage: report [--help|-h|-s] [FILE]
Print a sales statistics report ordered by product and salesclerk.

  --help|-h ... Help!
  -s ... sort it before producing the report
  FILE ... file name or - (stdin). If FILE is missing read from stdin

No more options or parameters allowed!
Additional argument was: 'xxx'
$ report -s -y
usage: report [--help|-h|-s] [FILE]
Print a sales statistics report ordered by product and salesclerk.

  --help|-h ... Help!
  -s ... sort it before producing the report
  FILE ... file name or - (stdin). If FILE is missing read from stdin

No more options allowed!
Additional option was : '-y'
```

Tja, und jetzt noch ein Tipp zum Schluss: Zeichne dir ein Zustandsdiagramm von einem endlichen Automaten und dir wird die korrekte Implementierung viel leichter von der Hand gehen!

4. Schreibe eine Funktion `process`, die einen Parameter, nämlich das Konfigurationsobjekt (Rückgabe von `parse_argv`) bekommt.

   Vorerst soll diese Funktion lediglich die Datei öffnen und die Daten einfach auf `stdout` ausgeben.

   Der einfachste Ansatz ist, die Methode `ReadAllText` der Klasse `System.IO.File` zu verwenden, aber beachte, dass deine Programme nicht abstürzen dürfen! Ist die angegebene Datei nicht vorhanden, dann soll vorerst nur der Fehlertext der entsprechenden Exception ausgegeben werden (zur Übung also).

5. Gut, das mit dem Ausgeben der Nachricht einer Exception funktioniert jetzt, kann das Programm jetzt so umgebaut werden, dass ein eigener Fehlermeldungstext anstatt des Fehlertextes der Exception erscheint:

   ```
   $ build/report -s abc.txt
   File 'abc.txt' not found!
   ```

   Verwende `string.Format`!

   Weiters soll sich das Programm in solch einem Fall mit einem Exitcode von 2 beenden. D.h. 1 bedeutet, dass die Benutzerschnittstelle nicht korrekt bedient wurde und 2 bedeutet, dass die angegebene Datei nicht vorhanden.

6. Ok, nur zur eigentlichen Implementierung. Dafür benötigen wird die einzelnen Zeilen und auch hierfür stellt `File` eine entsprechende Methode zur Verfügung…

   Refactoring: Stelle dein Programm so um, dass jetzt die Methode `ReadAllLines` verwendet wird.

7. In weiterer Folge soll ja nicht nur der Text ausgegeben, sondern die eigentliche geforderte Funktionalität programmiert werden. Daher ist die Datei als CSV Datei zu interpretieren.

   Deklariere eine weitere Struktur für einen Datensatz mit dem Namen `Record`, die alle notwendigen Attribute enthält (siehe Daten aus obiger Tabelle).

   Iteriere wie vorher über alle Zeilen der Datei und lege jeweils einen eigenen Datensatz an, der zu einer Liste (vom Typ `System.Collections.Generic.List`) hinzugefügt wird. Strings in Zahlen kannst du konvertieren z.B. mittels `double.Parse("123.45")` (wirft u.U. eine `FormatException` oder ohne Exception:

   ```
   string s="-123";
   int i;
   if (Int32.TryParse(s, out i))  // -> true
   ```

```
    Console.WriteLine(j);  // -> -123
else
    Console.WriteLine($"{s} not an int32!");
```

Danach, d.h. nachdem die Liste vollständig gefüllt wurde, sollen die Datensätze wieder auf die gleiche Art und Weise ausgegeben werden. D.h. auch hier handelt es sich um ein Refactoring (`Record` wird auch eine `ToString` Methode benötigen)!!! D.h. Die Funktion des Programmes ändert sich also nicht.

8. So, gemäß der oben erläuterten prinzipiellen Funktionsweise müssen die Datensätze nach dem Hauptkriterium (also dem Produkt) und innerhalb eines Hauptkriterium nach dem Nebenkriterium (also dem Verkäufer) sortiert werden muss.

   Du hast dazu mehrere Möglichkeiten, die im Theorieunterricht durchgenommen worden sind… Sinnvollerweise wählst du hier eine Variante, die ”in-place” sortiert, denn jeder unnötig angeforderter Speicher is ein verlorener Speicher (hmm, na ja eigentlich geht es hauptsächlich um die Laufzeit), aber wie auch immer: Just do it!

   Die Ausgabe der Datensätze bleibt bestehen und dient gleichzeitig zur Kontrolle, ob die Sortierung korrekt funktioniert hat.

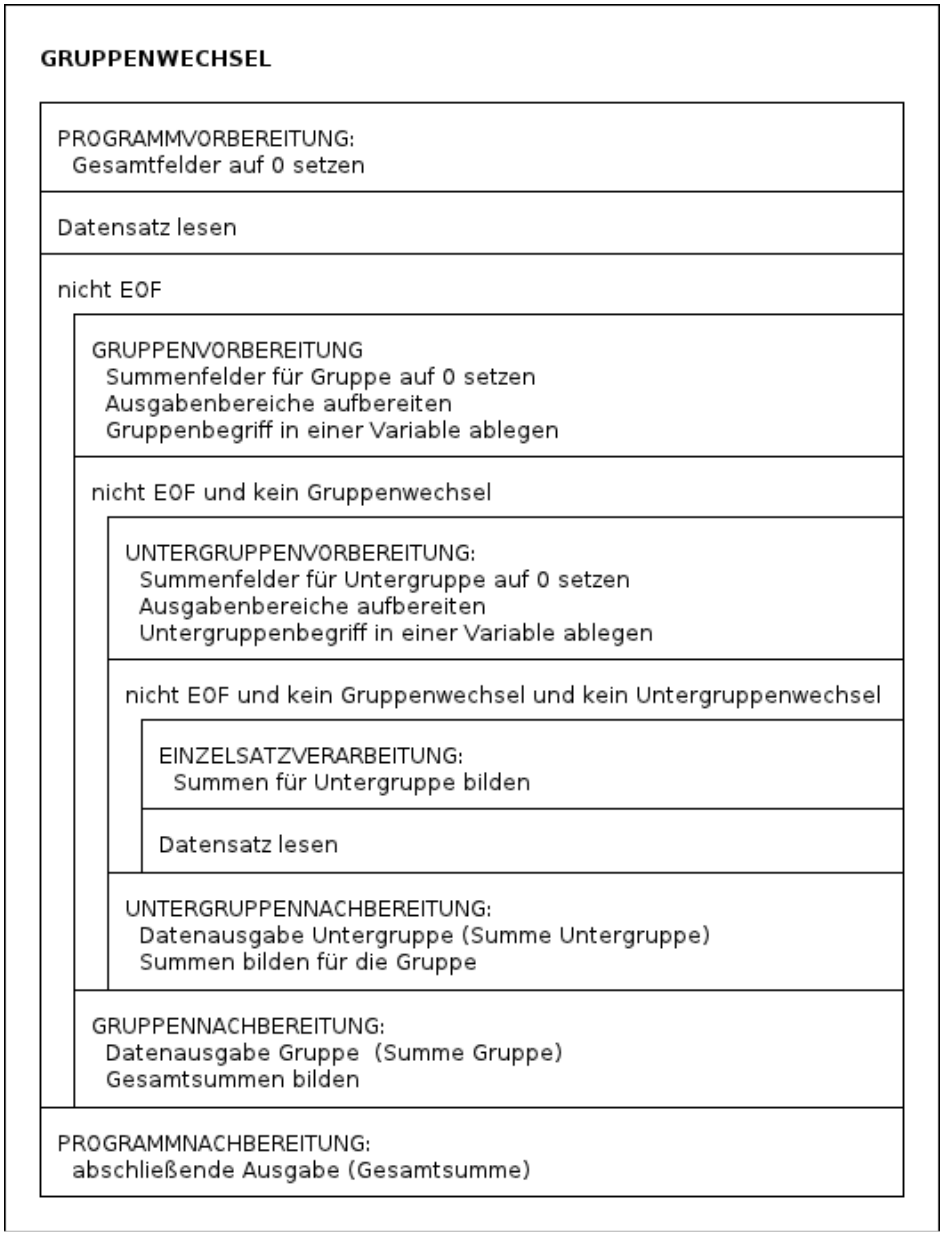
9. Rein zu Übungszwecken: Schreibe die sortierten Datensätze wieder in eine CSV Datei, wähle aber einen anderen Namen, damit die unsortierten Daten schön unsortiert bleiben `;)`

   Die Primitivmöglichkeit ist über alle Datensätze zu iterieren, um zu einer Liste von Strings zu kommen, aber… Alternativ gibt es auch noch `Select` aus `System.Linq` (analog zu `OrderBy`), muss aber nicht sein…

10. Von wegen einen Report erstellen. Davon haben wir bis jetzt noch gar nichts implementiert.

    Schreibe jetzt einen zweistufigen Gruppenwechsel!

    Der Algorithmus geht folgendermaßen:



Bei dieser Darstellung handelt es sich um ein sogenanntes Struktogramm (auch Nassi Schneiderman Diagramm genannt). Der Programmname (Gruppenwechsel) steht ganz oben. Danach folgen die einzelnen Schritte.

Der erste Schritt, der mit "Programmvorbereitung" überschrieben ist, sagt aus, dass die Gesamtfelder auf 0 zu setzen sind. D.h. es handelt sich um eine verbale Beschreibung der Aktion, die im Programm durchgeführt werden soll. Danach folgt eine `while` Schleife, deren Bedingung im ("nicht EOF") anzeigt, dass das Dateiende

(End Of File) noch nicht erreicht ist. Die umschließende Klammer gibt an, wie weit der Inhalt der `while` Schleife reicht.

Dieses Struktogramm gibt den allgemeinen Ablauf eines zweistufigen Gruppenwechsels an.

Wie kann man dieses Struktogramm aber jetzt umsetzen? Weiter zum nächsten Punkt!

11. Los geht's!

    Sieht man sich das Struktogramm genauer an, dann erkennt man, dass mehrere verschachtelte Schleifen enthalten sind und weiters erkennt man, dass dieses Struktogramm offensichtlich auf die Verarbeitung von Dateien ausgerichtet ist ("nicht EOF" bzw. "Datensatz lesen").

    Wir arbeiten allerdings derzeit mit einer Liste. An sich kein Problem, wir verwenden die Liste jetzt einmal und können in weiterer Folge das Programm so erweitern, dass es auch mit Dateien zurecht kommt.

    Um jedoch nicht zu viel an Refactoring später erledigen zu müssen, legen wir vorerst einen kleinen Refactoring-Schritt ein: Führe eine neue Funktion `void control_break(List<Record>)` ein, die die übergebenen Datensätze ausgibt und rufe diese anstatt der bereits bestehenden Ausgabe auf. Es sollte sich wiederum nichts an der Funktion unseres Programmes geändert haben.

12. Jetzt wird es ernst: Anstatt der Ausgabe in `control_break` soll der Report generiert werden und dazu implementieren wir den Gruppenwechselalgorithmus!

    a) Im Schritt PROGRAMMVORBEREITUNG steht, dass man die Gesamtfelder auf 0 setzen soll. Welche Gesamtfelder? Bei uns ist das einfach der Gesamtumsatz. Nennen wir diesen `total_sales`.

    b) Jetzt kommt schon die erste große Schleife. In dieser steht im Schleifenkopf "not EOF". Wir sind dann nicht am Ende, wenn es noch einen gültigen Datensatz gibt. Löse dies so, dass du Enumeratoren einsetzt, also:

       - einen Enumerator erzeugen: `IEnumerator<Record> rec_iter=records.GetEnumerator();`
       - zum nächsten Objekt: `rec_iter.MoveNext()` → `true` wenn vorhanden
       - auf das aktuelle Objekt zugreifen: `rec_iter.Current`

    c) Der nächste Schritt ist mit "GRUPPENVORBEREITUNG" betitelt.

- Hier sollen zuerst die Summenfelder der Gruppe auf 0 gesetzt werden. Die Gruppe nach der wir den Gruppenwechsel durchführen ist für uns die Umsätze je Produkt. Wir nennen daher die Variable `product_sales` und setzen diese brav auf 0.

- Weiters sollen wir die Ausgabebereiche aufbereiten. In unserem speziellen Fall reicht einfach die Ausgabe einer Leerzeile.

- Und den Gruppenbegriff in eine Variable legen. Der Gruppenbegriff ist für uns das Produkt, das wir gerade im Datensatz haben. Nennen wir die Variable `curr_product` (für aktuelles Produkt) und speichern wir uns hier das Produkt von unseren eingelesenen `Sale` Objekt ab.

d) Im nächsten Schritt kommen wir wieder zu einer Schleife. Für uns bedeutet das, dass wir wieder eine `while` Schleife benötigen. Wie `nicht EOF` zu behandeln ist, ist eh klar. Bzgl. Gruppenwechsel: ein Gruppenwechsel ist eine Änderung im Gruppenbegriff. Wie erkennen wir diesen? Indem wir in unserem Fall nachsehen, ob das Produkt des aktuellen Datensatzes verschieden zu dem in der Variable `curr_product` ist.

e) Wir kommen beim nächsten Schritt zur "UNTERGRUPPENVORBEREITUNG". Dieser Schritt ist analog zum Schritt "GRUPPENVORBEREITUNG".

- Das Summenfeld der Untergruppe nennen wir `salesclerk_sales`.

- Die Aufbereitung der Ausgabe ist noch einfacher: es ist in unserem Fall gar nichts zu tun.

- Die Variable für den aktuellen Verkäufer nennen wir `curr_salesclerk`.

f) Auf zur nächsten Schleife: Ein Untergruppenwechsel liegt bei uns vor, wenn sich der Verkäufer geändert hat. Ansonsten wie gehabt.

g) "EINZELSATZVERARBEITUNG": Einfach den Preis mit der Anzahl multipliziert zu `salesclerk_sales` addieren.

h) Datensatz lesen ist wieder besonders einfach, denn da müssen wir nur unsere Funktion `next_sale` aufrufen und das Ergebnis in der Variable `sale` ablegen.

13. Eine Sache ist noch offen: Wir haben eine Option `-s`, die angeben soll, dass die Daten zu sortieren sind. In diesem Fall sind die Daten zu sortieren, anderenfalls kann man diesen Schritt auslassen. Bitte implementieren.

14. Endlich fertig! Nur noch die Ausgabe unter Umständen hübscher machen.

6 Übungszweck dieses Beispiels:

- Einrichten eines Mercurial-Repositories wiederholen und üben
- Aufsetzen eines .NET Core Projektes
- C# lernen!
  - Funktionen und Funktionen mit optionalen Parametern
  - Ausgabe und Ausgabe formatieren
  - Umgang mit Kommandozeilenparameter
  - Exit-Code setzen
  - Umgang mit Dateien üben
  - Konvertieren von Strings in Zahlen
  - Collections
  - Strukturen kennenlernen
  - Exceptions abfangen
  - Sortieren nach mehreren Kriterien
  - Iteratorkonzept kennenlernen
- Wiederholen der Implementierung eines endlichen Automaten
- Gruppenwechsel kennenlernen
  - 2 stufigen Gruppenwechsel programmieren