REMOVING REDUNDANT WORK RELATED TO DATAOBJECT INSTANTIATION - DOIF

Mathias Walter Nilsen, Maria Elinor Pedersen, Robert Alexander Dankertsen

MAY 19, 2020
Høgskolen i Østfold

1. Intro

The project's *primary objective* is to alleviate redundant work related to reading and instantiating data from data sources into dataobjects. Where data comes from a single source like a file or a URL. The code for handling this is often copied throughout a project with minor changes and alterations to it. The identified changes are the following: Different URL or file name as a source, different class for holding data or different multiple sources altogether.

It further aims to handle data given in different formats like CSV and JSON. Whilst not limiting the number of possible dialects. Thereby allowing clients to utilize highly customized formats. Something that further allows for expanding the framework in the form of plugins. Which in turn will help mitigate duplicated code, often found when handling mentioned formats.

Another correlated goal under this is to allow the framework to automatically instantiate objects based on the data and available client classes. Something that would alleviate code known as boilerplate code. Which is simple duplicated code with smaller changes that often occur when reading data used for instantiating objects. Or whenever handling data in general.

In addition to this one often requires extracting concrete columns from a data set, which in turn is class fields for that object. Which may also be retrieved by respective methods. This code is mostly strong typed. Thereby requiring one to create code that manually collects that data. Something that could be automated with the usage of reflection. As reflection allows one to retrieve all information related to a class that might be used to get specific data from an object.

The secondary goal aims to allow for simple statistical calculations on datasets, or objects representing the data. Calculations available are different averages, standard deviation, standard error, covariance and correlation. Something that is often required or wanted when dealing with datasets.

Code like this is often strong typed and require concrete calls to statistical methods that take the actual data, along with additional required parameters. Something that could possibly be abstracted away by allowing the client to only specify concrete statistical calculations. Which would then be executed on an object containing said data.

While there are other frameworks and programs that offer statistical calculations, they do not allow for creating customized reports. Something that this framework could alleviate for the user. The reports consist of the different calculations that the framework has available. Here the user would either choose one of the predefined reports or make a unique one. Which could be done by setting required options.

When dealing with datasets it often requires one to execute multiple statistical calculations. This framework aims to do this smoothly and show the result in an intelligible way. Since statistics and probability usually are used intertwined, *the third goal* aims to provide calculations from probability models.

These objectives assume the user of the framework will specify necessary rules and settings in order to allow the framework to optimally handle any given content from a data source. Necessary rules will involve utilizing annotations when required and/or initializing a class with specific constructor parameters or a list of parameters. Which further is a compromise that aims to displease everyone equally whilst allowing the framework to handle a broader aspect of responsibilities. Like allowing for more file formats and/or other types of resources.

2. Group

The group consists of three second year students: Mathias Walter Nilsen, Maria Elinor Pedersen and Robert Alexander Dankertsen. Where everyone studies bachelor's in Computer Science at Høgskolen in Østfold, tough with varying specializations. Mathias and Maria studies a specialization in machine learning whilst Robert studies a specialization within data security.

3. Existing solutions

None was found though some libraries and smaller code snippets for handling certain parts of the framework existed. Like calculating different statistics, reading and parsing both JSON and CSV formats.

CSV

The Apache Commons CSV library (Apache Software Foundation, "Apache Commons CSV User Guide", 2020) is meant for parsing a CSV formatted string. The library has many features that are highly customizable and specialized for handling many lingos of CSV formats. However, the utilized API is both complex and difficult to learn as there is a lot of information to go through. This is more complex than it needs to be. Shown under is the easiest way to read a CSV file with default format. Other dialects exist, but they are very complex and therefore not shown.

CSV file:

```
author,title
Dan Simmons,Hyperion
Douglas Adams,The Hitchhiker's Guide to the Galaxy
```

CODE:

```
Reader in = new FileReader("book.csv");
Iterable<CSVRecord> records = CSVFormat.DEFAULT
    .withHeader(HEADERS)
    .withFirstRecordAsHeader()
    .parse(in);
for (CSVRecord record : records) {
    String author = record.get("author");
    String title = record.get("title");
}
```

JSON

There are primarily two main libraries for parsing JSON formatted strings. These are: GSON (Google, "GSON", 2020) and Jacksons JSON (Jacksons, "FasterXML/jackson", 2020). GSON is both highly efficient and sturdy while easy to use. Though it does not support annotations like Jacksons JSON. Shown here is how to parse using GSON, it takes a JSON string and a class definition to map it to:

```
new Gson().fromJson(jsonInput, T.class);
```

However, when starting to use the GSON library in a more complex way it quickly leads to complicated and duplicated code. Shown under is a code snippet that retrieves a list of objects, given a type Class and a JSON string. Something that allows one to utilize multiple classes with multiple JSON strings in order to prevent duplication of code.

```
GsonBuilder gsonBuilder = new GsonBuilder().setPrettyPrinting();
Gson gson = gsonBuilder.create();
Class<T[]> arrClass = (Class<T[]>) Array.newInstance(type,
0).getClass();
T[] arrangementArray = gson.fromJson(jsonTextFromFile, arrClass);
List<T> aList = (Arrays.asList(arrangementArray));
```

Jacksons JSON allows for more complex usages than GSON. A downside to Jacksons JSON is that it is not as easy to use. Jacksons allow for customizing the deserialization process. Though this customization requires the usage of annotations that are difficult to use. Combined with a hard to follow documentation on Baeldungs's website (Jackson, "Jackson Annotation", 2020) along with non-descriptive namings, this library makes it not ideal for the average developer. In this framework the JSON-handling are done by combining the efficiency of GSON and the idea of annotations in Jacksons.

Statistics Library

Apache Commons Math library (Apache Software Foundation, "Apache commons math3", 2020) contains functionality for performing various statistical calculations, as well as other mathematical functionality. Said library is also very simple to use whilst allowing for more complex calculations. Were all calculations are done via client called methods, which require them to send data into the method. This framework aims to allow clients to "select" which calculations they want to execute on collected data.

4. Framework Development Process

The framework was developed using test-driven development and scenario-driven design. Whilst the process itself consisted of regular sprints. Where each sprint lasted for a week and consisted of discussing the current progress and problems. Which in turn helped to track the overall progress and plan.

The first task was to research existing solutions. This was done in order to probe for similar frameworks that might solve the problem. After this a set of scenarios was developed to solve the problems as introduced in the first task. The next step involved creating pseudocode that corresponds to each scenario. Which in turn made up the initial design of the framework.

The next task involved creating an API Design which combined with user testing allowed for contiguous development. Where the API design would be affected by the testers demands. Thereby accommodating the users wishes and needs. Which would further allow more users to utilize the framework.

4.1. Scenarios

The following scenarios were utilized in creating this framework.

1. Instantiate JSON formatted data from a source into objects.
2. Instantiate CSV formatted data from a source into objects.
3. Specify which class fields and constructor to utilize for instantiating the object using the data.
4. Allow the client to extract all columns from instantiated objects.
5. Allow the client to determine specific columns to extract from instantiated objects.
6. Calculate simple average values (Mean, Median, Mid-range, Mode) on a dataset.
7. Calculate simple statistical values (Variance, Standard Deviation, Standard Error) on a data set. Does support both sample and population calculations.
8. Calculate simple statistical values on two data sets. Supported Calculations: Correlation between two datasets and Covariance between two datasets.
9. Allow client to create either a predefined or a custom statistical report on datasets.

10. Calculate values using different probability models. Supported models are Binominal Distribution and Poisson Distribution.
11. Allow a pool to handle instantiating objects from many sources.
12. Allow a pool to extract specified columns from n types of instantiated objects.

Scenarios 1, 2, 3, 4 and 5 is directly interpreted from the primary goal introduced in the intro. Whilst 6,7 and 9 is taken from the secondary goal. Though it only offers simplified calculations. Scenario 10 is from the third goal.

The following scenarios: 8, 11 and 12, was added later in the process. Scenario 8 with calculating simple statistics on two datasets was not applied in the initial design but was found to be useful later in the process. The same goes for scenario 11 and 12 that involves implementing pools for handling datasets.

4.2. API Design Specification

The following pseudocode is for the initial API design. The framework has changed but is still similar to the initial design.

Scenario 1 and 2

Collect data from a file that utilizes some sort of format type, like CSV or JSON:

```
ICollector collector = New "Type"Collector(File);
```

Another alternative that utilizes a factory pattern to guess the file extension:

```
ICollector collector = Collector.fromFileExtension();
```

Support for other extensions requires the client to implement the ICollector into their own collector class. Which therein does specify a set of concrete methods. Along with this, specific settings could be applied to the collector. Setting a delimiter is an example of this:

```
((CSVCollector)collector).setDelimiter(",");
```

Scenario 3

Specify which class fields and constructor to utilize for instantiating the object using the data. Start by fist annotating classes with @DTO.

```
@DTO
Class Myclass(){
      public type field1;
      public type field2;
      public MyClass(type field1, type field2){
            this.field1 = field1;
            this.field2 = field2;}}
```

Or optionally describe DTO Fields from a class to be instantiated by annotating the fields with @DTOField:

```
Class Myclass(){
      @DTOField
      public type field1;
      @DTOField
      public type field2;

      public MyClass(type field1, type field2){
            this.field1 = field1;
            this.field2 = field2;}}
```

Scenario 4

Allow clients to extract all columns from instantiated objects inside a collector:

```
collector.getAllColumns();
```

Scenario 5

Extract specific columns from the collected information. These need to be specified by the user:

```
Extractor.extractColumnFrom(Collector, String ...columnName);
```

Scenario 6

Calculate simple average values (Mean, Median, Mid-range, Mode) on a dataset:

```
Average average = new Average(data);
average.calcSum();
average.calcMean();
average.calcMedian();
average.calcMode();
average.calcMidRange();
```

Scenario 7

Calculate simple statistical values (Variance, Standard Deviation, Standard Error) on a data set. Does support both sample and population calculations.

```
SimpleStatistics simpleStatistics = new SimpleStatistics(data);
simpleStatistics.sampleVariance();
simpleStatistics.populationVariance();
simpleStatistics.standardDeviationFromSample();
simpleStatistics.standardDeviationFromPopulation();
```

Scenario 9

Allow client to extract a report from a Collector:

```
Extractor.extractReportFom(Collector, String ...columnName);
```

Scenario 10

Calculate values using different probability models.

```
Probability p = new Probability(int, probability);
p.calcPoissonDistribution(variable);
p.calcBinominalDistribution(variable);
```

4.3.Implementation Description:

The initial implementation utilized a single pattern, which was the factory method pattern. The pattern is used for creating a format specific collector based on the file extension. Thereby returning a new CSVCollector class if the file extension was CSV. This would allow the framework to support many more formats and be expanded upon more easily. As each new format would be added onto the existing ones.

All interfaces follow the principle: Interface Segregation Principle, which states the following: "Clients should not be forced to depend upon interfaces that they don't use.". Thereby avoiding polluted interfaces. As seen in ICollector where this interface only offers necessary methods for minimal functionality of a collector. Thereby making each concrete implementation handle its own specialty and settings. Something that also decouples the different responsibility areas. Which in turn follows another principle: Single Responsibility Principle.

A principle that states: "A class should have only one reason to change". Something that decouples complex bindings between different responsibilities. As each class only has one responsibility and thereby only one reason to change. Making the class both more robust and allows for a finer grain of classes. That in turn allows for both simpler refactoring and less probability of the code breaking.

Other principles used are the following: Liskov substitution principle, Dependency inversion principle, and Open-closed principle. An example of the latter is found in CSVCollector as this class allowed the client to extend its functionality but not modify existing functionality.

Dependency inversion principle states: One should "depend upon abstractions, not concretions.". Something that is seen in the different collectors that implements the interface: ICollector. Thereby abstracting away any implementation details. Which in turn prevents leaking those outwards.

Liskov substitution principle states: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.". Something that all collectors follows. As each collector takes expected parameters and has expected return value as enforced by the interface ICollector. Following all SOLID principles allows the framework to easily be expanded, maintained, refactored and encapsulates concrete implementation details.

4.4. Feedback from testers:

Mattia Lerario - Studying bachelor's in Computer Science at Høgskolen in Østfold – CEO at LERATECH solutions:

Mattia expressed that he wanted an easier way to extract columns related to scenario 3 and 4. He further pointed out that the documentation was both confusing and hard to follow. The latter has been corrected by making the documentation more precise and refined.

Extracting of columns was not changed due to having limited time left for further development. The critique has been noted and would have been acted upon had there been enough time. For further development one might give a concrete name to a class field and use that to describe a column instead.

Magnus Møllevik - Studying bachelor's in Computer Science – At Høgskolen in Østfold

Magnus was overall pleased with the solutions as they both where intuitive, and each method had functionality that correlated to its name. Which in turn followed regular naming conventions and the overall Java conventions. Though there were some places where the framework could improve. One example of this was to remove redundant casting of the object list gotten from a dataCollector, something that is related to scenario 1 and 2. The resulting method does this casting based on the class that is passed in, shown here:

```
List<TrumpWord> list = dataCollector.getAllObjects(TrumpWord.class);
```

Lars Emil Knudsen – Assistant Professor at Høgskolen in Østfold:

He was asked about the overall initial API-design. His response to this was the main reason for doing a rework of the whole framework. Including implementing the ability to automatically instantiate objects based on the dataset, instead of making the user to do this manually.

The resulting feedback led to this:

```
Ihandle handler = new CSVHandler();
ICollector collector = Collector.newCollector(source, handler);
Collector.collectData();
List<Object> objects = collector.getAllDataObjects();
```

The design allowed one to vary where that data comes from as well as handling said data in a specific way. Thereby allowing the users to either utilize predefined handlers or even implement their own.

Lars Vidar Magnusson – Associate Professor – At Høgskolen in Østfold:

Lars Vidar mentioned that the following classes Resource, Collector and Exctractor, were named in such a way that it was difficult to understand their distinct roles. The outcome of this was to simply rename the following classes: Resource → DataSource, Extractor → DataExtractor and Collector → DataCollector. Which in turn is consistent with the principle of keeping naming conventions throughout the framework consistent and intuitive.

Ole-Edvard Ørebæk – Student Assistant – Studying Master's in Applied Computer Science – At Høgskolen in Østfold:

At some point it was required for the framework to expand in order to allow clients to specify n datasets. He was then offered two proposed solutions and their pros and cons respectively. The proposed solutions were these: Proposed method 1:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
Public @interface CSV{
        String[] sources();
        Class<? extends Resource> sourceClass() default Resource.class;
        Class<? extends CSVHandler> handler() default CSVHandler.class;
```

Implemented as:

```
@DataObject
@CSV(sources  = {“https://source”, “file.csv”})
@JSON(sources = “file.json”)
Public class ComplexPoolTest{…}
```

Utilize annotations to describe how to handle the data and from where those Resources come from.

Proposed method 2:

```
DataSource.newResource().fromFile("file1").fromFile("file2")
.fromFile("file3");
```

Utilize a builder to configure and allow that Resource to gather data from multiple locations. This is also relatively easy to use and allows for some extendibility.

The result from this feedback was to implement the second proposed method. This turned out to be in general the most efficient and most consistent way of doing this. Which ended up pleasing more users of the framework.

5. Data Object Instantiation Framework with Statistics (DOIF)

This showcases the current framework solution for the problem. A solution that has been formed both by user input and common design principles. It was further developed with concrete scenarios in mind, which is referenced by number.

Abstract concepts

These were formed during the development cycle and are therefore important for further understanding of the framework. As these are often referenced throughout the documentation.

Dataset

Where collected data is pertained in a list describing all rows, where each row consists of an array that represents the individual columns of that row.

Dataobject

Any class that holds only data and has no functionality itself. Such classes must be annotated with @DataObject. Assumes all fields are of a primitive type and/or its wrapper counterpart. In addition to this all fields are publicly accessible or has dedicated getters.

Datacollector

A data collector is used for collecting data from a resource were data is handled with a handler. The data is used to instantiate objects which then stored in an RBTree inside the DataCollector.

Dataextractor

Extracts a single or multiple data columns using collected data from a DataCollector and returns data in its primitive form. Also allows one to extract a statistical report using pre-defined settings. Does perform somewhat complex operations on collected data and may therefore have extra overhead related to this.

Datasource and Datahandler

A datasource is a place in which data is accessible from. This could be a file or a URL. A datahandler is the implementation for handling data from a datasource. As this data is formatted in a specific way.

Datareport vs Reportcollection

DataReport is responsible for creating an instance of a class that extends the Statistics class. The parameters in Datareport has a report with the corresponding dataset(s). Reportcollection consists of all the different reports the framework has to offer and a reportbuilder.

5.1 API

This is the resulting implementation from defined scenarios that solve the initial problem correlated to data instantiation from datasets that are formatted in various ways.

In order to get started with this framework, the user hast to allow an object to be used as a dataobject. One needs to annotate that class with @DataObject shown under:

```
@DataObject
public class ShowAPIDTO { public final String string;}
```

It is assumed there is corresponding getters for each class field and a corresponding constructor. Where the constructor contains parameters that matches each class field.

Scenario 1 and 2 – Collect and instantiate data from a DataSource into DataObjects that is JSON or CSV formatted

Utilized data formatted in JSON:

```
[
  {"string": "thing1","int": 1, "double": 1.0},
  {"string": "thing2","int": 2, "double": 2.0},
  {"string": "thing3","int": 3, "double": 3.0},
  {"string": "thing4","int": 4, "double": 4.4}
]
```

Utilized data formatted in CSV:

```
"thing1",1,1.0
"thing2",2,2.0
"thing3",3,3.0
"thing4",4,4.4
```

Note: The classes must contain fields and a constructor that matches the data.

Step one is to define and build a datasource for the data, which describes where it comes from. This may either be a file or a URL. Something that is done like this:

From File:

```
String path = System.getProperty("user.dir") + "/files/DTOJson.json";
DataSource dataSource = DataSource.newResource().fromFile(path).build();
```

From URL:

```
String url = "https://someAPI.com";
DataSource resource = DataSource.newResource().fromURL(url).build();
```

The next step is to define a handler for that datasource. If the data is formatted in JSON or as CSV, one could utilize the built-in handlers showcased like this:

For JSON:

```
IHandle handler = new JSONHandler();
```

For CSV:

```
IHandle handler = new CSVHandler();
```

The following settings helps one to further customize the handling of CSV formatted data:

```
CSVHandler handler = new CSVHandler();
handler.setDelimiter(";");
handler.setSampleEachLine(true);
handler.skipEmptyLines(true);
handler.removeDoubleQuotes(true);
handler.removeSingleQuotes(true);
handler.isSingleColumn(true);
handler.skipLineIndexes(0);
```

Each setting has the corresponding effect:

- SetDelimiter: Sets the delimiter that separates each line into segments.
- SetSampleEachLine: Samples each line for the type definition.
- skipEmptyLines: Skips any line that is empty or only contains spaces.
- removeDoubleQuotes/removeSingleQutotes: Removes all quotes from each line, either double or single quotes.
- isSingleColumn: Every column is handled as one column. Thereby making it possible to for example read every word in a text file as individual objects.

Step three is to create a collector by utilizing a corresponding builder and sending in the previously defined datasource and handler as parameters. Which is done like this:

```
IDataCollector collector = DataCollector.newCollector(dataSource,
handler).build();
```

In order to collect data and instantiate that into corresponding objects annotated with @DataObject, one must call the method collectData(). This method will further store all dataobjects inside a tree structure which is stored, then kept inside the collector.

The final step is to retrieve the object instances representing the data is, shown here:

```
List<Object> objects = collector.getAllObjects();
objects.forEach(System.out::println);

Where the result from this is:
showcaseAPI.ShowAPIDTO@3b81a1bc
showcaseAPI.ShowAPIDTO@64616ca2
showcaseAPI.ShowAPIDTO@13fee20c
showcaseAPI.ShowAPIDTO@4e04a765
```

It is also possible to retrieve a list of strong typed objects, instead of a list of just objects:

```
 List<TrumpWord> list = collector.getAllObjects(TrumpWord.class);
```

Internally this will just cast to a list of that type, so if the object does not match the internal object a class cast error is thrown.

Scenario 3 - Specify class fields and constructor

It is possible to be more concise by telling which constructor and which fields to utilize whenever mapping data to that class. Something that is done like this:

```
@DataObject
public class ShowAPIDTO implements Comparable<ShowAPIDTO>{
    @DataField
    public final String string;
    @DataField
    public final int anInt;
    @DataField
    public final double aDouble;
    @DataConstructor
    public ShowAPIDTO(String string, int anInt, double aDouble) {
```

Note: The implementation of Comparable is just there to tell how to compare this object. As all instantiated objects are stored in a generic tree structure internally. From now on the used DataObject classes are these.

Scenario 4 - Extract all columns

This assumes that you have a collector. If not see the previous scenarios for more information.

Extract all columns by using class fields and its corresponding output:

```
var extractor = new DataExtractor<>(collector);
var columnMap = extractor.extractColumnsUsingFields();
columnMap.values().forEach(System.out::println);
```

```
-------Showcasing API - DataExtractor - Fields ------
[1, 2, 3, 4]
["thing1", "thing2", "thing3", "thing4"]
[1.0, 2.0, 3.0, 4.4]
```

When extracting with the class method all getters are called. It further calls all methods that begins with “get” and returns a primitive type. By convention this also means all methods that have no parameters. This yields the following result:

```
var extractor = new DataExtractor<>(collector);
var columnMap = extractor.extractColumnsUsingMethods();
columnMap.values().forEach(System.out::println);
```

```
-------Showcasing API - DataExtractor - Methods ------
[1, 2, 3, 4]
["thing1", "thing2", "thing3", "thing4"]
[1.0, 2.0, 3.0, 4.4]
```

Scenario 5 - Extract certain columns

This assumes that you do have a collector. If not see scenarios 1 and 2 for more information.

When extracting with specified strings the method will first try to find any fields that match. If none is found it will try to match any methods with that name. Example:

```
var extractor = new DataExtractor<>(collector);
var strings = Arrays.asList("string", "anInt", "aDouble");
var columnsUsingStrings = extractor.extractColumnsUsingStrings(strings);
columnsUsingStrings.values().forEach(System.out::println);
```

Which yields the following result:

```
["thing1", "thing2", "thing3", "thing4"]
[1, 2, 3, 4]
[1.0, 2.0, 3.0, 4.4]
```

Getting the inner class that holds the data is done like this:

```
Class<?> clazz = extractor.getDataObjectClass();
```

One might explicitly tell which class to utilize like the method under. This class is utilized in the rest of this scenario.

```
Class<ShowAPIDTO> clazz = ShowAPIDTO.class;
```

Using methods to extract data columns is shown here:

```
var methods = Arrays.asList(clazz.getMethod("getString"),
clazz.getMethod("getAnInt"), clazz.getMethod("getaDouble"));
var columnsUsingMethodsMap =
extractor.extractColumnsUsingMethods(methods);
columnsUsingMethodsMap.values().forEach(System.out::println);
```

```
[1.0, 2.0, 3.0, 4.4]
["thing1", "thing2", "thing3", "thing4"]
[1, 2, 3, 4]
```

It is further possible to use fields to extract data columns, like this:

```
var fields = Arrays.asList(clazz.getField("string"),
clazz.getField("anInt"), clazz.getField("aDouble"));
var columnsUsingFieldsMap = extractor.extractColumnsUsingFields(fields);
columnsUsingFieldsMap.values().forEach(System.out::println);
```

```
["thing1", "thing2", "thing3", "thing4"]
[1.0, 2.0, 3.0, 4.4]
[1, 2, 3, 4]
```

Another possibility is to only extract a single column by utilizing either a string, method or field. Shown here:

```
var columnUsingField = extractor.extractColumnFrom(fields.get(1));
```

Scenario 6 - Calculate simple average values on one dataset

Primarily the user needs to create a dataset manually with primitive number types. The Average class takes an array-type as a parameter. The class accepts double[], Double[], Number[] and List<Number>. All the methods return a double value.

```
Average avg = new Average(data);
avg.calcTotalSum();
avg.calcMidRange();
avg.calcMode();
avg.calcMedian();
avg.calMean();
```

Scenario 7 - Calculate simple statistical values on one dataset

To begin, the user needs to create a dataset manually with primitive number types. The SimpleStatistics class takes an array-type as a parameter. The class accepts double[], Double[], Number[] and List<Number>. All methods return a double value.

```
SimpleStatistics simpleStatistics = new SimpleStatistics(data);
simpleStatistics.calcStandardDeviationFromSample();
simpleStatistics.calcSampleVariance();
simpleStatistics.calcStandardDeviationFromPopulation();
simpleStatistics.calcPopulationVariance();
simpleStatistics.calcStandardErrorFromSample();
simpleStatistics.calcStandardErrorFromPopulation();
```

Scenario 8 - Calculate simple statistical values on two datasets

Firstly, the user needs to create two datasets manually with primitive number types since Correlation and Covariance calculations expect two arrays. The class takes arrays as parameters. Both classes accept double[], Double[], Number[] and List<Number>. Meaning a tight coupling between super constructors and all extensions. All the methods return a double value.

```
Correlation corr = new Correlation(data1
corr.calcCorrelationCoefficientFromSample();
corr.calcCorrelationCoefficientFromPopulation();

Covariance cov = new Covariance(data1, data2);
cov.calcCovarianceFromSample();
cov.calcCovarianceFromPopulation();
```

Scenario 9 - Create a simple statistics report, which supports calculations on one or two datasets

DataReport uses an array of different reports from ReportCollections and one to two manually created datasets. The different types of reports available are Average report, SimpleStatistics report and CovarianceCorrelationReport. CovarianceCorrelationReport requires two datasets, but you can apply two datasets on every report.

The method excecuteReport() returns a HashMap of the calculation method used and value. prettyPrintReport() will return a more readable report that is sorted by key values. The following code shows how to create a general report with the HashMap and the prettyPrintReport() output.

```
DataReport c1 = new DataReport(ReportCollection.getFullAverageReport(),
dataset1);
System.out.println(c1.executeReport());
c1.prettyPrintReport();
```

```
-------Full Average ReportCollection with executeReport Method applied-------

{Median, Dataset: 1=4.5, Total sum, Dataset: 1=33.0, Average, Dataset: 1=4.125, Mode, Dataset: 1=6.0, Mid range, Dataset: 1=3.5}

--------------------------ReportCollection--------------------------

Average, Dataset: 1                             4.125
Median, Dataset: 1                              4.5
Mid range, Dataset: 1                           3.5
Mode, Dataset: 1                                6.0
Total sum, Dataset: 1                           33.0
------------------------------------------------------------------------
```

The framework gives the user the option to create a customized report. This is done by obtaining the report-builder and choosing what calculations the custom report will consist of. Further, the user needs to build the report and specify if the calculations are to be applied on one or two datasets.

```
DataReport c2 = new DataReport(ReportCollection.getBuilder()
        .calcAverageMean()
        .calcSampleVariance()
        .calcCovarianceFromSample()
        .calcCorrelationFromSample()
        .build(),
        dataset1, dataset2);
c2.prettyPrintReport();
```

There is also the possibility to create a report with an extractor. This method assumes that the user has a collector. Here the user can specify if they want to createReportUsingFields() or createReportUsingMethods() if they need it.

```
var extractor = new DataExtractor<>(collector);
var report = extractor.createReport();
```

One could define custom report settings like this:

```
extractor.setReportOptions(ReportCollection.getFullAverageReport());
```

Scenario 10 – Calculate probability given models

The different types of calculations related to Binominal probability and Poisson probability takes a variable X the user wants to calculate the probability on. Each calculation returns a double value. Expected value and variance do not accept any parameters.

The Binominal Distribution class takes the number of attempts and the probability as parameters. The first is given as an int and the latter as a double.

```
BinominalDistribution bin = new BinominalDistribution(int, double);
bin.calcBinominalProbability(int);
bin.calCumulativeProbabilityLessThanEqual(int);
bin.calCumulativeProbabilityMoreThanEqual(int);
bin.calCumulativeProbabilityMoreThan(int);
bin.getBinominalCoefficient(int);
bin.calcExcpectedValue();
bin.calcVariance();
```

The Poisson Distribution class takes lambda (λ) and per unit of time as parameters with lambda given as an int and the latter as a double.

```
PoissonDistribution p = new PoissonDistribution(int, double);
p.calcPoissonDistribution(int);
p.calcCumulativeProbabilityLessThanEqual(int);
p.calcCumulativeProbabilityMoreThan(int);
p.calcCumulativeProbabilityMoreThanEqual(int);
p.calcExcpectedValue();
p.calcVariance();
```

Scenario 11 - Using CollectorPools

Firstly, the user needs to declare the paths like this:

```
String path = System.getProperty("user.dir") + "/files/showcaseAPI.csv";
String path2 = System.getProperty("user.dir") +
"/files/showcaseAPI2.csv";
List<DataSource> sourceList =
DataSource.newResource().fromFile(path).fromFile(path2).buildAll();
```

Remember that by chaining calls, one must call the method: buildAll(), as this will return a list of data sources. One may also declare all paths once which is shown below.

```
List<DataSource> sourceList = DataSource.newResource().fromFiles(path,
path2).buildAll();
```

Here the fromFiles method takes a vararg argument, making it able to also handle arrays.

Secondly one needs to create a collector pool by utilizing the previously declared sourceList like this:

```
var dataCollectorPool = DataCollectorPool.newCollectors(sourceList, new
CSVHandler()).buildAll();
```

It is further possible to declare a mapping from individual sources to individual handlers shown here:

```
Map<DataSource, IDataHandler> map = new HashMap<>();
map.put(sourceList.get(0), new CSVHandler());
map.put(sourceList.get(1), new JSONHandler());
```

After this is done, the user is able to collect and instantiate data shown below.

```
var dataCollectorPool = DataCollectorPool.newCollectors(sourceList, new
CSVHandler()).buildAll();
```

It is also possible to utilize custom handler to map each datasource, like this:

```
dataCollectorPool.collectAllData();
dataCollectorPool.iterator().forEachRemaining((collector) ->
System.out.println("Collector: " + collector.getAllObjects() + "\n"));
```

```
Collector: [showcaseAPI.ShowAPIDTO@32e6e9c3, showcaseAPI.ShowAPIDTO@5056dfcb, showcaseAPI.ShowAPIDTO@6574b225, showcaseAPI.ShowAPIDTO@2669b199]

Collector: [showcaseAPI.ShowAPIDTO@2344fc66, showcaseAPI.ShowAPIDTO@458ad742, showcaseAPI.ShowAPIDTO@5afa04c, showcaseAPI.ShowAPIDTO@6ea12c19]
```

Scenario 12 - Using ExtractorPools

Note that this builds on the previous scenario. Primarily the user needs to create a data extractor pool and extract either all or specified columns by doing this:

```
IDataExtractorPool dataExtractorPool = new
DataExtractorPool(dataCollectorPool);
var res = dataExtractorPool.extractAllColumnsFromFields();
res.forEach((k, v) -> {
    System.out.println("key: " + k  + "\n\t\tValues: ");
    v.forEach((k1, v1) -> System.out.println("\t\t\t\t" + v1));
});
```

It is further possible to utilize a map to specify which methods or fields to use from each class. Shown here:

```
dataExtractorPool.extractAllColumnsFromFields(@NotNull Map<Class<?>,
List<Field>> classListMap)
dataExtractorPool.extractAllColumnsFromNames(@NotNull Map<Class<?>,
List<String>> classListMap)
dataExtractorPool.extractAllColumnsFromMethods(@NotNull Map<Class<?>,
List<Method>> classListMap)
```

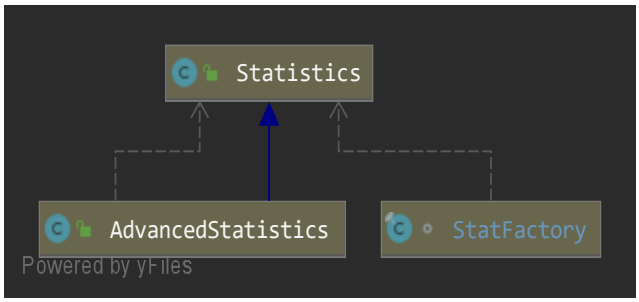
Under is the following output. Note that each class referenced in the map gives its corresponding fields which in turn is mapped to a list, which contains the actual data for that field(column).

```
key: class showcaseAPI.ShowAPIDTO
        Values:
                ["thing1", "thing2", "thing3", "thing4", "thing5", "thing6", "thing7", "thing8"]
                [1.0, 2.0, 3.0, 4.4, 5.0, 6.0, 7.0, 8.4]
                [1, 2, 3, 4, 5, 6, 7, 9]
```

Additional functionality provided by the framework

During the development, the following functionality is available for public usage. Thereby adding to the current functionality. Said functionality includes different tree implementations and standard tree traversal types implemented as iterators. The trees supported are AVLTree, RBTree, and BSTree. The user also has access to parse string values to their primitive type counterpart. The Parse class can also check if a class is a primitive type/number and gets a comparator for given primitive type or its corresponding wrapper type. In addition, this the framework offers a way of reading and writing to and from a file and URL. Due to probability calculations, combinations and permutations calculations are also available.

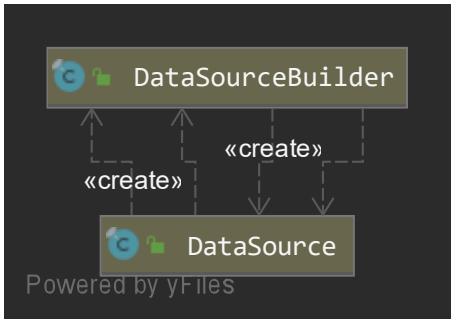
5.2. Utilized Design patterns



Factory method pattern is found in the class StatFactory which instantiates a corresponding statistics class using a corresponding constructor and a corresponding class. This is further done dynamically as the framework finds a corresponding class via registered report methods and inheritance.

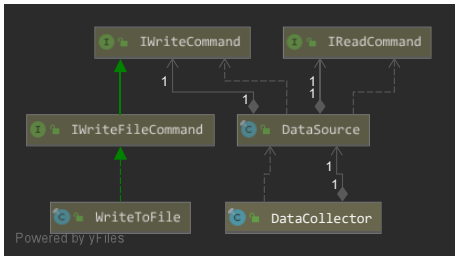
The pattern abstracts away any explicit way of instantiating a certain class in different locations inside the code. This is done by the factory, and therefore leaves less work for everybody else. This further makes it easy to expand by adding another class that extends Statistics class and has appropriate constructors. As all retrieved classes from the factory is compatible due to inheritance. Thereby decoupling implementation details away from other parts of the framework.

By having all instantiation in one place we are following the Single Responsibility Principle. Which helps with maintainability as the class now only has one reason to change. But this pattern adds to the overall complexity due to it having more interfaces and classes than would be necessary. Though this tradeoff is very much worth it.



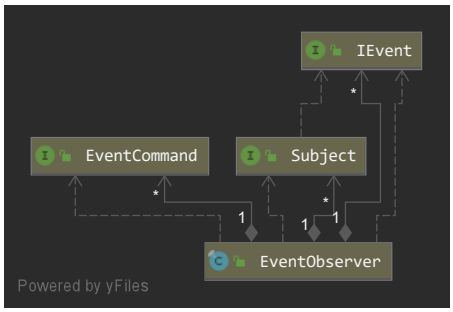
Several places utilize *the builder pattern* as this helps to abstract away complex settings and parameter lists, which in turn may affect how the class initially works. An example of this is seen in DataCollector class. As it has several settings that influences how the collection of data is performed. By setting these beforehand it allows the user to get more predictable results. This in turn, allows for highly customized classes.

This pattern is also utilized in on the following classes: DataExtractor, DataExtractorPool, DataSource, ReportCollection – to build a report array.



A very simple usage of *the command pattern* is found in Resource where it is used for the ReadCommand to execute the action: read. This alleviates and abstracts away the complexity of always knowing implementation details. Allows one to place custom callbacks for reading from a file or from a URL.

This can easily be expanded to support multiple and different commands and does not further require changes on existing classes. One way to expand on this is to allow other commands like saving information or deleting all information to that resource. The same pattern is used for the different DataHandlers that handle formats like CSV and JSON. This allows the client to utilize their own method for handling data.



Observer pattern is used for observing hidden exception events inside the framework. These are all available to be observed by clients as well as executing a callback command whenever one happens. The observer observes other events as well, like whenever a DataCollector finishes collecting information or whenever a DataExtractor finishes its job. The observer will automatically broadcast to all its registered subjects.

This is particularly useful whenever one wishes to execute a callback function on completion during async execution. This pattern further decouples each subject from the observers detailed implementation and makes it possible to switch between different observers as well as different subjects indifferently. Which is a side effect from using interfaces and their main applicability.

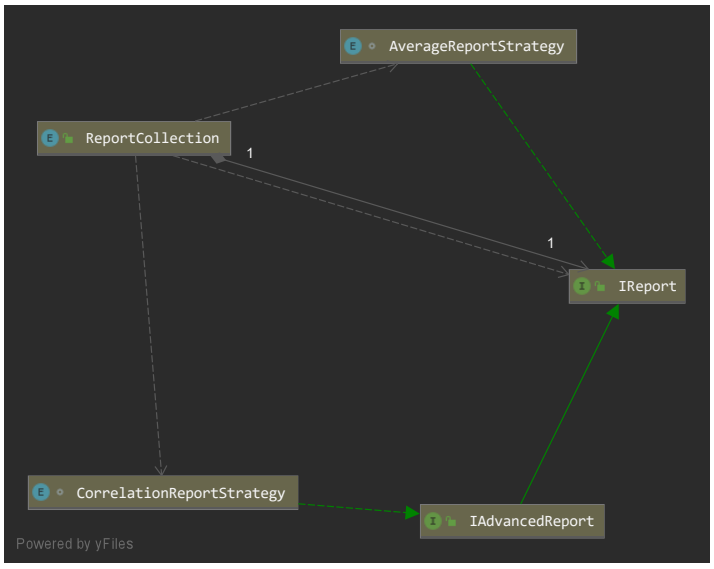
Another addition is that it makes both the observer and the subject easy to extend due to inheritance. Though this in turn may cause a cascade of updates as the observer does not know concrete implementation details for each subject. One update might therefore lead to another.

It was chosen to utilize this pattern for allowing multiple collectors to be run async. Thereby allowing this pattern to update and react on every subject's behalf. Which is something that the client will specify. Something that allows the framework to handle the async aspect for the client. Both the subjects and the observer may therefore belong to different layers in this framework.

Complex pattern composition (Statistics package)

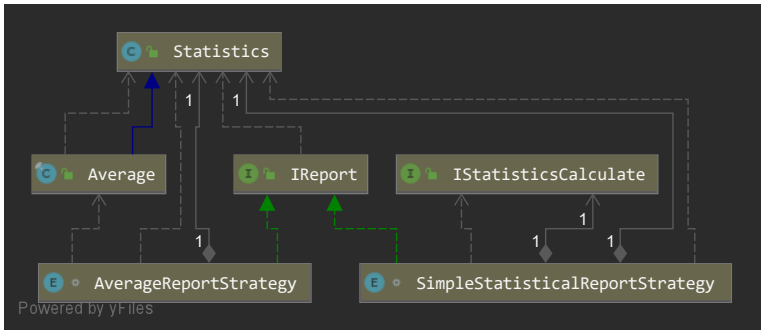
Utilized patterns in this package are: factory method, builder method, facade, and strategy pattern. These all work in conjunction in such a way that all pros from each pattern is kept, whilst limiting the number of cons.

A builder pattern is used for interfacing and building a set of custom reports based on the different enums. This abstracts away all concrete details about the underlying enum (ReportCollection). The builder returns an array of report enums which is used by DataReport in order to further find and delegate the correct calculation.



This brings us to another pattern: *Facade*. This pattern is clearly seen in ReportCollection as this class holds information and details regarding all other report enums. Which helps to abstract away any subsystem details. The pattern allows for weak coupling, which in turn allows one to expand and/or remove without making major changes. Another feature is that the pattern does not prevent direct access to each subsystem, thereby allowing for both simple and more advanced usage.

Facade pattern is a pattern that provides a simplified interface for a set of complex classes, thereby shielding the client from subsystem details. Which in this case, is all strategy classes. Something that enables weak coupling between different subsystem modules. Which might limit and/or remove circular dependencies and thereby making it easier to port this framework to other systems for future usage.



Each subclass Strategy, like AverageReportStrategy, utilizes *the strategy pattern* where each enum specify the method to utilize for the calculation. Thereby having each enum describe their own calculation method.

Strategy pattern encapsulates the different implementations/algorithms and allows one to vary implementation on demand. Thereby making a family of related algorithms contained in one place for access. which eliminates the many condition statements that would have been required throughout the code.

A small downside is that the client must know about the different strategies (Report types), though this is taken care of, by using a builder. This pattern might lead to an increased number of objects inside the framework, which in turn leads to more memory usage. Another downside is overhead related to the communication between the context and the concrete strategy.

It might seem as there is many downsides to this pattern, but the impact on efficiency in our framework is not noticeable. Something that is due to it only handling a small amount of strategy objects at any given time.

All these patterns highly intertwined and together forms a very complex integration. Thereby resulting in added complexity which might be difficult for others to understand. Though this also allows one to gain most, if not all, of the positive sides of each pattern. Thereby making it both easier and more efficient as things like potential large overhead, complex integrations, and large list of configurations are abstracted away. In addition to this one can follow the SOLID principles, which in turn furthers the expandability of the framework.

6. Discussion

6.1 Further development

The current framework solution could be further expanded in several ways. A more extensive point would be to utilize namespaces in order to allow for two-way communication on a desired object. Which in turn may be mapped to a given resource or n resources. Another similar development would be to allow one to name the annotated class fields. Such that they may be utilized by other parts in the framework. Like whenever one is extracting data from a class or collecting data from a dataset.

The statistics package currently only has a few of the most important and basic calculations in statistics when in theory it should be a much larger package with access to more complex calculations. The package and all its functionality have much to improve upon. A way of improving this would be to mitigate the calculation done inside the statistical module. Something that may be done by being aware of pre executed calculations. Thereby removing redundant calculations and making it more efficient to calculate the different statistics. Another way of improving this would be to incorporate and utilize apache commons math3 instead of creating our own calculation methods. Thereby alleviating redundant calculations, whilst getting access to many more complex ones. Which in turn reduces future development time.

Right now, there are few predefined statistical reports available, but could easily be expanded upon. Since the statistics package utilizes a complex design pattern composition it is easy to create new reports as more statistics and probability calculations become available in the framework.

The stability of the framework and its robustness could be improved by creating more tests to support the various features. Which in turn would help to make it easier to expand and refactor existing functionality. A second way would be to add further heuristics for picking the correct data type inside the handlers. Currently it only chooses the first one and last one in the dataset. Which potentially leads to picking the wrong data type.

6.2. Work report

Each person was given dedicated tasks and responsibilities that had deadlines. Main responsibility for both delegating tasks and executing administrative work was given to Mathias W. Nilsen. In addition to this, he was given the main responsibility for creating the underlying architecture for the whole project, which also includes architecture design for each individual package. As well as overall functionality for the following packages: core and utilities.

Maria E. Pedersen was given full responsibility for the statistics package and all its functionality. In addition to this, she was responsible for following up on finished tasks. Other responsibilities included: refactoring and creating tests for implemented functionality. Robert Alexander Dankertsen was the one to refactor and supplement tests for implemented functionality. Another responsibility included bug testing finished tasks.

Even though each group member was given main tasks, everyone helped each other when needed and worked on different parts of the framework that they were primarily assigned. Because of this it was easier to share a full report, since splitting the workload into different reports did not really work.

The work was done in iterations where each iteration consisted of a set of predefined tasks and lasted 1 week. Each week further had 2 group meetings each lasting 2-4 hours. Though these meetings quickly became shortened due to the outbreak of COVID-19. Thereby restructuring both weekly and daily tasks. The new structure consisted of a more frequent and dynamic schedule of meetings as well as cutting said meetings to sessions lasting 10-60 minutes. Something that did affect the overall progression of the framework.

Change of lecturer led to a change of curriculum focus which in turn led to reconstructing and removing certain features from the framework. Something that hindered the overall progression, making it harder to dedicate and create new tasks that the individual was able to do.

This assignment allowed us to reflect and gain experience working as a group. As each person could draw knowledge from each other. The group did not encounter any major problems or major discussion during the assignment.

Sources

Apache Software Foundation. (n.d.). Apache Commons CSV User Guide. Retrieved May 13, 2020, from http://commons.apache.org/proper/commons-csv/user-guide.html

Apache Software Foundation. (n.d.). Apache Commons Math. Retrieved May 13, 2020, from http://commons.apache.org/proper/commons-math/download_math.cgi

Google. (2020, May 2). GSON. Retrieved May 13, 2020, from https://github.com/google/gson

Jacksons. (2020, May 11). FasterXML/jackson. Retrieved May 13, 2020, from https://github.com/FasterXML/jackson

Jackson. (n.d.). Jackson Annotation. Retrieved May 13, 2020, from https://www.baeldung.com/jackson-annotations