



Reimplementation of an algorithm to calculate pairing lists in the environment of complex sports events

Projektarbeit 1 (T1_2000)

im Rahmen der Prüfung zum
Bachelor of Science (B.Sc.)

des Studienganges Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Raphael Müßeler

Januar 2018

-Sperrvermerk-

Abgabedatum:	01. Oktober 2018
Bearbeitungszeitraum:	16.10.2017 - 31.10.2018
Matrikelnummer, Kurs:	4508858, TINF17B2
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	Axel Uhl

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit 1 (T1_2000) mit dem Thema:

Reimplementation of an algorithm to calculate pairing lists in the environment of complex sports events

gemäß § 5 der "Studien- und Prüfungsordnung DHBW Technik" vom 29. September 2015 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den August 21, 2018

Müßeler, Raphael

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland

Sie darf als Leistungsnachweis des Studienganges Angewandte Informatik 2017 an der DHBW Karlsruhe verwendet und nur zu Prüfungszwecken zugänglich gemacht werden. Über den Inhalt ist Stillschweigen zu bewahren. Veröffentlichungen oder Vervielfältigungen der Projektarbeit 1 (T1_2000) - auch auszugsweise - sind ohne ausdrückliche Genehmigung der SAP SE nicht gestattet.

SAP und die SAP Logos sind eingetragene Warenzeichen der SAP SE. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in dieser Arbeit berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedem benutzt werden dürfen.

Contents

Abkürzungsverzeichnis	V
List of Figures	VI
List of Listings	VII
1 Introduction	1
1.1 Sailing software solutions	1
1.2 Global Sponsorships	1
1.3 Motivation	1
1.4 Outlook on this work	2
2 Requirements	3
2.1 What exactly is a pairing list?	3
2.1.1 Regattas and series	3
2.1.2 Pairing list template	4
2.2 Former solution	4
2.2.1 Quality factors	4
2.2.2 Understanding the MatLab algorithm	6
2.2.3 Disadvantages of the MatLab algorithm	8
3 Definition of the project	10
3.1 Algorithm requirements	10
3.2 Requirements of the UI	10
4 Architecture and Tools	11
4.1 Development landscapes	11
4.1.1 Git	11
4.1.2 Hudson	12
4.1.3 Eclipse IDE	12
4.1.4 Tests	12
4.2 Runtime architecture	12
4.2.1 GWT	12
4.2.2 OSGi	13
4.3 Other tools	14
4.3.1 BugZilla	14
5 Realization	15
5.1 Implementation	15
5.1.1 PairingListTemplate	15
5.1.2 The actual pairing list implementation	16
5.1.3 Template factory	16
5.1.4 PairingFrameProvider and CompetitionFormat	16

5.1.5	CompetitorAllocations	16
5.2	Development of the main algorithm	17
5.2.1	Forming concepts of concurrency	17
5.2.2	Prefixes	18
5.2.3	Different suffixes	20
5.2.4	A new concept	20
5.2.5	Comparison of all concepts	21
5.3	Extensions and improvements of the algorithm	21
5.3.1	Distribution of competitors among the boats	21
5.3.2	Reducing the boat changes	22
5.3.3	Empty spaces	22
5.3.4	Flight repetitions	22
5.3.5	Incremental calculation of the association matrix	23
5.3.6	More ideas or extensions	23
5.4	Integrating in the UI	23
5.4.1	Admin Console	23
5.4.2	Interaction between back end and front end	23
5.4.3	Dialogs	24
5.5	Additional features	25
5.5.1	Exporting template to CSV	25
5.5.2	Filling the race logs	26
5.5.3	Print functionality	27
6	Conclusion and outlook	28
6.1	Conclusion	28
6.2	Outlook	28
	Bibliography	VIII

List of abbreviations

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CI Continuous Integration

DTO Data Transfer Object

DVCS Distributed Version Control System

GWT Google Web Toolkit

IDE Integrated Development Environment

JVM Java Virtual Machine

RPC Remote Procedure Call

UI User Interface

UML Unified Modeling Language

VCS Version Control System

List of Figures

2.1	Example for a pairing list	3
2.2	Example for an association matrix	7
4.1	Illustration of the distributed version control system (https://git-scm.com/figures/18333fig0103-tn.png , Jan 2018)	11
4.2	Layers of the OSGi framework (https://www.osgi.org/wp-content/uploads/layering-osgi.png , Feb 2018)	14
5.1	Implementation of the feature as UML (latest version)	15
5.2	Prefix creation with backtracking-principle	18
5.3	Setup Dialog	24
5.4	Dialog for the purpose of displaying the PairingListTemplate	25

List of Listings

2.1	Algorithm of shuffling competitors within a group	7
4.1	AsyncCallback interface	13
5.1	<i>createPrefix()</i> method	19
5.2	allSeeds array	20
5.3	PairingListLeaderboardAdapter class	26

1 Introduction

1.1 Sailing software solutions

Sailing sports is not as popular as football or athletics, yet there are world championships and similar competitions. But what does software have to do in sailing sports? Sailing is a very complex team sport that even requires a license, since a sailor has to know all about maneuvers, inshore waters, wind etc. So there is a lot of potential for software to be useful.

SAP Sailing provides several software solutions that are respectively responsible for their field of activity. The basis of these solutions is GPS tracking as well as the wind measurements. To give an illustration for areas of responsibilities, the 'SAP Sailing Analytics' solution deals with real time analytics around race rankings, speeds, maneuvers etc. of live sailing events. Spectators can follow the sailing events on the Internet and sailors can watch their statistics and analytics as well as optimizing strategies. Besides the solution of SAP Sailing helps event managers to manage such sailing events and much more. By dint of several android and iOS applications, the world of sailing competitions gets more and more digitized.

1.2 Global Sponsorships

The **Global Sponsorships** department provides show-case solutions for various sports such as basket ball, tennis, sailing etc. but also in the entertainment sector such as for the Cirque du Solail or the Elbphilharmonie Hamburg. The sub-department I have worked in has developed the SAP Sailing solutions. For about six years this part of the Global Sponsorships department deals primarily with the sailing sports. However, other sports like tennis or riding are currently in the process of being developed.

1.3 Motivation

As I will explain in the following paragraphs, creating pairing list before implementing this feature has been a big obstacle for the user. The team I have worked in aims to simplify the process of generating pairing lists and furthermore to get in touch with big projects such as SAP Sailing Analytics. Besides that improving my skills in Java and of course coding within a team were reason as well for joining the project.

1.4 Outlook on this work

In the following I will be responsive to requirements that are necessary to understand the rest of this work. With the next chapter I will depict all requirements that the feature has to fulfill in the end. After explaining the most important points of the SAP Sailing Analytics' architecture, I will explicate the process of the algorithm as well as the UI component. Since I have not dealt with the algorithm exclusively, I will split up this work into about 70% back-end and 30% front-end.

2 Requirements

2.1 What exactly is a pairing list?

In sailing, pairing lists are used to structure competitions. Within these competitions - also called **regattas** - there are different participants competing against each other. Since the host of such a competition event provides a certain number of boats, the number of competitors who can sail at the same time is limited. Therefore, the competitors are divided into **groups** or **fleets** and **flights** or **races** (in the following I will use the terms groups and flights due to consistency and usage within the code). Within a flight, every competitor sails only once. The number of competitors per groups is determined by dividing the total number of competitors by the number of boats.

Flight 0	Group 0	5	1	3	4
	Group 1	0	9	10	2
	Group 2	7	11	8	7
Flight 1	Group 3	6	11	9	2
	Group 4	8	7	4	3
	Group 5	1	5	10	0
Flight 2	Group 6	5	9	2	8
	Group 7	10	6	7	11
	Group 8	4	3	0	1

Figure 2.1: Example for a pairing list

The difficulty in creating such pairing lists now lies in the fact that each competitor sails against each other as often as possible. I will describe further requirements for the algorithm in chapter 2.2.1.

2.1.1 Regattas and series

Regattas in general describe competitions within sailing sports. Regattas are split in several **series**. There are different categories of series, for instance Medal or Finale series in which competitors can be eliminated. However, creating a pairing list for a Medal Series e.g., is only useful if the previous series is already finished, otherwise the competitors that will participate in the Medal series are unknown. Series are sequences of flights.

2.1.2 Pairing list template

When talking about pairing lists, in most cases the term refers to its template. A **pairing list template** in general is actually just a table of numbers in which each row represents a group and each number within a row represents the number of the competitors. Moreover the table is separated into packages where the package's size corresponds to the number of groups. Its scheme is described in 2.1.

As a pairing list with numbers can be applied to different competitions that have the exact same number of competitors, boats, groups and flights, it is called template. After creating such a template, each number is assigned randomly to a specific competitor.

2.2 Former solution

The previous solution has been an external tool implemented in **MatLab**. MatLab is mathematical software that provides the ability of creating diagrams, analyzing data and everything that goes along with that. Furthermore MatLab is also a computer language, that is specialized on mathematical operations and functions.

In the following I will explain how the MatLab algorithm works, because it is essential for the understanding of this work.

2.2.1 Quality factors

As there is the problem of comparing two different pairing lists, factors that describe a pairing list were defined. The quality of a pairing list is given by the three following factors:

1. The most significant factor that numerically describes how good the returned pairing list is, is determined by how evenly the competitors are spread over the groups. In fact this is measured by the frequency of encounters, with which the competitors meet.

Since every single encounter has to be considered and the individual pairing lists are to be compared using a single value, the mathematical operation standard deviation is used. The following applies:

$$s = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2} \quad (2.1)$$

n is the number of elements (in this case the number of total competitors), and \bar{x} is the

empirical mean for which the following applies:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (2.2)$$

The empirical mean, in this case, indicates the average number of encounters. Considering that, specifies the standard deviation the average difference to the empirical mean. It should be mentioned that the lower the value of the standard deviation, the better the competitors are distributed among every single flight.

2. Secondly the algorithm must ensure that every competitor sails equally frequent on each boat, so that at the end of this regatta nobody can appeal against the distribution of competitors onto boats. Here, too, the standard deviation of the frequency of encounters is determined, except that it is not the encounters between the competitors that are considered, but rather the encounters of competitors and boats.
3. Since a group starts at specific point on water, it gains more expenditure the more boat changes there are. Another objective of the algorithm was to minimize the number of boat changes between each flight.

However, this stands in contrast to the previous quality factor. If the algorithm would distribute the competitors onto the boats equally (which cannot be accomplished due to the approximation of each encounter of the competitors with the boats to the empirical mean), there will be no boat changes from one flight to the next.

However, the algorithm, that minimizes the number of boat changes, only finds a local optimum and is therefore a *greedy algorithm*. In order to achieve the global optimum, the entire solution space must be searched for all possible permutations $P_{n_g} = n_g!$. For the minimal number of boat changes Δb applies:

$$\Delta b = \frac{n_c}{n_g} - \left(\frac{n_c}{n_g} \frac{1}{n_g} \right) = \frac{n_g n_c - n_c}{n_g^2} \quad (2.3)$$

n_c is the number of competitors, whereas n_g describes the number of groups. The mathematical expression $\frac{n_c}{n_g}$ represents the number of competitors per group.

These factors are essential for the whole algorithm and even though the algorithm has developed a lot, the quality factors are unaltered, since it is a well-tried measuring technique for the quality of a pairing list.

2.2.2 Understanding the MatLab algorithm

Since the owner of the MatLab algorithm has found no 'systematic' way of calculating such pairing lists. In fact the algorithm follows the Monte Carlo principle, i.e. it is a randomized and non-deterministic algorithm.

The algorithm tries a given number (**iterations**) of pairing lists to find out the best one. The number of iterations is set to 1.000.000, which is an empirical value. It turned out later, that the number of iteration can be reduced to 100.000, since it reduces the time that the algorithm needs to calculate a pairing list.

Each iteration the pairing list template will be filled flight by flight as follows:

The first competitor of the first group is set by random. It is called **seed**, because it affects the following generation of groups. In order to calculate a well distributed pairing list, there must be something to establish it on. At this point the trial-principle comes into play. The number of combinations for a single pairing list correlates with n^m , whereby n is the number of competitors and m is the number of flights. It turned out that the quality completely depends on the seeds, because from here on, the algorithm is deterministic. It is also possible to have the same combination twice in a single runtime.

After generating a random seed, the algorithm now picks the next competitor. In order to ensure that the best possible competitor is chosen, every competitor will be checked as follows:

1. Firstly, the algorithm checks whether the pairing list contains the respective competitor. This is necessary to warrant that every competitor occurs only a once within one flight.
2. Secondly, it is being checked whether the sum of all encounters, that the current competitor has to the others, is the smallest of all. Thus it is ensured, that competitors who have less encounters will be selected. If two competitors with the same number of encounters appear, the first one will be taken.
3. At last, the algorithm looks for the lowest maximum of encounters that the respective competitor has, so that competitors that have many encounters with the other competitors within the current group, will not be picked as next.

After setting a competitor, the association-matrix will be updated by incrementing the specific row of the competitor by 1. The figure 2.2 gives an example of how such an association matrix could look like.

According to this process, the pairing list is being filled flight by flight and group by group. At the end of a single iteration, the outcome of this is a calculated pairing list.

Each iteration, the quality of the current pairing list is determined and then compared to the quality of the current best pairing list. If it is better, the best pairing list will be replaced. This

	1	2	3	4	5
1	-	2	3	1	4
2	2	-	2	4	3
3	3	2	-	3	2
4	1	4	3	-	1
5	4	3	2	1	-

Figure 2.2: Example for an association matrix

ensures that at the very end of the algorithm, the best pairing list is returned.

Shuffle competitor positions

The defined quality factor of a fair distribution of competitors onto boats is handled as well, but in a non-deterministic manner. To illustrate how the assignment of competitors to boats is handled, the index of the competitor within a group specifies the boat index. In the following method (implemented in MatLab) only the order within the groups will be shuffled.

```

1  function [PShuffled] = shuffle(POld)
2      %SHUFFLE individually permutes all m rows in a m by n matrix
3      %NO further details
4      m = size(POld,1);
5      n = size(POld,2);
6      PShuffled = zeros(m,n);
7      %Perform permutation for every row 1 to m
8      for i=1:m
9          randOrd = randperm(n);
10         for j=1:n
11             PShuffled(i,j)=POld(i,randOrd(j));
12         end
13     end

```

14 **end**

Listing 2.1: Algorithm of shuffling competitors within a group

Here it should be underlined that the expenditure of time is quadratic. The time complexity of the algorithm is proportional to:

$$T_{shuffle}(n) \in O(n^2) \quad (2.4)$$

2.2.3 Disadvantages of the MatLab algorithm

There are several disadvantages that go along with the MatLab algorithm. In the following I will list the most important ones:

- **Integration in the project:** Since the algorithm is implemented in MatLab, there is no integration with the rest of the tools. Integrating the algorithm into the project is followed by several benefits: First of all the usability will not be impaired and restricted anymore. For instance it is not accessible from the Internet and not applicable to the user yet. In addition the usage of Excel-Sheets is no more necessary, so the user can display the pairing list and its action within the project. Furthermore the completed pairing list can be applied to the race logs. This implies that the content of the pairing list will be saved in database for building persistence.
- **Time and time complexity:** The time that the algorithm took to generate new pairing lists can be up to five or six minutes (those values are taken from specially measured test runs after transferring the MatLab algorithm into the whole project and translating it to Java code). Since the usage of the algorithm might be frequent, it is very urgent to improve the algorithm's time to run.

Moreover, for the time complexity of the algorithm applies the following:

$$T(n_i, n_f, n_g, n_c) \in O(n_i \cdot n_f \cdot n_g \cdot n_c) \quad (2.5)$$

n_i is the number of iteration that the algorithm tries out, whereas n_f , n_g and n_c describe each the number of flights, groups and competitors.

Obviously this time complexity ought to be reduced, because the time complexity consists only of factors, not addends. In addition to that high time complexity of the basis algorithm, other methods are called that produce expenditure of time as well. Moreover, the whole algorithm processes sequentially and not in parallel (I will explain on that point later).

- **Unstructured proceeding of the algorithm:** According to trial principle, some combinations might occur twice as already mentioned. There might be a systematic way of solving this problem, so that the algorithm achieves a higher performance. In addition the shuffle-method follows no systematic way at all, as well as the generation of a pairing list.
- **Expandability:** Another primary problem with the algorithm is that it cannot be expanded. Since there is no integration, the functionality and the usability are static in its manner. Features that the customer desire, cannot be applied.
- **Costs:** Besides all that MatLab requires licenses that are cost-intensive. These expenses could be saved.
- **'Clean Code':** Within the MatLab algorithm, variables and methods are not well labeled, so the maintainability for a developer is very time consuming and limited. Furthermore there are just a few comments and those are not articulated well. According to the 'Clean Code'-principle, this problem should be solved.

3 Definition of the project

There are several demands that I will list in the following ordered by priority:

3.1 Algorithm requirements

1. The main aspect, the algorithm should comply with, is that the returned pairing list ought to be of a good quality. Since this feature is also used by event managers, the fair distribution of competitors to fleets is essential.
2. Secondly, the performance of the algorithm should be fast enough for the pairing lists to be calculated from the UI within a acceptable time frame.
3. In every sport there are leader boards, leagues or something. Since the department Global Sponsorships tries to reach out to other sports in the future like e.g. Tennis or riding, this algorithm might be applied to these sports. This means for the implementation, it should be kept abstract.

3.2 Requirements of the UI

1. The user should be able to easily generate pairing lists. Furthermore the user should be able to not only show the pairing list template (I will later explain this in detail) but also the actual pairing list with the allocated competitors.
2. In addition the user should have an export possibility for the template. This might be important for exporting this template into an Excel-Sheet, since some customers actually work like this.
3. The pairing list with the competitors should be displayed printable. This is important, because costumers prefer using a paper sheet at sea rather than an electronic device.

4 Architecture and Tools

4.1 Development landscapes

4.1.1 Git

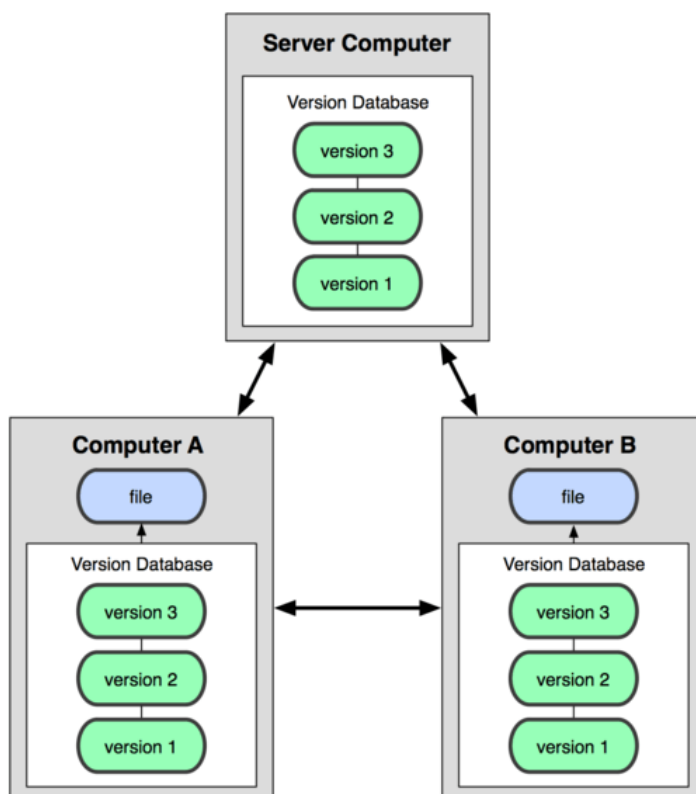


Figure 4.1: Illustration of the distributed version control system (<https://git-scm.com/figures/18333fig0103-tn.png>, Jan 2018)

tory with all snapshots. In case of damage of the server, all users still have the latest version and it can easily be restored.¹

Furthermore Git provides several more features, however, I will not highlight those like e.g. branches or tags, which allow multiple developers to keep track of all changes.

Git is a version control system that manages files and their history. Git is a main component within the project and it forms the base in the process of developing. Git allows to restore older versions of the project and it provides the ability for developers to work on different places within the project at once. Opposed to other version control systems (VCS), Git is a distributed version control system (DVCS) and it works as follows:

Each time a user created new version of the local repository called snapshot e.g. after finishing a feature or fixing a bug, it can be uploaded (called push) to the server. If another user now wants to get the latest snapshot, he easily pulls it from the server. But instead of storing only the latest version locally, the user get the whole repository with all snapshots.

¹see Git and Software Freedom Conservancy, *Git - About Version Control*.

4.1.2 Hudson

The Hudson tool focuses mainly on two jobs: First Hudson builds and tests software projects continuously. In other words, Hudson provides a continuous integration system (CI). This makes it quite easy for the developers to integrate new features or changes. Besides test failures are recognized earlier than in a productive system. In addition Hudson provides detailed error messages and screenshots that are recorded when an UI test fails.

Furthermore Hudson deals with monitoring executions of externally-run jobs. This means that registered developers are being informed if a build or a test failed. This allows the developer to react fast and fix the problem.²

4.1.3 Eclipse IDE

Eclipse is an integrated development environment (IDE) and mostly known for the Java IDE. However, the Eclipse Foundation provides other IDEs e.g. for C++ or PHP. The advantage that goes along with developing in Eclipse IDE, is the Eclipse Marketplace allows the developer to add several extensions and customizations.

4.1.4 Tests

The SAP Sailing solution deals with two kind of tests: Firstly the JUnit framework is a Java-based testing framework. It is used to test separate units like methods or classes.³

Secondly there is Selenium framework. It provides automated software tests of web applications. By dint of Selenium tests, specific scenarios that describes a sequence of actions in the UI can be tested.⁴

4.2 Runtime architecture

4.2.1 GWT

The Google Web Toolkit (GWT) is an open source web framework, which provides the ability to create web applications in Java. The communication between client and server works with the help of so called Remote Procedure Call (RPC) services and serialization. RPC allows for objects to be sent via the standard Hypertext Transfer Protocol (HTTP). The result will be returned as an asynchronous call which is a core functionality of AJAX (Asynchronous JavaScript And XML) development.⁵

²see Eclipse Foundation, *Hudson - Meet Hudson*.

³see JUnit, *JUnit 5 - User Guide*.

⁴see ThoughtWorks, *Selenium - Web Browser Automation*.

⁵see Google, *[GWT] Documentation - Client*.

Serialization

Serialization in general is the process of translating objects states or data structures to something that can be saved into a database or passed into RAM and it provides the ability to translate it back. Main use case is to restore a specific state of an object. This is used e.g. as an interface between client and server, as somehow back end data ought to be passed to the front end. Within in the project, serialization is important in context of RPC. So before sending such an object to the client or the server, it is serialized and compressed, so that it can be passed to the receiver.

In order to work with serialization in Java, the interface `java.io.Serializable` must be implemented by the respective class that will be serialized. Each class, that can be serialized, must have a `serialVersionUID` that is used to verify that the sender and the receiver of the serialized object have loaded classes for the object that have the same *serialVersionUID*.⁶

Asynchronous calls

As already mentioned such RPC services work with asynchronous call. Such calls are method calls or the like which are working parallel to the actual runtime or rather asynchronous calls are the basis for a non-blocking programming model.

To make this asynchronous programming work, an interface is required. In the SAP Sailing Solution there already is such an interface called *AsyncCall* that provides two methods: *onSuccess()* and *onFailure()*. The principle of asynchronous calls is comparable to the event handling in Java.

```
1 public interface AsyncCall<T> {  
2     void onFailure(Throwable caught);  
3     void onSuccess(T result);  
4 }
```

Listing 4.1: AsyncCallback interface

4.2.2 OSGi

The OSGi framework provides a concept for modularization of Java applications and its services using an component model. However, this assumes a Java Virtual Machine (JVM). This component model is defined as a component registry or rather a service registry, whereby in this context the definition of service is rather to equate with the definition of interfaces. OSGi foregrounds the components (bundles) that publish its interfaces (services) via registry locally on the JVM. Besides it supports the redeployment of a component's life cycle.

⁶see Oracle, *Oracle Java Documentation - Serializable Objects*.

The OSGi framework allows to dynamically run or even update service applications, so called bundles, during runtime. This enables to run different and independent applications in parallel on the same JVM as well as to administrate and update the whole life cycle of the application.⁷

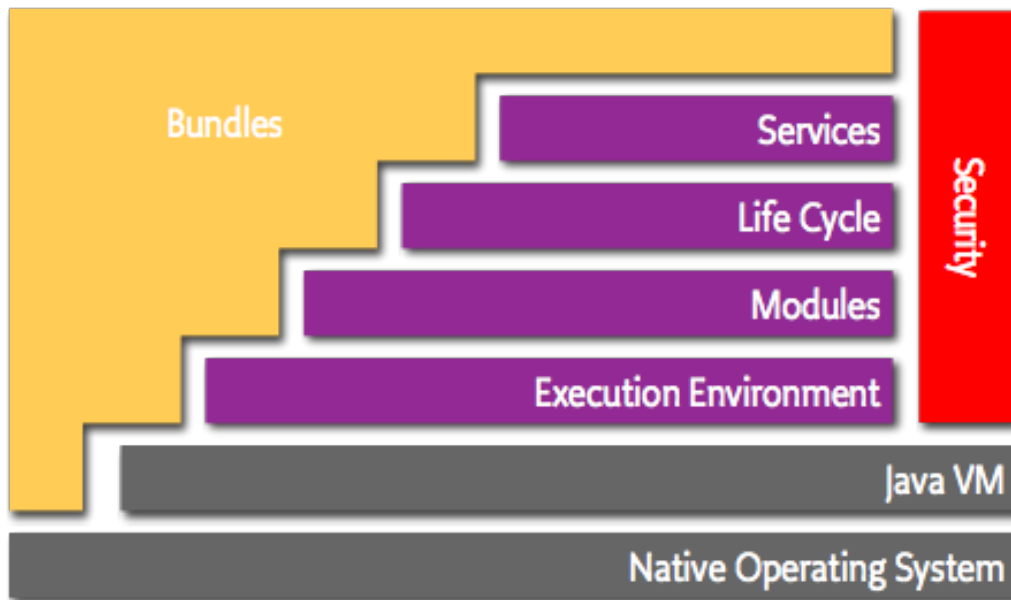


Figure 4.2: Layers of the OSGi framework (<https://www.osgi.org/wp-content/uploads/layering-osgi.png>, Feb 2018)

4.3 Other tools

4.3.1 BugZilla

BugZilla is a tool to organize all bugs within a project. A 'Defect Tracking System' such as BugZilla allows developers to track of the bugs and code changes. Besides by dint of BugZilla the communication between developers as well as the submission and review of patches can take place.

Within the SAP Sailing Analytics project, all bugs are tracked in BugZilla. This means that BugZilla has a great impact on the process of developing.

⁷see OSGi Alliance, *OSGi Alliance - Architecture*.

5 Realization

5.1 Implementation

In the following I explain the feature's implementation and the idea behind it. The figure (see 5.1) illustrate the relations represented by an UML diagram.

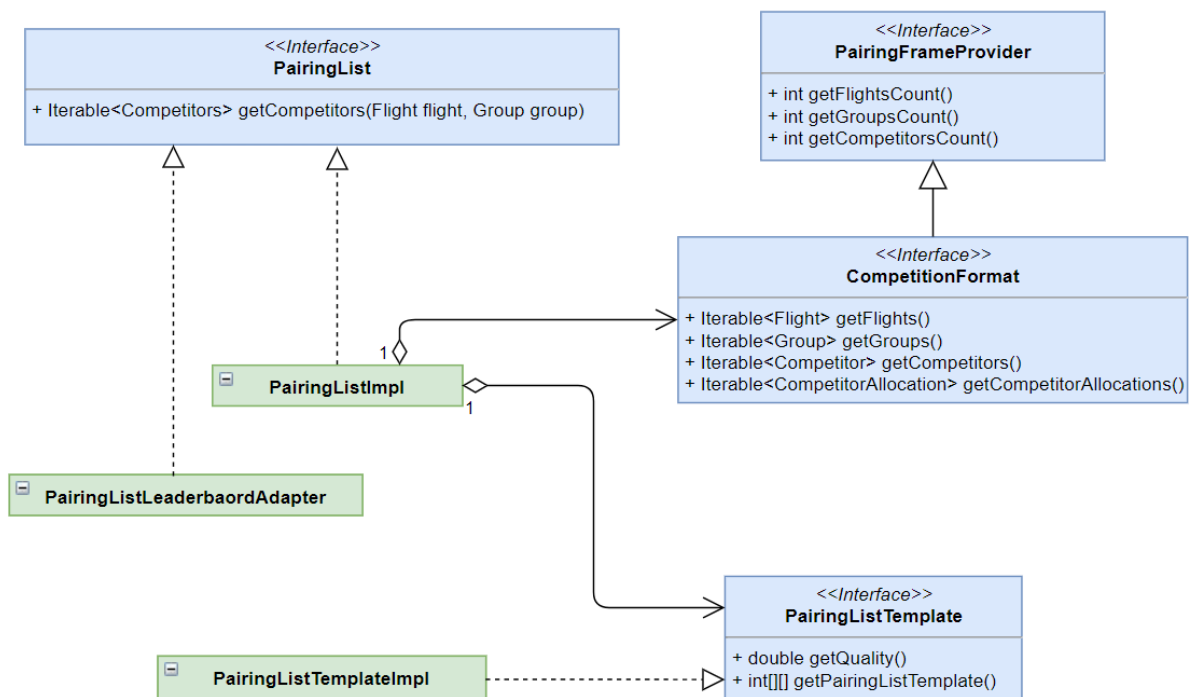


Figure 5.1: Implementation of the feature as UML (latest version)

5.1.1 PairingListTemplate

In the *PairingListTemplate* class, the whole concept and calculation of the algorithm takes place. The idea of first creating a template that contains only the number of each competitor came up since it turned out that pairing lists with the same parameters (such as number of flights, groups and competitors) ought not to be calculated twice. In turn this has the effect that out of a single template several pairing lists can be created.

Not to forget to mention that two pairing lists with the same competitors might be different from each other, because each time a pairing list is created out of a template, the competitors

within the pairing list are going to be assigned to another and randomly chosen competitor number. This leads to the fact that after shuffling the competitors, a competitor probably does not have the same number as before.

5.1.2 The actual pairing list implementation

The *PairingList* interface only has a single method called *getCompetitors()* that returns the specific competitor objects specified by the parameters flight and fleet by means of the corresponding *PairingListTemplate*. Since this API is kept abstract, this class uses generic types for the flights, groups and competitors. Later on the *CompetitorAllocations* will be added (see 5.1.5) and its corresponding features. The other implementation of the *PairingList* will be explained in chapter 5.5.2.

5.1.3 Template factory

The *TemplateFactory* is used to generate a pairing list. It follows the singleton approach which means that while runtime, only one instance of this object exists. This is necessary, because the factory stores pairing list with different combinations (i.e. the number of competitors, groups and flights).

By generating a new *PairingListTemplate* by means of the factory's method once, the *PairingListTemplate* is being saved into a *HashMap*. If the user wants to generate a new *PairingListTemplate* that has the same parameters, this method looks up in its *HashMap* and returns the respective *PairingListTemplate*.

Since it turned out that in the end the algorithm only takes a few seconds to calculate a pairing list, the functionality of storing pairing lists with the same combination, was removed. This has the advantage that if the user does not agree with a pairing list, it can easily be recalculated.

5.1.4 PairingFrameProvider and CompetitionFormat

The *PairingFrameProvider* and the *CompetitionFormat* are interfaces that provides the required parameters. By means of the *PairingFrameProvider*, only the numbers of flights, groups and competitors can be passed. However, the *CompetitionFormat* is used to fetch the specific instances of flights, groups and competitors. This interface uses generic types as well.

5.1.5 CompetitorAllocations

There is a bug within the project that deals with the problem that boats cannot exists independently of competitor. Since now this important for generating pairing lists, the bug's fix is

prepared. Forward-looking the interface to this bug has already been implemented. However, this functionality was added later. After finishing this project, this bug was solved.

Since the boats will be matched to the competitors when generating a pairing list, the *PairingList* interface has a generic type called *CompetitorAllocation*. The *PairingList* method *getCompetitors()* now returns a *Pair* of *Competitor* and *CompetitorAllocation* instead of a *Competitor*, so that the solution can build on this. The other interface is explained in chapter 5.5.2.

5.2 Development of the main algorithm

Within the following scenarios, the generation of a single flight is always in parallel to the MatLab algorithm. The following concepts introduce not only a heuristic, that ought to help creating pairing list with a good quality without using the Monte-Carlo-principle, but also other improvements like concurrency and reducing storage allocations.

5.2.1 Forming concepts of concurrency

After the first implementation of the origin MatLab algorithm in Java, due to improving the performance the idea of working with concurrency came up. Nowadays the usages of concurrency is becoming more and more popular, since almost every engine has multiple cores. This allows a real concurrency so that several tasks can run at once.

Processes and threads There are basic units within the concurrent programming: Processes and Threads.

All processes have a self-contained execution environment, which means that it has its own memory space. The meaning of processes is often equated with applications. However, an application can have several processes. Java applications in general only have one process and thereby for Java programming processes are not as important as threads.

Similar to processes, threads have its own memory as well. However, the creation of threads require fewer resources than the creation of processes. Furthermore processes can contain several threads, but at least one. This particular one is called main thread. Threads share the process' resources, which can lead to communication problems e.g. if two thread are about to access on the same memory space.¹

Handling of threads To avoid communication problems as already mentions, Java provides a class called *ThreadPoolExecutor*. It is used to organize the access on shared resources. In fact this means, it takes care that all incoming task will complete once the thread is ready

¹see Oracle, *Oracle Java Documentation - Processes and Threads*.

to go on. Tasks are represented by the so called interfaces *Runnable* and *Callable*. The difference is just that a *Callable* can return something.²

The idea was to put each iteration into a single task. After completing all tasks, the list of results is being compared to determine the best one. Each iteration does the same as in the MatLab algorithm. As expected the runtime of the algorithm was reduced by the number of cores minus 1 (this core is responsible for the all other tasks). So instead of 5, it took just about 1.5 minutes (using 4 logical cores).

Since in this case all tasks should return a pairing list, the problem of getting the results of each task simultaneously arises. The class *Future* solves this problem. By submitting a task, such *Future* object is returned and used to determine whether the respective task is already done or not. Its method *get()* waits for the result and returns it. Besides this method blocks as long as the result is not done.³

5.2.2 Prefixes

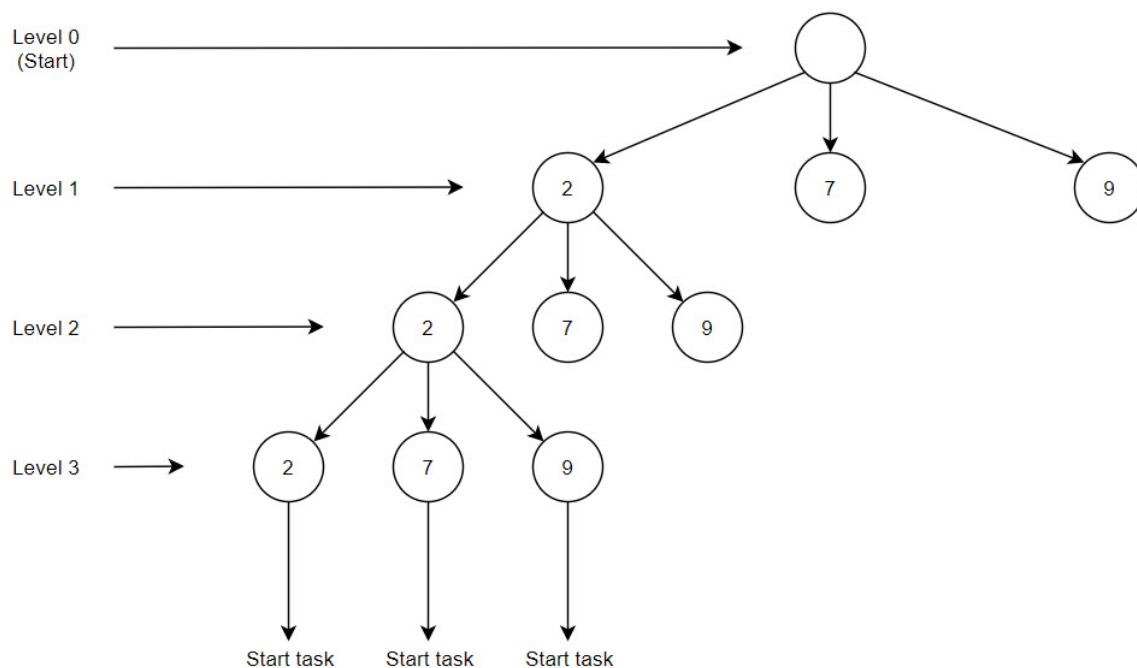


Figure 5.2: Prefix creation with backtracking-principle

To improve the performance again, the idea of generating a so called prefix came up. This works as follows:

²see Rheinwerk, *Rheinwerk Computing :: Java ist auch eine Insel – 14.4 Der Ausführer (Executor) kommt*.

³see Rheinwerk, *Rheinwerk Computing :: Java ist auch eine Insel – 14.4 Der Ausführer (Executor) kommt*.

First the algorithm creates so called prefixes (origin name: constant flights). The creation of all prefixes work like the backtracking-principle (see 5.2.2).

After starting this backtracking method, a set of seeds (see 2.2.2) is being created randomized. The size of this set defines the node within one level. Each node is responsible for generating a single flight with the specific seed (represented in the figure by the numbers in the nodes). This single flight is being passed to the next nodes so that these can append a new flight.

At a specific level (I will touch on this later), a prefix is completely created. From now on, the algorithm starts a new task, where the suffix is going to be calculated as before. These tasks can now run in parallel as already mentioned (see 5.2.1).

The following (see 5.1) shows the method implemented in Java. I will discuss the method calls *incrementAssociations()* and *decrementAssociations()* later (see 5.3.5). When calling the method, the parameters *currentPLT* and *futures* are empty and the *level*-parameter is 0.

```

1  private ArrayList<Future<int[][]>> createPrefix(int flights, int ↵
    ↳ groups, int competitors, int[][] associations,
2      int[][] currentPLT, int[] seeds, ArrayList<Future<int[][]>> ↵
    ↳ futures, int level) {
3  if (level < this.maxConstantFlights) {
4      // calculate Flights for current recurrence level
5      for (int seedIndex = 0; seedIndex < seeds.length; seedIndex++) {
6          int[][] temp = this.createFlight(groups, competitors, ↵
    ↳ associations, seeds[seedIndex]);
7          for (int m = 0; m < groups; m++) {
8              System.arraycopy(temp[m], 0, currentPLT[level * groups + ↵
    ↳ m], 0, competitors / groups);
9          }
10         level++;
11         associations = this.incrementAssociations(temp, associations);
12         this.createPrefix(flights, groups, competitors, ↵
    ↳ associations, currentPLT, this.generateSeeds(flights, ↵
    ↳ competitors),
13             futures, level);
14         level--;
15         associations = this.decrementAssociations(temp, associations);
16     }
17     // reset last recurrence step
18     for (int i = level * groups; i < level * groups + groups; i++) {
19         Arrays.fill(currentPLT[i], 0);
20     }
21 } else {
22     // Task start

```

```

23     Future<int[][]> future = executorService.submit((new ↵
        ↵ SuffixCreationTask(flights, groups, competitors, ↵
        ↵ currentPLT, associations, seeds.length)));
24     futures.add(future);
25     // reset last recurrence step
26     for (int i = maxConstantFlights * groups - 1; i >= ↵
        ↵ maxConstantFlights * groups - groups; i--) {
27         Arrays.fill(currentPLT[i], 0);
28     }
29 }
30 return futures;
31 }

```

Listing 5.1: *createPrefix()* method

Problems

Some problems occurred while defining the number of seeds or rather nodes per level and the max number of prefixes and its relation. For a specific case and combination, those approaches work. However, considering boundary values will break this approach, because either the required time is too high or the result is not acceptable.

5.2.3 Different suffixes

As already mentioned, after creating the prefixes the algorithm starts a new task and continues as before (see 2.2.2). Since thus there is still no systematic way, a concept was created that works very similar to the creation of the prefixes. Instead of randomly selecting seeds, the creation of the suffix follows the backtracking-principle. Each level is synonymous with a flight. However, the search space is also restricted. But against all expectations the result has not worsened.

5.2.4 A new concept

Even though this suffix-concept has improved the runtime, the same problem occurred as before (see 5.2.2). Hence a new concept was designed.

Instead of generating the seeds just before creating the flight, the algorithm creates an array that contains all seeds:

```
1 int[][] allSeeds = new int[iterations][flightCount];
```

Listing 5.2: allSeeds array

If there are more tasks than (logical) cores, then the runtime is even impaired. Though there is much concurrency, it exists just a little parallelism. This issue is fixed by this concept in the following way:

First the algorithm sorts the *allSeeds*-array. It has to be taken into account that within the seed combinations, the array is not sorted. Only the seed combinations as a whole are sorted. The *allSeeds*-array is split up as much as the number of available cores (the number of available cores is $n - 1$ and n being the number of logical cores). After dividing this array, each task starts with one part of the array.

From here on, each task creates pairing lists depending on the length of its part of the array. After creating a pairing list, it is compared to the best. If all tasks are finished, the best result of the respective tasks is returned.

5.2.5 Comparison of all concepts

After considering the development of the algorithm, all ideas have to be compared with each other. In the following I will summarize every single approach.

MatLab algorithm in parallel Although this approach is an improvement compared to the original MatLab algorithm, it still takes too long to calculate a pairing list. Besides the solution is still not a systematic one.

Systematic prefix This approach is in contrast to the previous approach partially solved in a systematic manner. However, some combinations are excluded, but this does not effect the result's quality. This approach does not handle boundary values though. Furthermore it is much more efficient.

Systematic prefix and suffix This systematic solution restricts the search space of possible combinations a lot and it is even more efficient than the previous approach. Though the problem of handling boundary values still exists.

Upfront calculated seeds This approach is solved systematic as well, but without restrictions concerning boundary values. Besides it is more efficient than all other approaches (regarding parallelism and concurrency).

5.3 Extensions and improvements of the algorithm

5.3.1 Distribution of competitors among the boats

One of the project's requirements was to enable the fairest possible distribution of competitors among the boats. Just as with the creation of a single flight, an association-matrix is created.

This association-matrix is very similar to that matrix, except that the competitors are applied over the boats. Afterwards, a decision is taken that is very similar to picking the next competitor in fleet only that this decision is based on a different matrix.

In concrete terms, this means that the matrix has as many rows as boats and as many columns as competitors. Each cell thus indicates how often a competitor has driven on a boat. Starting from this matrix, the competitors within the fleets are swapped. The decision as to which competitor will be placed in which position is similar to the decision as to which competitors are distributed among the fleets (see 2.2.2).

5.3.2 Reducing the boat changes

As it is mentioned in the project's requirements, in some cases the number of boat changes between each flight ought to be reduced. This leads to the fact that event managers have less effort carrying competitors to the boats, since flights do not take off shore.

It is now looked, how many matches the last group of a flight has with the groups of the following flight. Based on the number of matches, the fleet with the most matches is switched to the first position.

5.3.3 Empty spaces

It may occur that the number of competitors does not match with the number of boats (e.g. 16 competitors and 6 boats). For this case the pairing list has empty spaces. This problem has been solved in the following manner:

The algorithm determines the next possible competitor number (in this case 18 competitors) and remembers the number of so called dummies (in this case 2). After returning the pairing list, each competitor number higher than the original competitor number will be replaced with -1. This ensures that even though in some races not every boat is matched to a competitor, the pairing list is still well distributed corresponding to the quality factors (see 2.2.1).

5.3.4 Flight repetitions

Another customer request was to duplicate a pairing list. This is used to decrease the boat changes between the races. If e.g. there are 15 flights, 3 fleets, 6 boats and 18 competitors and the flight-multiplier or rather divider (the name was changes later) is set to 3, then the algorithm will calculate a pairing list with 5 flights and each flight is being repeated 3 times.

5.3.5 Incremental calculation of the association matrix

Previously, the association matrix was always calculated on the basis of the entire pairing list for each calculation. This has been improved by adding only the matrix of a single flights to an existing pairing list. In the context of creating the prefixes using the backtracking principle, this 'addition' could also be reversed.

5.3.6 More ideas or extensions

In the following I will outline more ideas that has been tried out but considered as not efficient. Another reason was that the priority was set low and thereby it has not been implemented.

Incremental calculation of the standard deviation The idea was to eliminate specific combinations if the quality is worse than the current best. In others words, the algorithm should consider whether the current pairing list is worth to go on calculating. However, it proved that the effort of this calculation is even greater than continuing the calculation of the pairing list.

Several quality factors One idea was to not only return the standard deviation of the association-matrix. Another value worth considering is the quality of the distribution of boats and competitors. This feature could be displayed in a diagram that visualizes both parameters.

Sorting the seeds Considering the last and current approach, an idea was to sort within the *allSeeds*-array the combinations for one pairing list individually. This has the advantage that e.g. $[0,0,1,2]$ and $[0,0,1,3]$ would have the same beginning and thus these flights would be calculated only once. However, this idea was not implemented due to lack of time. Furthermore it may be that sorting the combinations impair the time effort.

5.4 Integrating in the UI

5.4.1 Admin Console

The Admin Console is the part of the SAP Sailing Analytics where the main administration takes place. It is used to create leader boards, regattas, to match the competitors on the regattas, etc.

5.4.2 Interaction between back end and front end

The communication between server and client works with serialization (see 4.2.1). The data exchange works with so called data transfer objects (DTOs). So for many server side classes,

there is a respective client side DTO. Within this particular DTO, only the necessary data is stored.

Now with the help of the *SailingService* interface, which is a RPC service, the client can communicate with the server. In the following chapter I will go in detail about this.

5.4.3 Dialogs

GWT allows to create dialogs. The generation of pairing list takes place in such dialogs. The project SAP Sailing Analytics has its own dialog class called *DataEntryDialog*. It defines some standards every dialog in the Admin Console has, e.g. an OK- and Cancel-Button.

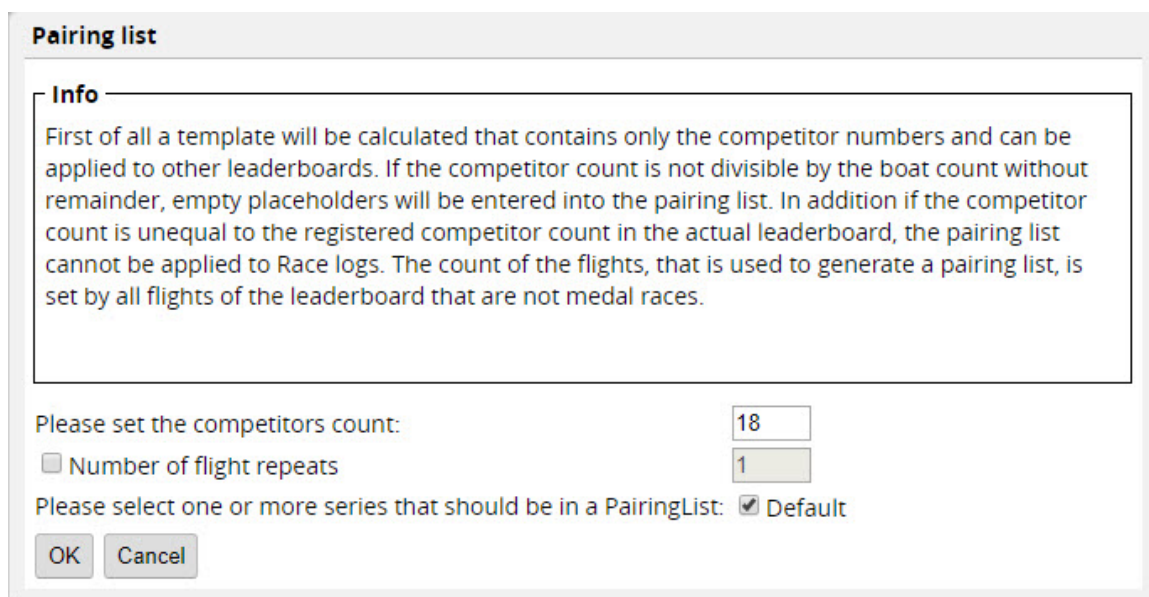


Figure 5.3: Setup Dialog

There are two dialogs: the first dialog (see 5.4.3) provides the ability to modify some settings such as the flight repetitions (see 5.3.4). Furthermore it provides options to modify the competitor number. If there are some competitors assigned to the current regatta, then this specific count is taken. Besides the user can select a series or several, that have the same parameters (flight and group count). By selecting a series, the number of flights and groups can be determined.

The second dialog (see 5.4.3) shows the calculated *PairingListTemplate* and more information such as the quality (see 2.2.1). Here it is possible to export this template to CSV (see 5.5.1), insert the pairing list into regatta (see 5.5.2), recalculate a new template based on the current parameters as well as showing the pairing list as a print preview.

Validator

To validate the user's input, the so called *Validator* interface is used. By entering any input, the *Validator* checks its correctness. If the input is not valid, the dialog's OK-button is disabled. This ensure, that no non-valid input reaches the back-end.

The first dialog (see 5.4.3) uses such a *Validator* to validate its input, e.g. the number of competitors must be greater than (or equal to) the number of groups.

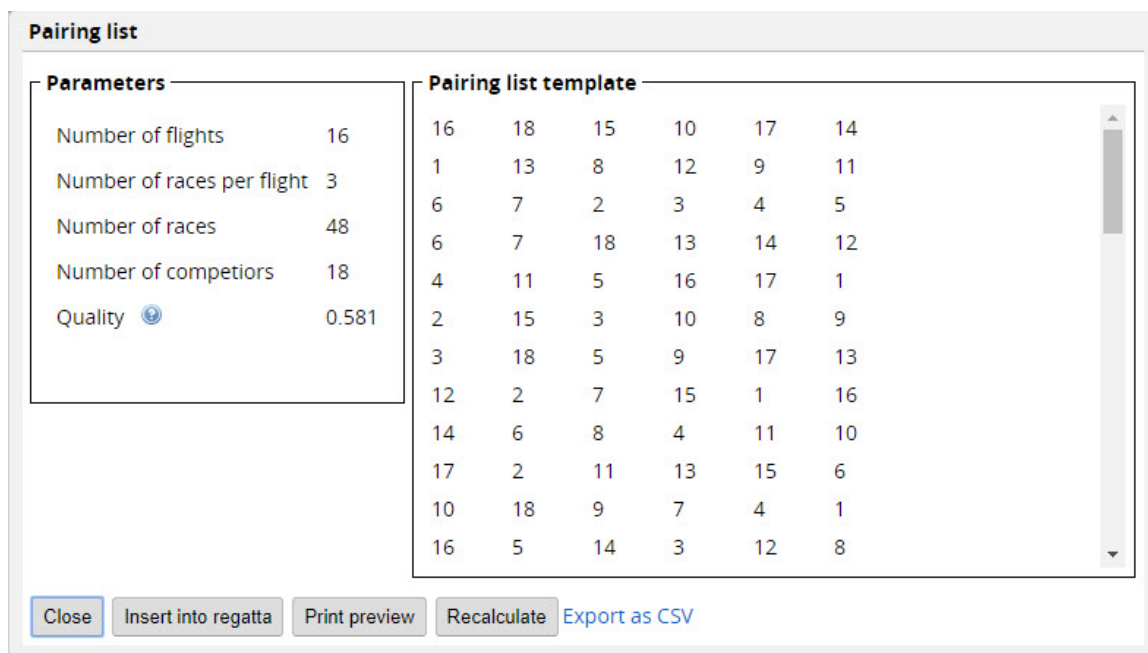


Figure 5.4: Dialog for the purpose of displaying the PairingListTemplate

5.5 Additional features

5.5.1 Exporting template to CSV

Since Excel-sheets are still in use of allocating the competitors to races, the user can export this pairing list template as a CSV file. Those can be used to import into Excel-sheets and CSV files are structured by means of commas (CSV stands for "comma seperated values"). By clicking the anchor on the setup dialog (see 5.4.3), a CSV file is downloaded.

The CSV file is created using a *StringBuilder*. The *StringBuilder* class is an utility class provided by Java (in the package *java.lang*) that helps putting strings together and performing other string operations.

5.5.2 Filling the race logs

The race logs define a log that logs all events handled on races. To give an example, the competitor registration will be logged. The aim was to register all competitors to the respective races that are defined by the pairing list. This has the advantage that the pairing list and its data can automatically be added into the Admin Console instead of doing this manually.

As already mentioned by pushing the button "Insert into regatta" on the dialog, a RPC is fired passing the *PairingListDTO* object to the client. Based on its data, the competitors are being registered on the respective race and its race log. In doing so, competitors that are already registered in this race log, will be revoked (but not deleted), unless they belong to this race.

PairingListLeaderboardAdapter

Since *bug2822* was not fixed yet, an interface on which the bug can build on has been created. As already mentioned, the *PairingList*'s method *getCompetitors* returns a *Pair* of a *Competitor* and a *CompetitorAllocation*. Therefore the *CompetitorAllocation* has to be registered in the race logs as well. This ensures that the competitors sail on the correct boat. For this purpose a new implementation of the *PairingList* interface ought to handle fetching the competitors from race logs called *PairingListLeaderboardAdapter* (see 5.1).

```

1  public class PairingListLeaderboardAdapter implements ↵
    ↳ PairingList<RaceColumn, Fleet, Competitor, Boat> {
2      @Override
3      public Iterable<Pair<Competitor, Boat>> getCompetitors(RaceColumn ↵
        ↳ raceColumn, Fleet fleet) {
4          List<Pair<Competitor, Boat>> result = new ArrayList<>();
5          int boatIndex = 0;
6          for (Competitor competitor : ↵
            ↳ raceColumn.getCompetitorsRegisteredInRacelog(fleet)) {
7              result.add(new Pair<Competitor, Boat>(competitor,
8                  new BoatImpl("Boat " + (boatIndex + 1), new ↵
                    ↳ BoatClassImpl("49er", true), "DE" + boatIndex)));
9              boatIndex++;
10         }
11         return result;
12     }
13 }
```

Listing 5.3: PairingListLeaderboardAdapter class

The interface to *bug2822* is already fixed (see ll. 9 in 5.3). Instead of creating new random boats, the method *getCompetitorsRegisteredInRacelog* will return a *Pair* of *Competitors* and its *Boats* after

fixing *bug2822*.

This class finds use in the print functionality since the print view is generated out of the information of the race log.

5.5.3 Print functionality

There are two functionalities to differentiate. First the print preview functionality and second the origin print functionality.

The print preview allows the user to display the pairing list that has just been created using the dialog. By pushing the respective button (see 5.4.3) a new dialog opens that shows the print preview. The source of the pairing list is the dialog where the template is shown.

The origin print functionality is used to print the actual pairing list that is registered in the Race logs. By means of the *PairingListLeaderboardAdapter* (see 5.5.2) the information will be fetched and displayed. The print view is a new *EntryPoint*. The user reaches the *EntryPoint* by clicking the print icon.

EntryPoint In GWT the *EntryPoint* class is a module of GWT. The *EntryPoint*'s method *onModuleLoad()* contains the code that will be executed when launching the application. In other words, each *EntryPoint* is synonymous with a web application.⁴ To give an example, the Admin Console is an *EntryPoint*.

⁴see Google, [GWT] *Documentation - Client*.

6 Conclusion and outlook

6.1 Conclusion

All in all, all objectives and requirements of the project were met. The user can easily create a pairing list via the UI within a acceptable time frame. Besides there are several export functionalities (print view and CSV export) and registering the pairing list into race logs was simplified. Furthermore each pairing list has an acceptable quality concerning all quality factors. The implementation of the whole project can be applied to other sports.

6.2 Outlook

In this work I have mentioned more ideas that would make this feature even more efficient or easier to use. Still there is a great potential for extensions. Besides there are some interface where bug2822 can build on it, after fixing this bug. This would make this project complete, since some functions do not work yet, e.g. the print functionality.

Bibliography

Internet references

Eclipse Foundation. *Eclipse - Desktop & web IDE*. URL: <http://www.eclipse.org/ide/> (visited on 02/02/2018).

– *Hudson - Meet Hudson*. URL: http://wiki.eclipse.org/Hudson-ci/Using_Hudson/Meet_Hudson (visited on 02/02/2018).

Git and Software Freedom Conservancy. *Git - About Version Control*. URL: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> (visited on 01/28/2018).

Google. *Google Web Toolkit Documentation - Ajax Communication: Introduction*. URL: <http://www.gwtproject.org/doc/latest/tutorial/clientserver.html> (visited on 01/10/2018).

– *[GWT] Documentation - Client*. URL: <http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsClient.html> (visited on 01/28/2018).

– *[GWT] Documentation - Client*. URL: <http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsClient.html> (visited on 01/28/2018).

JUnit. *JUnit 5 - User Guide*. URL: <http://junit.org/junit5/docs/current/user-guide/> (visited on 02/02/2018).

Oracle. *Oracle Java Documentation - Processes and Threads*. URL: <https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html> (visited on 01/15/2018).

– *Oracle Java Documentation - Serializable Objects*. URL: <https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html> (visited on 01/10/2018).

OSGi Alliance. *OSGi Alliance - Architecture*. URL: <https://www.osgi.org/developer/architecture/> (visited on 01/15/2018).

Rheinwerk. *Rheinwerk Computing :: Java ist auch eine Insel – 14.4 Der Ausführer (Executor) kommt*. URL: http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_14_004.htm (visited on 01/21/2018).

ThoughtWorks. *Selenium - Web Browser Automation*. URL: <http://www.seleniumhq.org/> (visited on 02/02/2018).