UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN
INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MASTER

DESIGN AND DEVELOPMENT OF A SCHEDULING INGELLIGENT SYSTEM FOR YOUTH CAMPS ASSOCIATIONS BASED ON CONSTRAINT PROGRAMMING

DAVID PÉREZ SANZ

2021

TRABAJO DE FIN DE MASTER

Título:	DISEÑO Y DESARROLLO DE UN SISTEMA INTELIGENTE DE PLANIFICACIÓN PARA ASOCIACIONES JUVENILES BASADO EN PROGRAMACIÓN CON RESTRICCIONES
Título (inglés):	DESIGN AND DEVELOPMENT OF A SCHEDULING INGELLIGENT SYSTEM FOR YOUTH CAMPS ASSOCIATIONS BASED ON CONSTRAINT PROGRAMMING
Autor:	DAVID PÉREZ SANZ
Tutor:	CARLOS ÁNGEL IGLESIAS FERNANDEZ
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO DE FIN DE MASTER

DESIGN AND DEVELOPMENT OF A SCHEDULING INGELLIGENT SYSTEM FOR YOUTH CAMPS ASSOCIATIONS BASED ON CONSTRAINT PROGRAMMING

FEBRERO 2021

Resumen

Como es bien conocido, la vida es un conjunto de problemas a los que la sociedad debe enfrentarse continuamente para la consecución de unos objetivos propuestos. Estos problemas están presentes en todos los contextos del día a día: académico, profesional o incluso en la vida familiar. Existe un tipo concreto de problemas conocido como los problemas de planificación, cuyo principal objetivo es asignar unos recursos a otros de forma ordenada y cumpliendo con algunas restricciones predefinidas. Este tipo de problemas es bastante común; por ejemplo, en la creación de grupos de trabajo en empresas o para la elaboración de turnos. Las planificaciones elaboradas deben respetar algunos requisitos como las horas de trabajo, tiempos de descanso, vacaciones, convenios de los trabajadores, preferencias, etc. Algunos ejemplos son: los turnos de enfermeras en hospitales, la distribución de profesores y aulas en centros educativos, los turnos de personal de servicios 24x7, la asignación de tareas en proyectos, el horario de conferencias de congresos o la planificación de exámenes.

También existe este tipo de problemas en las asociaciones juveniles a la hora de definir los grupos de trabajo cuando comienza el año. Su elaboración es muy compleja debido a todos los requisitos que deben cumplir, tanto en la estructura como los propios del personal involucrado, y se debe dedicar mucho tiempo para conseguir una solución adecuada. Mediante este proyecto se proporciona a los grupos scout una aplicación web capaz de generar planificaciones, en pocos minutos, una vez se introducen los requisitos del personal y de los grupos de trabajo, sin exigir ningún esfuerzo humano adicional.

El desarrollo de la aplicación web se ha realizado en varias fases. En primer lugar, se han estudiado los detalles de la problemática para conocer el contexto y las características del problema a resolver. Esta información permite realizar el análisis de requesitos para extraer qué funcionalidades concretas debe disponer la aplicación a desarrollar. El resultado del análisis de requesitos proporciona la información necesaria para elaborar la arquitectura del sistema. Finalmente, se detalla un ejemplo introduciendo datos sintéticos en el sistema, similares a los datos reales que la aplicación recibiría, y se analiza la solución proporcionada.

Palabras clave: Programación basada en restricciones, Grupo scout, Planificación, OptaPlanner, Spring, MySQL.

Abstract

As all we know, life is a set of problems that society must face continuously to achieve the goals proposed. Problems appear in all contexts of life; academic, professional, or even in family life. One type of these problems is scheduling and planning problems. The objective here is to assign some resources to others in an orderly manner while complying with some restrictions predefined. The presence of this type of problem is common in society, for example, for creating workgroups in businesses or for work shifts definition. The planning should respect some requirements such as working hours, rest time, vacations, workers' conventions, preferences, etc. This context includes subjects like nurse rostering in hospitals, schools teachers and classrooms planning, 24x7 services personnel coverage, cloud balancing systems, project task assigning, congress conference schedule, or exams planning.

This problem is present in youth associations when they have to define a planning with the workgroups at the beginning of the year. The preparation of a planning gets complex with all the considerations that workgroups and personnel request, and people should spend a long time to obtain a suitable solution. The project provides a web application capable of generating valid plannings for scout associations in some minutes by introducing the data of the personnel and workgroups requirements, and without any additional human effort.

The process of developing the application requires some steps to elaborate the accurate solution to the problem. The project exposes the problematic to know the context and the exact problem to solve as the first step. This information allows us to perform the requirement analysis to extract the characteristics of the application to develop. The requirement analysis provides the necessary information to elaborate the architecture of the system. Finally, the thesis includes an example where the system receives synthetic data similar to the real data that the application would receive to analyze the solution that it provides.

Keywords: Constraint Programming, Scout association, Planning, Scheduling, OptaPlanner, Spring, MySQL.

Agradecimientos

En primer lugar, me gustaría agradecer a Carlos el esfuerzo, la ayuda y los consejos proporcionados para la elaboración del proyecto.

En segundo lugar, quería reflejar en esta memoria unas pocas palabras sobre la Escuela Técnica Superior de Ingenieros de Telecomunicación que durante todos estos años me ha brindado los conocimientos necesarios para poder dedicarme hoy en día a una profesión que me encanta. Han sido años que han requerido mucho esfuerzo y dedicación, pero sin ellos, estoy seguro de que no sería la persona que soy ahora.

Por otro lado, me gustaría dar las gracias a Sofía por todo el apoyo y la energía que me ha transmitido durante todo este tiempo. Todo sería más difícil sin tu ayuda.

No querría finalizar esta memoria sin agradecer a mis padres todo lo que han hecho para que sea la persona en la que me he convertido.

Contents

List of Figures

CHAPTER 1

Introduction

This chapter introduces the context of the project, including a brief overview of all the different parts that will be discussed in the project. It introduces a series of objectives to be carried out during the realization of the project. Moreover, it will introduce the structure of the document with an overview of each chapter.

1.1 Context

The world contains so many youth camp associations with diverse themes, such as multi-adventure camps, religious associations, or even the well-known scouts[1]. These associations' main objectives are to teach children social and ethical values in a crosscutting way to traditional schools.

Many young people of all ages enroll in scout associations, and also associations have people in charge of these children. These people's primary functions are taking care of children and educating them in the scouting knowledge they should have at their age. The personnel in charge of carrying these tasks are the scout leaders, Scouters, or *monitores*[2].

Depending on the age, the association distributes children into small groups of pupils of a range of three years apart, called *sections*. These small groups' size determines whether to split them into smaller groups to facilitate tasks to scout leaders and provide personalized attention to their children. The name of these smaller groups is *units*.

When I was eight years old, I became part of one of these scout associations. I had participated in almost all of the activities until adulthood, when I started collaborating as a scout leader. As scout associations are NGOs, the work that scout leaders perform is entirely volunteer. One of the main problems identified in this scope, while being scout leader, was the difficulties while composing scout leader teams in charge of each unit. Commonly, this task takes hours and even days if done manually, but using the technology present nowadays, it can be automated and performed effortlessly in several minutes.

This thesis's main subject is the development of an intelligent system allowing the automation of distributing the set of scout leaders into the units of a scout association. Moreover, the project covers the development of a web application to enable the use of the system for non-technological people, accessing anywhere and anytime.

[1] People usually call Boy Scouts wrongly to these associations, but the name changed progressively worldwide when girls became part of this groups. The only country that maintains this name in the USA, where they receive the name of Boy Scouts of America, although there are other groups named Girl Guides, just for girls.

[2] Spanish term

1.2 Project goals

The following general objectives have been defined within this Master Thesis.

- To acquire more in-depth knowledge of intelligent systems based on constraint programming.
- To obtain greater ability in problem description and requirements gathering.
- To apply functional programming techniques for the development of efficient programs working with data flows.
- To broaden knowledge in web applications deployment for their main components: client and server.

1.3 Structure of this document

The remaining of this document is structured as follows:

Chapter 2 covers the main essential concepts used to develop the project's solution. In particular, the chapter includes subjects as constraint programming, functional programming, and web application development.

Chapter 3 explains the problematic in detail that the application solves. The chapter includes the problem characteristics, the details of scout associations, the properties that a valid planning should have, and an example of the problem.

Chapter 4 contains the requirement analysis with all the use cases and actors. After performing the analysis, the chapter details the functional and non-functional requirements.

Chapter 5 explains the architecture of the application and the implementation using the technologies presented in ***Chapter 2***. The chapter explains the architecture of the application's three main components and its subcomponents: the intelligent system, the web server, and the persistence system.

Chapter 6 contains a case study of the application developed and an analysis of the results obtained. This chapter details a problem, the application configuration, and the analysis of the results generated with the application.

Chapter 7 gathers the conclusions of the whole thesis as a result of the work performed. The chapter covers the achieved goals, conclusions, and future work.

CHAPTER 2

Enabling Technolgies

This chapter offers a brief review of the main concepts and technologies that have made possible this project. The chapter centers in three sections: Constraint programming, Functional programming and Web applications.

2.1 Constraint programming

2.1.1 Definition

[12] provides an accurate definition of what the term constraint programming refers to.

> *Constraint programming*[1] *is a paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, and operations research.*

To do so, users define constraints to achieve a feasible solution with the variables involved. The program does not need to specify the steps to solve the problem [22], as in imperative programming.

In this context, problems contain a set of variables, whose possible values belong to a domain, and a set of constraints [25]. In other words, a tuple (X, D, C) being:

- $X = x_1, x_2, ..., x_n$ the set of **variables**.
- $D = d_1, d_2, ..., d_n$ the set of **domains**, where d_i is a finite set of potential values for x_i.
- $C = c_1, c_2, ..., c_m$ the set of **constraints**.

Constraint programming would probably not obtain the optimal solution to a problem because its design does not contemplate to optimize functions. Instead, the main objective is to find feasible solutions complying with the constraints determined. This behavior is because constraint programming focuses on the constraints and variables rather than the objective function. Moreover, a constraint programming problem may not have an objective function [12].

A simple example [25], using mathematics and following the previous notation, could be:

$$x, y, z \in \{0, 1\}, x_1 + x_2 = x_3$$

[1] In this case, the word *programming* refers to the arrangement of a plan and the reader should not misunderstand it as the concept of programming in a computer language.

Where:

- x, y, z are variables.
- $d_x = d_y = d_z = \{0, 1\}$ are domains.
- $x + y = z$ is the unique constraint in this problem.

A pair (S, R) defines a constraint C, where:

- $S = (x_{i_1}, x_{i_2}, ..., x_{i_k})$ are the variables of C, and known as **scope**.
- $R \subseteq d_{i_1}, d_{i_2}, ..., d_{i_k}$ are the tuples satisfying C, and meaning the **relation**.

So in the previous problem, another expression of the constraint $x_1 + x_2 = x_3$ could be:

$$((x_1, x_2, x_3), (0, 0, 0), (1, 0, 1), (0, 1, 1))$$

A tuple $\tau \in d_{i_1} \times ... \times d_{i_k}$ satisfies C iff $\tau \in R$.

The extensional representation remains in the arity of a constraint [25], being the size of its scope. So depending on its arity, a constraint could be:

- Arity 1: unary constraint
- Arity 2: binary constraint
- Arity n: n-ary constraint

However, the description of constraints is usually compact and uses an intensional representation [25]. This means that a function determines a constraint with scope S:

$$\prod_{x_i \in S} d_i \to \{true, false\}$$

So the satisfying tuples are precisely those whose result to the constraint function is true. With this notation, expressing constraints becomes more manageable and provides a useful way to implement them.

A solution for a constraint programming problem is an assignment of values to variables $(x_1 \mapsto v_1, ..., x_n \mapsto v_n)$ such complying these requirements:

- Domains are respected: $v_i \in d_i$
- The assignment satisfies all constraints in C

Nevertheless, we consider a constraint programming problem solved when we find a solution even if it is not the best (*Constraint Optimization Problem*) [25]. To solve a constraint programming problem, we do not need to find all the possible solutions. This concept is essential at the time of using this technique to solve problems. If the solution should be optimal, we should consider other alternatives depending on the solution's requirements.

2.1.2 Categories of problems

Constraint programming is an extremely successful technique for reasoning about assignment problems. This relies on its formulation; values from a domain assigned to decision variables, following defined restrictions. This technique has two assumptions on these type of problems [12]:

- The definition of the problem is completely certain, not leaving any room for doubt.
- Problems are static, and the requirements do not change during the finding of a solution.

Not all problems found based on assignment comply with the two assumptions because the world is dynamic. For example, if we thought of assigning tasks in an office, we can prepare an approximation of tasks in advance for a project. However, things happen, and personnel's computers might break, the material could be delivered late, or employees may become in a disease. However, in many cases, an approximated deterministic and static model may be enough to define the problem going alone with the previous premises. Users treat the dynamism by reformulating the problem continuously, to adapt solutions for changes, or even defining a more complex problem with more restrictions according to the possible issues that might arise to avoid having to plan new solutions.

In other words, problems could be one of the following broad categories when talking about real-world problems[12]:

- Problems where a solution is enough, known as **uncertain problems**.
- **Dynamically changing problems** that require multiple solutions. This category contains three types, depending on the actions the solver should take to comply with the problem requirements:

- The solver **reacts while the problems change**.
- The solving process **records information about the problem structure**, and uses it during the reaction phase.
- The solver **searches pro-actively for solutions** that anticipate the expected changes.

Uncertain problems do not include a complete description of the problem, but the solving process can produce a first initial unchangeable solution. The imprecision in the problem description leads the progress of developing accurate solutions. Depending on the imprecision, uncertain problems consider three important cases; the problem is intrinsically imprecise (***fuzzy problems***), the problem **has more than one realizations**, or the problem has **probability distributions** over the full realizations.

In **fuzzy problems**, they capture the imprecision by partially satisfying constraints. Constraints receive a value of a range within completely satisfied or completely unsatisfied. For example, when talking about the prize of a configuration that must be *cheap* and a fuzzy membership function defines this behavior. This technique characterizes the extent of constraints' satisfaction, defining concepts as *fairly cheap* or *relatively expensive*. So the problem has a rank of different solutions classified to search for the optimal ones.

For **problems with multiple possible realizations**, firstly, the solving process needs to identify the facts of the incomplete description. This may be because the complete set of variables X is unknown or the problem does no fully specifies the domains D or constraints C. This case needs to reformulate the definition explicitly specifying the constraints, so domains would be unary constraints using values from a universal set, and the constraint set defines the variables implicitly. So the constraint set contains all the uncertainty.

Problems based on probability include two different formalisms. One involves uncertainty over the problem constraints so that each one has associated probability. The goal here is to find the solution with the highest probability, assuming that the solution will not have an assignment with a probability of 1. The other assigns a probability to the uncontrollable parameters, and the objective is to make a decision with the maximal probability of being a solution to the full problem.

Problems that change over time, known as **dynamically changing problems**, should respond to alterations caused by a user, an external agent, or the environment. This usually happens during the finding of a solution. A dynamic problem is a chain of problems where an external agent adds and removes constraints. The solving process should solve the problems of the chain sequentially to obtain the overall solution.

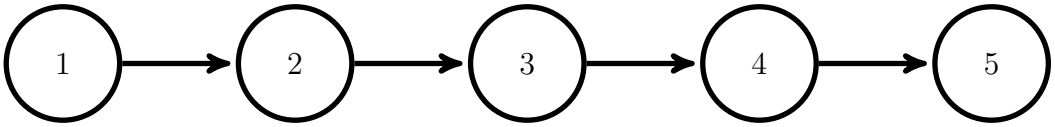

Figure 2.1: Example of dynamically chaging problems chain

In some cases, we do not know how the problem will change in the future. For this cases, **the solver should react while the problem changes**. To be efficient, solvers should explore the past history of problems and solutions to have a reference for providing the new solution. Some solving methods keep all the assignments from the previous solution as an starting point, then progressively modify them to obtain a compatible solution to the actual problem. This methods receive the name of *Local Repair methods*.

The **solver could also gather information while finding solutions**, having prepared in advance for future changes. This behavior is useful because of the premise that the following problem's structure has similarities with the previous ones. For each problem in a dynamic sequence, the solver stores the path taken to the solution to perform future searches during resolutions. This information is useful to search and prune the solution space based on previously taken decisions.

Finally, some problems could be recurrent; this means that some issues could repeatedly happen in time. For example, the occasional temporary loss of a resource due to reliability problems. To produce robust solutions, the solver should be able to acquire and process this kind of information. The solver could improve the initial solutions by **reasoning about likely changes** and the ones provided may be likely to remain or modified at a little cost. The solver should monitor changes while generating solutions to elaborate the distribution of probability. The solver would penalize then solutions that use frequently lost values.

2.1.3 Constraint solving

The solving process of constraint programming problems can use multiple techniques:

- **Generate and test:** Brute-force method consisting of generating all possible candidate solutions [25]. The set of solutions contain a solution per each combination of values from the domains of the variables. Then, the solver tests each solution to discover the compliance with the restrictions. This method is extremely inefficient, so solvers do not use it widely.
- **Backtracking search:** Algorithm that incrementally builds candidates to the so-

lutions and discards candidates as soon as they could not form a valid solution [66] [25].

- **Local search:** Method which iteratively improves a solution, based on their variables' assignments until satisfying all constraints [66] [25]. Every step, the solver modifies the value of a variable so that the solution is close to the previous one in the assignment space.
- **Dynamic programming:** Breaks the problem down into simpler sub-problems recursively [66]. The solver solves problems optimally finding solutions to the sub-problems until reaching the optimal structure. Some problems can not use this method globally (*decision problems*), but the solver can split them in some points to solve them partially with this manner. The method combines mathematical optimization procedure and a computer programing one.

The exploration of solutions is a searching tree where the root corresponds to the variables' empty assignments, and each branch assigns a different value to a variable. Each node contains as many children as values in the domain of the variable concerned. The generate and test method needs to visit each of the leaves until finding a solution, so computational complexity becomes elevated [25].

$$O(m^n \cdot e \cdot r)$$

Where

- n is the number of variables.
- m is the size of the largest domain
- e is the number of constraints
- r is the largest arity

Backtracking search performs a depth-first transversal, which in the worst case has the same complexity as generate and test method, but in practice works bette [25]r.

2.1.4 Example: N-Queens

N-Queens problem [25] is one of the most common examples of constraint solving problems. The problem consists of: given $n \geq 4$, set n queens on an $n \times n$ chessboard so that they do

not interfere with each other in terms of attack. The chessboard must have one queen per row so that two of them do not match in the same column or diagonal.

To solve this problem, the generate and test method is very inefficient, so firstly, we will solve the problem using the backtracking search method [25]. To do so, the algorithm maintains partial assignments consistent with the constraints. This means the algorithm explores leaves and their children while considering valid the assignments. When the solver runs into a non-complying solution, it undoes the last assignment and stops exploring that leave and its children. The steps will be [25]:

1. Initial stage will maintain empty assignment of the variables.
2. Every step, assign a value to a variable in its domain.
3. When the solving process detects a partial assignment that can not extend to a solution, do *backtracking*: undo last decision and stop exploring that branch.

With local search with constraint propagation [25], the strategy is similar to the previous one but with specific differences. With this method, the search tree receives a previous prune removing values from the domains to obtain a valid solution. The main advantage is the smaller search tree compared to previous methods, but the time to spend on each node is higher. Constraint propagation considers some types of propagation with different trade-offs between pruning power and costs in time.

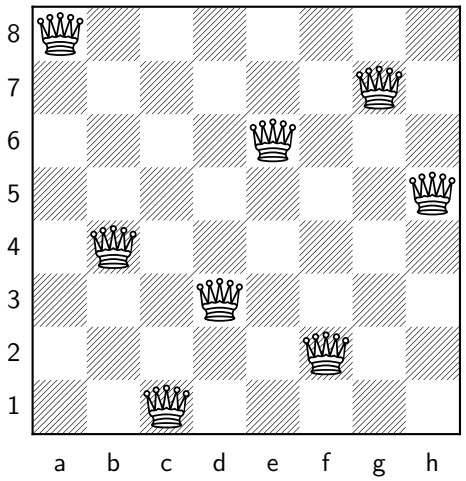

Figure 2.2: N-Queens problem with $n = 8$, possible solution

Finally, all algorithms should converge to a valid solution like the one shown in Figure 2.2. The conclusions related to the solution are that the chessboard should have one queen

per row, one per column, one per diagonal, and the maximum number of queens that a chessboard of $n \cdot n$ can accommodate is n.

This problem is a kind of conceptual challenge, so it satisfies the two premises about constraint solving problems. The problem definition does not contain uncertainty, and the requirements do not change during the solution's execution. The solving process does not need to consider approaches or look for complex alternatives to find a feasible solution, like when treating with dynamic or uncertain problems.

2.1.5 Optaplanner

OptaPlanner [16] is an open-source, lightweight, and embeddable AI constraint solver design to optimize planning and scheduling problems. It abstracts mathematical equations to code, applying constraints on plain domain objects and combining sophisticated Artificial Intelligence optimization algorithms with very efficient score calculation.

For OptaPlanner, a planning problem has an optimal goal, based on limited resources and under specific constraints [15]. Optimal goals could be to maximize or minimize some variables, or even both of them. While some variables need to be as maximum as possible, others need to have minimum values, such as buying the highest quality products and minimizing the total budget of buyouts. To reach these goals, programs should consider the available resources such as the number of people, the amount of time, physical assets, or even total budget.

Figure 2.3: OptaPlanner Planning Problem Definition
source by:
`https://docs.optaplanner.org/7.45.0.Final/optaplanner-docs/html_single/`

OptaPlanner framework solves constraint satisfaction problems efficiently, combining optimization heuristics and metaheuristics with very efficient score calculation. Basic techniques such as brute force algorithms will take too long to solve a problem because of the wide range of combinations of the problem's variables. Furthermore, a quick algorithm will return a non-optimal solution and may not comply with the required constraints. OptaPlanner deals with these problems, establishing a trade-off between them and providing a right solution in a reasonable time.

2.1.5.1 Types of constraints

Optaplanner defines a basic approach depending on the importance of the constraints that uses two levels [15]. The framework initially proposes two types of constraints, that a user can customize for a determined program:

- **Hard constraints:** A solution must fulfill all of them to consider the solution feasible. An example of them in an employee rostering problem could be to respect an eight-hour working day.
- **Soft constraints:** A solution should fulfill them, and the algorithm should take them into account, but they do not determine the feasibility of the solution. An example of a soft constraint following the previous problem could be the preference of an employee of working in a morning shift instead of a night shift.

Some problems could define only hard constraints. Others may need to define both of them, and other problems would require higher-level implementations to reach a feasible solution due to its complexity.

2.1.5.2 Problem solutions

With variables, their domains, and constraints, the search space becomes typically massive [15]. Inside, a problem can have some types of candidate solutions:

- **Possible solution:** OptaPlanner documentation defines this category for all candidates contained in the space, whether they comply with the constraints or not.
- **Feasible solution:** Refers to solutions not breaking hard constraints, but they may not comply with all the soft constraints or even none of them.

- **Optimal solution:** Solution with the highest score. At least, a problem has one optimal solution, but it can have more. If a problem does not have feasible solutions, the optimal solution would not be feasible too.

- **Best solution found:** Solution with the highest score found in a given amount of time. The main characteristic is that likely it would be feasible and, with enough time, it might be an optimal solution.

Usually, a problem has a massive amount of possible solutions. To find an optimal solution for an implementation, a program needs to evaluate a subset of possible solutions. The algorithm's election could be difficult in advance, as the programmer does not know the problem's complexity on most occasions. OptaPlanner allows users to easily choose the optimization algorithm by typing a few code lines so that people can test to find a suitable algorithm for each problem.

2.1.5.3 Solver

The solver is the component in charge of finding a solution to the problem [16]. This element test combinations and evaluates their performance until getting the combination with the best score.

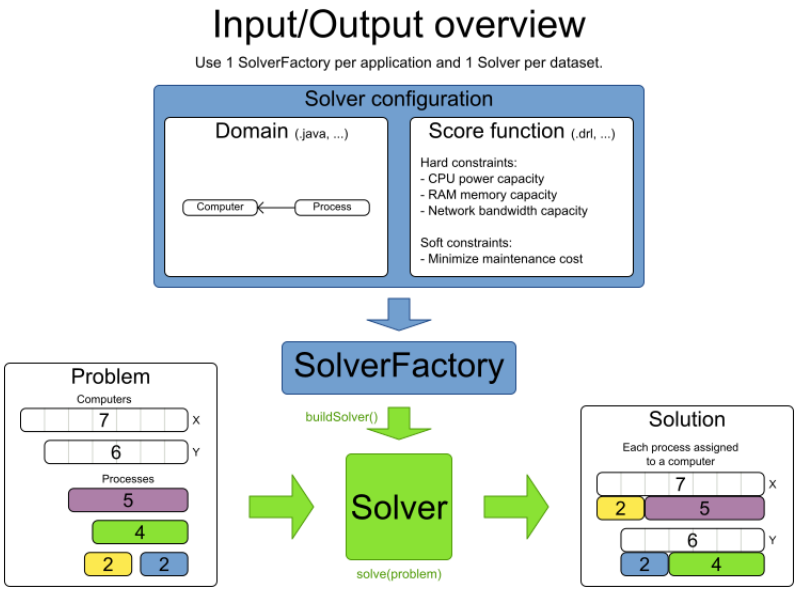

Figure 2.4: OptaPlanner Solver diagram
source by: `https://www.optaplanner.org/`

The solver receives all the components of a problem: the variables, the domains of their values and the constraints. The user defines the score function that assigns a score to the set of combinations from a solution depending on the constraints that this solution meets. To generate a solver with a specific configuration, OptaPlanner provides a class called *SolverFactory* that returns a solver configured and ready to solve a certain type of problems.

2.2 Functional programming

2.2.1 Definition

Functional programming [2] has become really trendy these days in many languages because of the broad advantages that it can provide on the developed software. Many companies use this technique to develop some components in their applications, such as Facebook, Yahoo, or Amazon [7].

A proper definition of what this concept consists of could be the process of building software by composing pure functions, avoiding shared state, mutable data, and side effects. Functional programming is declarative, rather than other standard programming techniques considered imperative, and application states flow through pure functions.

Functional code is more concise, predictable, and more manageable to test than imperative or object-oriented one. Functional programming is a programming paradigm that changes the traditional way of conceiving software development, basing it on some fundamental principles.

2.2.2 Concepts

As seen before, functional programing has some important concepts that this section exposes.

2.2.2.1 Pure functions

Pure functions refer to functions that, given the same inputs, always return the same outputs without side effects [7]. This means that this function will always perform the same actions independently of the program's variables' values.

Another essential property of these functions, called *referential transparency*, consists in

the equality of substituting a function call with its resulting value without maintaining the meaning of the problem.

Using pure functions has some advantages [1] as:

- Improved readability and maintainability because a function can perform some tasks over given arguments, instead of searching in global variables or states.
- Easier incremental development because as functions perform actions in a closed and reduced scope, the code is easier to refactor and changes to design.
- Effortless for testing and debugging, as functions could be considered small components and tested separately.

2.2.2.2 Function composition

The combination of functions produce a new function [7]. This concept is similar to math function composition, where if there are two functions $f(x) = 3x + 2$ and $g(x) = x^2 + x + 1$ the composition $(g \circ f)(x)$ will be:

$$h(x) = (g \cdot f)(x) = (3x + 2)^2 + (3x + 2) + 1 = 9x^2 + 15x + 7$$

So solving $h(2)$ will produce the same result as solving $f(2)$ and then $g(f(2))$ which in both cases will be 73. Equally occurs with computational function combining, so that the developer could split complex functions into simpler ones and call them when needed.

2.2.2.3 Shared state

If an item, such as a variable, object, or memory space, exists in a shared scope or as the property of an object shared across scopes [7]. These scopes can be object scope, shared variables between objects, program scope, or even shared state between multiple software.

The main drawback with shared states is traceability because the debugging process requires understanding the shared variable's complete history to know the reasons for the effects of a function. In a program with a shared state between components executed in different processes, threads, or even applications executing in different machines, race conditions can appear and cause undesirable issues.

Another critical problem with the shared state is the relation between the order in which the program calls functions and the final result. Changing this order can cause a cascade of

failures because functions acting with a shared state are also timing-dependent. Avoiding shared state and using pure functions, timing, and order of function calls do not change the functions' result. Moreover, function calls are entirely independent of other function calls, so changing and refactoring the program is simpler.

2.2.2.4 Immutability

Mutability means the possibility of change of something. If something is mutable, it could suffer alterations internally or externally due to external agents or internal behavior [7]. So the opposite, immutability, means something does not perform changes after created.

This becomes an important concept in functional programming because saying that a program is mutable means a program is lossy, so strange bugs can appear then. Some functional programming languages implement unique immutable data structures named *trie data structures* [4]. The program's execution can not modify the properties of these structures regardless of the level of the property in the hierarchy. *Trie data structures* use structural sharing to allow sharing reference memory location, using less memory, and better performance for certain operations.

2.2.2.5 Side effects

Side effects are the changes in the application state that are visible from outside the called function [7]. Functional programming does not use them, providing clearness when understanding the code and making it easier to test.

Like the immutability case, some functional programming languages implement what they call *monads* [60]. These structures provide isolation and encapsulation of side effects from pure functions.

2.2.2.6 Higher order functions

Some patterns as object-oriented programming tend to use attributes and methods in the object's context and forcing them to work only with the types defined of their parameters [7]. Functional programming has some functions that can operate with multiple types without having to determine various implementations using different signatures.

These functions are higher-order functions, and normally, they can take another function or functions as arguments. They can even return another function as a result of the

execution. A program commonly uses higher-order functions for multiple purposes, such as to abstract or isolate actions, effects, or async flow control. Moreover, to create utilities that can receive any arguments or partially apply a function to its arguments and to create curried functions. Also, for function composition or to orchestrate some functions from a list passed as an argument and return the composition as a result.

A common example of this concept is function `map()`, present in every functional programming language. The function can map over objects, strings, numbers, or any other data type because it takes a function as an argument that appropriately handles the given data type.

2.2.2.7 Commonly used data structures

Functional programming uses some specific data structures such as containers, *functors*, lists or streams [7]. They are not unique from functional programming, but they provide advantages such as efficiency or concurrency into multiple processes or even machines, this structures facilitate this performance.

Functors [3] are some special structures whose main characteristic is that they can be mapped over. They are containers with an interface whose methods can apply a function to the values inside it. Functors are *mappable*.

2.3 Web applications

A web application [63], generally known as Web App, is an application software hosted in a web server that serves resources to clients. Computer-based software programs differ from web applications because the device's operating system stores them locally.

Web apps [61], typically, do not require any additional installation but the browser. As technology keeps changing constantly, some sites need some browser complements to perform specific tasks to provide a better user experience. Nowadays, the internet contains many web apps, from small dynamic tools to graphic software or browser video games. Web technologies have grown so that companies usually produce two options for applications; the native OS software and the web app ones. Both of them have the same functionalities, and even web apps include some additional ones such as sharing with other users or integration with other web apps.

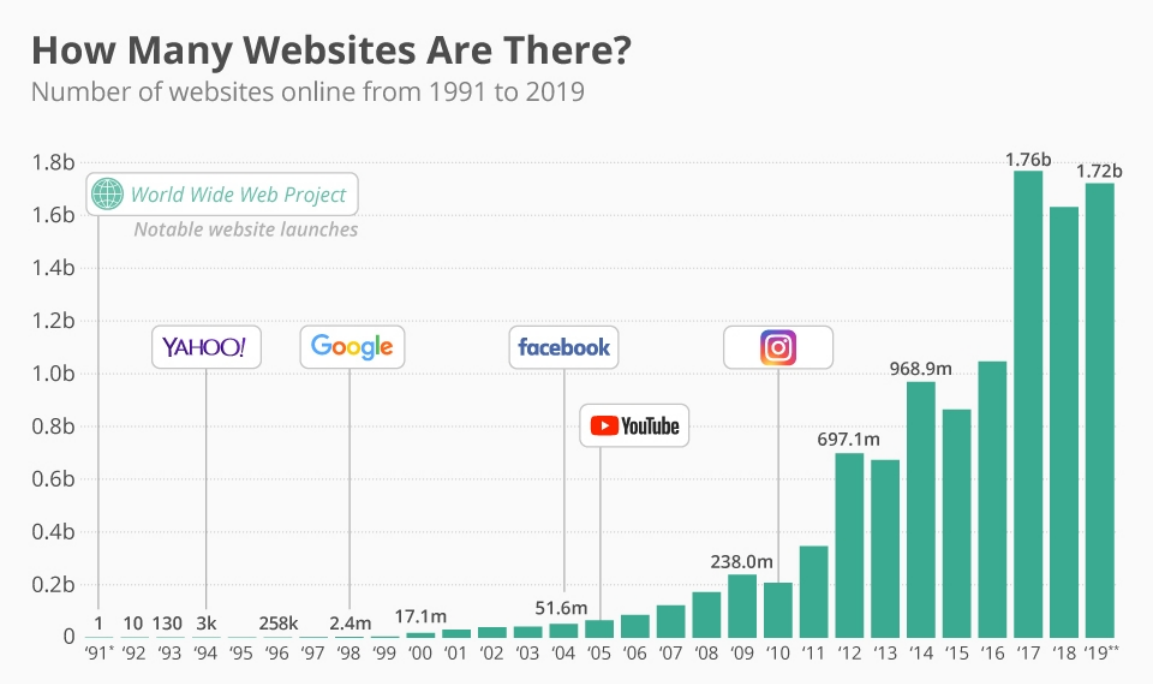

Figure 2.5: Number of websites over time
source by:
https://www.statista.com/chart/19058/how-many-websites-are-there/

Figure 2.5 shows the growth of website over time, showing the number of active sites per year. As the image outlines, the number of web apps has increased significantly over the last twenty years from the creation of the web. The tendency shows how web apps become increasingly more important for the supply of services in society.

2.3.1 Native apps and Web apps

2.3.1.1 Native apps

Developers create native apps [61] based on the requirements and characteristics of the platforms that will execute them. Applications should run in a broad set of OS such as traditional computers running Windows, macOS, or Linux and newer devices such as smartphones with Android, iOS, or even IoT gadgets. Actually, companies focus on the new developments of native apps on smartphones or mobile devices. This fact is important because it has resulted in a new category called Native Mobile Apps or Mobile Apps. This category covers all the developed software running on mobile devices.

The main characteristic of native apps is that they only run on a dedicated platform [61]. However, their main drawback is that if an application needs to have compatibility with multiple OS, developers should build a different application for each platform, considering how OS manages information, I/O, and all the components. This results in a higher effort in all aspects, a more significant budget, higher development time, more personnel, etc. Nevertheless, native apps fit perfectly to the platforms they run and can use all OS modules.

2.3.1.2 Web apps

Unlike native apps, web apps [61] do not need to perfectly suit in all the platforms because their access is through a web browser. Their main advantage is the facilities they provide to users, which use them on every browser from all devices. Moreover, developers need only to implement just one application, in contrast with the previous case. This case has an exception: each browser has its own implementation, so some web apps components could not work correctly on some browsers.

Another advantage of web apps over native apps is the update manner: developers just have to upload the new version to the hosting server. In native apps, users have to download manually and install the newly developed version of each platform's software. This becomes important when discussing security because web apps rapidly correct vulnerabilities found, but native apps need users to become aware of applying updates.

Nowadays, web apps can provide a broad range of services to users [61]. For instance, Google suite includes multiple web apps such as Google Maps, Gmail or even its own search engine, or Amazon with online shopping, Amazon Video or Amazon Music, or Facebook with WhatsApp, Instagram, or Facebook social network. These applications usually have two versions, the web app and a native app for each platform that acts as an interface to connect with the server. These second apps do not follow precisely the definition of native apps shown above because they are network connection dependent.

Some traditional native applications, such as Microsoft Office environment, are moving into web apps to extend their native applications' features. For example, Office offers services such as Word, Excel, or PowerPoint online, providing users an alternative to installing their native apps. Another feature of web apps is collaborative work, allowing users to create, read, edit, or delete documents simultaneously. Web apps provide this functionality because they centralize the information in servers, and users can share the view of a document. Depending on the application, sometimes people can use a web app locally without having an Internet connection.

	Native App	**Web App**
Platform	Platform dependent	Platform independent
Data storage	Users device	Commonly in the server, but can store some data in the browser
Native OS functions	Commonly used	Can use some of them in some occasions but it is not common and usually require user to give permissions
Source	Repositories, App stores and downloaded from websites	Web access
Setup	Imperative	Not needed
Update	Users should install manually	Service provider implements them and are available to all users
Internet connection	Commonly not needed	Imperative

Table 2.1: Native apps and Web apps comparison

Table 2.1 shows a comparison of the main characteristics of both types of applications [61]. Both of them have some important specifications, but none is the best one in this scope as they fit different requirements.

2.3.2 Client-Server architecture

Client-Server architecture [64] is primarily the most used architecture model for the development of web services [62]. This architecture basically defines the separation between providers and service demanders, receiving the names of servers and clients, respectively. Clients requests servers for resources, which are responsible for creating a response that clients can interpret. This separation allows developers to concentrate all the business logic in a single machine and ensures clients' simplicity, not requiring, in many cases, a high computing capacity.

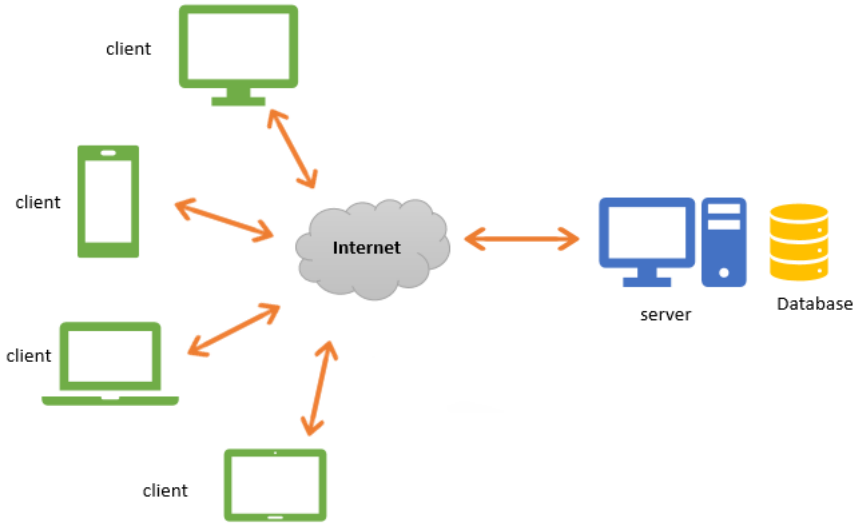

Figure 2.6: Client-Server architecture diagram
source by: `http://www.qababu.com/2019/07/api-testing-05-client-server.html`

The use of this architecture has certain advantages in this area, such as [62]:

- Clients share logical and physical resources. This occurs by centralizing the servers in a Data Processing Center, which is responsible for managing the servers.
- Clients initiate the requests for resources when and servers wait passively for connections. When clients receive the resources requested, they close the connection so that the components do not need to maintain a constant connection.
- Transparency with the physical location of clients and servers. Clients often do not know where the location of the responding server. This feature allows traffic balancing, assigning the less busy server to the more demanding clients and avoiding the saturation of some servers that receive a vast amount of requests.
- Independence of hardware and software platforms, as well as service encapsulation. The details of the different implementations used on the server-side are transparent to clients.
- Simple maintenance, as the architecture disperse responsibilities. Service providers can replace, repair, upgrade, and relocate a server while clients remain unaffected.
- Enhanced security management, because servers and resources have better control access to ensure that only authorized clients can perform some types of requests.

Moreover, servers require authorization to manipulate data and to apply updates.

Nevertheless, the client-server architecture also has some drawbacks that are important to choose this architecture or to consider other alternatives [62].

- Service providers should continuously monitor the server's load to detect and prevent server overload. This could happen when servers receive a considerable amount of simultaneous client requests producing traffic congestion.
- If servers do not receive the traffic correctly balanced, when a critical server fails, the impact of centralized architecture can cause an interruption in the service.

2.3.3 Java Web Applications Development

The development of web applications can use many technologies present in the market. At the beginning of the Internet, web servers' main goal was to provide resources located in their network. Nowadays, they perform so many business logic activities and their architecture has become definitely more complicated. New technologies have appeared in order to carry on different tasks and improving performance significantly. Some well known are [67] NodeJS, Ruby, Go, Java, Python, C#, Scala, PHP...

For this project, the technology selected is Java because of the simplicity of integrating the web application with the OptaPlanner constraint solver modules.

2.3.3.1 Model-View-Controller pattern

The most commonly used model for developing Java web applications is Model-View-Controller [6], dividing responsibilities into these components.

- **Models:** the layer responsible for data structure and management, containing mechanisms to operate and store information. The primary use resides mainly to perform functions as creation, recovery, modification, and deletion of data, so the defined models should implement these functions.
- **Views:** They maintain the code which produces the visualization of the user interfaces. Usually, they are dynamic and use some additional languages allowing servers to include code or data. This code coexists with other common languages, and when the server renders web pages, the content results in an organized and elegant way.

- **Controllers:** They connect the other two components and contain all the logic of the application. They are the orchestrators deciding what to do based on the data and requests received. To do so, they use functions of the model and render the corresponding views as the response.

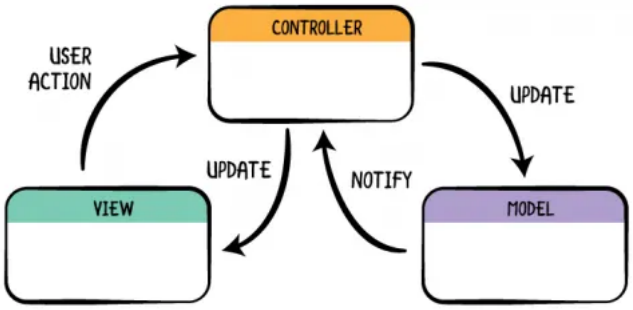

Figure 2.7: Model-View-Controller diagram
source by: `https://stips.wordpress.com/2019/04/15/model-view-controller-mvc-en-ios-un-enfoque-moderno/`

2.3.3.2 Web Application Structure

Java bases web applications in using a structure called Servlets for controllers and JSPs language for the development of views [23]. Also, dynamic web applications WAR file, the one generated for production, in Java follows a particular structure shown in Figure 2.8.

- **Static files:** As its name outlines, files which do not change such as images, styles, media, documents or scripts.
- **Java classes:** Contains the logic of the application including servlet files and additional Java classes.
- **Manifest file:** The *MANIFEST.MF* file, where developers should list all the extensions that the WAR needs.
- **Lib files:** Stores libraries as jar files.
- **Deployment Descriptor:** *web.xml* file, containing originally a mapping for servlets, welcome pages, security configurations, session timeout settings, etc...

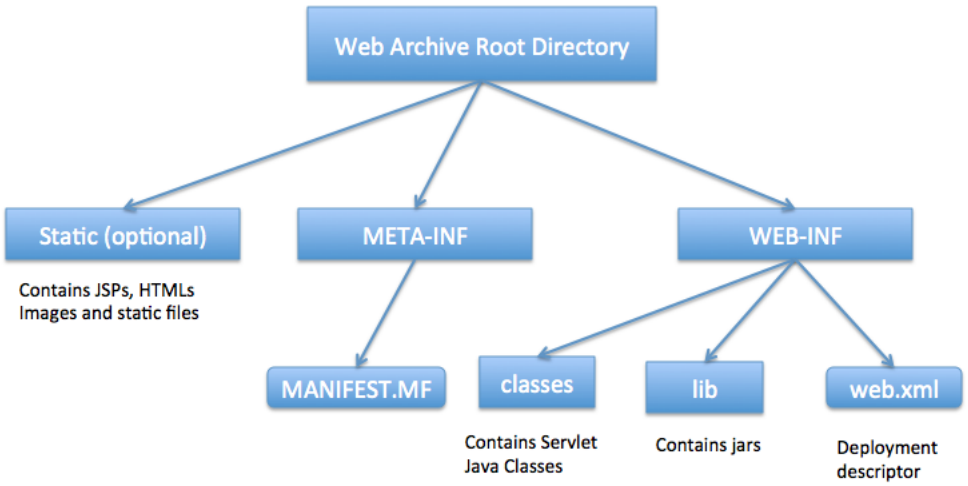

Figure 2.8: Dynamic Web Application WAR file structure
source by: `https://stackoverflow.com/questions/30796114/using-property-files-in-web-applications`

2.3.3.3 Web server frameworks and modules

The development of web application implies the use of many developed frameworks and modules to make easier the web application implementation, this section exposes the ones used in the project.

Spring Framework [46]: Designed to provide a comprehensive programming and configuration model for Java-based applications. Spring offers infrastructure support at the application level, providing a complete model for both the configuration and programming of business applications developed with Java. Moreover, the main advantage is the change in programmers' focus, allowing programmers to stay centered entirely on business logic.

Spring is a modular framework that allows programmers to use only the required modules for their applications. The framework uses some of the existing technologies, such as ORM, JEE, Quartz and JDK timers, logging frameworks, and other visualization frameworks. Spring provides templates for various technologies such as: JDBC, Hibernate, and JPA, so programmers do not need to write extensive code, as these templates simplify work [21].

Spring Web MVC [47]: Is the web module built on the Servlet API and included in Spring from its origins. The module dispatches requests to handlers through a dispatcher servlet

and configurable mappings, view resolution, locale, and theme resolution. This module's main tags to identify the application components are *@Controller* and *@RequestMapping*. The framework uses the first to recognize the controllers and the second one to define mappings between URIs and their controller's servlet. Programmers can easily configure requests and responses parameters using this module.

Spring Security [48]: Module part of the Spring Framework in charge of the authentication and access control for a web application. Some of its features are the protection against attacks like session fixation clickjacking, cross-site request forgery, a base login form, and a logout method, among other things. The module offers some customizable options such as to use an own login form, the algorithm used to encrypt passwords, the definition of user details, possible roles for user authorization, or the way to recover users from storage. Moreover, the framework allows the integration with other authentication and authorization network methods such as Kerberos, OAuth, or SAML.

Apache Tiles [9]: Provides an easy mechanism for creating modular view templates. The user can define basic templates with a set of components that are common on multiple views. When the application renders some view, the module loads each of the components of the view's template. This behavior has some advantages: developers do not need to rewrite code and reuse the common components easily. Moreover, this module allows users to develop applications with a uniform look and feel effortlessly.

String Template [24]: This module allows users to create string templates with some parameters introduced dynamically. Users store the template in a file, and when required, the program loads the template and injects the values of the parameters defined in the template. The result is a string that the application can use for multiple purposes. The behavior is similar to the defined with views in a dynamic application, where controllers inject the values into the views' templates.

Apache Tomcat [10]: Is an open-source web container that hosts Java servlets and JSP. The container includes a compiler that generates servlets from JSP and executes dynamic web applications from their WAR file. Apache Tomcat works as a web server in environments with a high level of traffic with high availability. The language of development for Apache Tomcat is Java, so it can work in every operating system that contains a Java virtual machine [59].

2.3.3.4 Persistence

Web applications traditionally use a persistence system to keep data between sessions. The most common persistence systems are databases, that store data usually in servers. Actually, users can find two types of databases: relational and non-relational databases. The first one stores information in a structured way and requires defining the attributes of an entity primarily. The second one can store any data type and does not require information to have a closed structure.

MySQL [54] is the most popular open-source relational database management system. It uses a client-server architecture, so a server stores data and web applications act as clients. One of its main characteristics is the reading speed while not using transactional functions, a commonly used feature in web programming. Furthermore, the language used for performing queries is an extensive subset of SQL and databases give a critical relevance of data indexing. MySQL would not be the best candidate for concurrent environments because of their problems in this aspect, but this project does not use concurrency [68].

At the time of developing web applications, they manage persistence through DAOs and Data Repository services. One of their main advantages is the abstraction of the application from the storage methods, reducing the use of query languages when implementing the server's code. Moreover, if the server changes of persistent system, the application's modifications will consist of adapting DAOs and Data Repository services to the new environment.

CHAPTER 3

Problem characterization

This chapter explains all the details of the problems that the system will try to solve. Firstly, the chapter details the problematic and how people solve these problems actually. The following section contains the main characteristics of a scout association that concern this problem. Afterwards, a section presents the groups of restrictions that a solution should comply with a brief explanation. Finally, the chapter contains an example of a problem and a possible solution.

3.1 Problematic

This project's main goal is the development of an intelligent system able to create feasible groups of scout leaders to take care of different groups of children.

A scout association performs multiple activities during the school year with children of all ages. The association divides these children into small groups between fifteen to thirty kids per group so that scout leaders can easily take care of them. The groups of children are immutable for all the activities performed by the scout association. These groups of children have a team of scout leaders associated that does not change during the whole year.

When every school year starts, the scout leaders join various meetings to start with the activities preparation. The agenda of one of these meetings include the elaboration of the scout leaders' teams. As the scout leaders' teams are immutable, the planning should be as perfect as possible. If the association needs to make some changes to the groups, the impact should be minimum on the actual planning.

To do this planning, all of the scout leaders give their opinions of some aspects to prepare the best scout leaders' teams. This makes more complex the activity of creating a planning, typically spending a considerable amount of time. The people of the association have tried some attempts to make the task of planning easier; for instance, sometimes the primary responsible of the association prepares multiple drafts and presents them in the meeting. Naturally, the drafts suffer modifications because they do not consider all the essential aspects regarding scout leaders, unit preferences, methodology knowledge, etc.

Making an unsuitable group planning could have a negative impact on the association. This could involve problems between scout leaders, trouble with children, instabilities in scout leaders' teams, overconfidence between children and their scout leaders, lack of personnel for some activities and so much more. Wrong planning could cause instability in the whole association.

3.2 Scout Association Structure

Scout associations have a common structure composed of *sections*. *Sections* are composed in turn by one or more *units*.

The concept of section was created because of the methodology to use based on the ages and the point in the maturity process where kids are. That is the reason why the smaller

sections, called Beaver Scouts and Cub Scouts, are set in children's stories to make easier for them knowledge acquisition. When they keep growing over eleven years old, this is no longer necessary and scout leaders can reflect with them in a similar adult manner.

Figure 3.1 shows an example of the structure of an association. In this case, the association contains five sections. The names of the sections are the common ones used for every group, as scout associations follow the methodology of international organisms such as the WOSM [27]. Scout associations could have fewer sections for some reasons like the lack of scout leaders, the lack of children in this age range, or even the lack of expertise on some sections methodology.

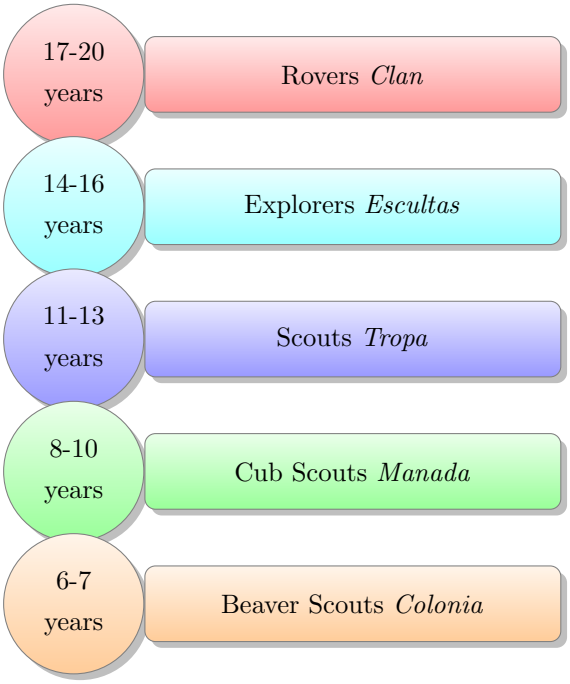

Figure 3.1: Structure of reference of sections of a scout association

As another example, Figure 3.2 illustrates the structure of units of an existing scout group association in Spain. The association contains nine units and the five sections exposed previously in Figure 3.1. This association has more than two hundred children and thirty scout leaders, so it has a considerable size. Moreover, the association has nine units, each one with a scout leaders' team associated. This one is not a common scout association because of its size. However, the solution is based on this association because it defines the most complex situation the system would face.

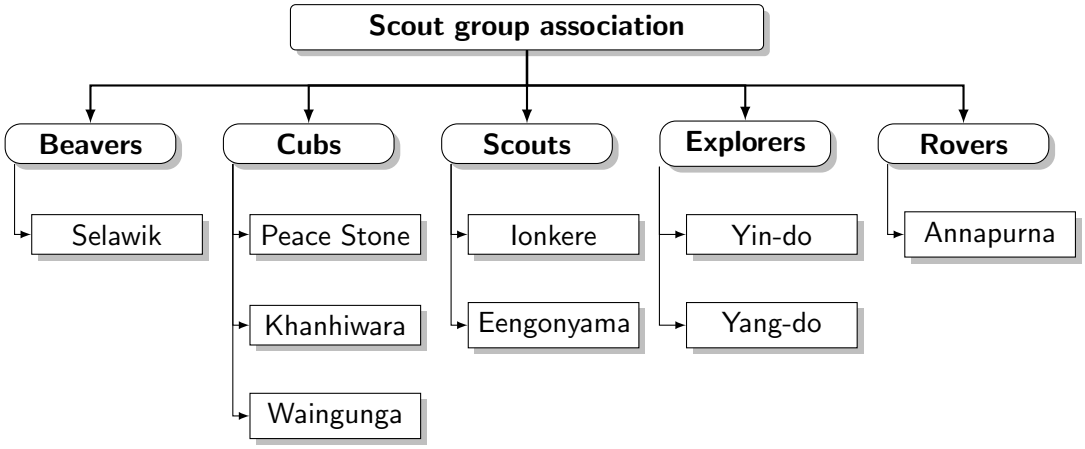

Figure 3.2: Example of Scout group association unit structure

3.3 Feasible planning characteristics

An optimal planning is a set of assignments of scout leaders into teams that comply with some restrictions. The analysis of the details of scouts' associations provides the set of constraints that valid plannings need to have. Before detailing the constraints required, we should describe the activities a scout association carries out.

Scout associations perform multiple activities with children during the school year. These activities could be of three different types:

- **Weekly meetings:** Performed usually on weekends, and lasting less than one day. The location where the meeting takes place is typically in a park outdoors in order to allow children to move freely.
- **Weekend camps:** Small camps of one night's stay located close to the place of residence. Generally, leaving on Saturday morning and returning on Sunday afternoon.
- **Long term camps:** At the end of every scholar term, the association organizes a long term camp. Two of them take place at Christmas and Easter, and there is another in summer. Firsts ones usually last for five days and the summer camps duration is about fifteen days.

With the information about a scout association's activities, we could extract some important characteristics that scout leaders' teams should have. The following classification enumerates the restrictions that each planning should respect to consider it optimal.

Structural restrictions:

These group contain constraints related to the structural composition of every unit. The aspects of these constraints are the identity of scout leaders and the size of the scout leaders' teams.

- **No repeated scout leaders:** The association performs all the activities simultaneously, so a scout leader can not be in two units at the same time.
- **Number of scout leaders per unit:** Each unit has a determined number of children of a range of ages. Based on this fact, each unit must have a fixed number of scout leaders per unit that the planning must respect.

Availability restrictions:

The availability of scout leaders is an important fact to consider in terms of attendance to activities. The association may not have scout leaders' teams where any members can not attend one or more association activities.

- **Weekend camps attendance in a unit:** As weekend camps take place during weekends, the scout leaders' team may require to have at least one person who can attend to them.
- **Camps attendance in a unit:** Like the previous case, but here the number of scout leaders who can attend would not be so high as to weekend camps. The main reason is that the association would have scout leaders in an employed situation so that they might have incompatibilities.

Social restrictions:

Social restrictions respect relations between scout leaders. Every workgroup can have problems between people that disrupt harmony in the workgroups. Their main goal is to ensure that every scout leader is comfortable with their colleges.

- **Banned colleagues in the same unit:** People whom a scout leader can not work with because their personalities have incompatibilities. A person can not work with others because of heavy problems in the past between them or because they have been in a relationship or so much more.
- **Preferred colleagues in the same unit:** The opposite as the other case, the idea is to respect as possible the people whom a scout leader would like to work with. This

criterion is not mandatory, but as much as it is respected, more satisfied the scout leaders would be in their teams.

Restrictions related to kids requirements:

Due to the age and maturity of the kids, the units might have some restrictions.

- **Mixed scout leaders' teams:** Units containing younger children may require to have a father figure and a mother figure.
- **Age of scout leaders in the unit:** Units containing younger children may be suitable for younger scout leaders, and older scout leaders fit better in units with older kids. When kids become teenagers, the ways to reason with them change, and older scout leaders become more suitable because of their maturity.

Training restrictions:

Scout leaders' teams need to have people with knowledge in some aspects, such as the unit's methodology or how the association performs processes. The project considers only one restriction, but in the future can appear more of them.

- **Scout leaders with experience in the unit:** Scout leaders' teams may require to have people with experience in the teams in order to train newer scout leaders.

Restrictions related to preferences:

Scout leaders may have some preferences in the assignation. The planning should comply as much as possible to guarantee the satisfaction of the personnel. Like the previous case, this category can receive future requirements, while now it only contains one.

- **Preferred units to be in:** Scout leaders may prefer to be assigned to some units than in others.

The ideal planning is the one that complies with all these requirements, but in practice, plannings can not meet all of them. The main reason is that trying to optimize one criterion could break others. If we prioritize the requirements, the task of creating optimal planning becomes more manageable. While the planning optimizes some high priority requirements, it breaks others could without having high relevance.

3.4 Problem example

This section details an an example of a real problem to solve by the intelligent system developed in this project.

A scout association contains three of the sections shown in Figure 3.1 (Cubs, Scouts and Explorers), and one unit per section. At the beginning of the school year, the association has eight possible scout leaders with some of their characteristics shown in the table of Figure 3.1. The association has fifty five children, divided into groups of ages from fifteen to twenty kids per section.

For simplicity reasons, this example does not take into account preferred units and preferred colleges. The availability is shown in terms of: *WC* meaning weekend camps attendance and *C* meaning camps attendance.

Name	**Gender**	**Age**	**Experience**	**Availability**	**Banned people**
Javier	Male	18	No	WC, C	Teresa
Antonio	Male	19	Yes	WC, C	-
Sofía	Female	18	No	-	-
María	Female	22	Yes	WC, C	Juan Carlos
Juan Carlos	Male	21	Yes	C	María
Manuel	Male	27	Yes	WC	-
Teresa	Female	24	Yes	WC, C	-
Laura	Female	23	Yes	WC	-

Table 3.1: Example of scout leaders reduced characteristics

As shown, the problem has much important information to consider for producing a feasible solution, such as how many scout leaders each unit needs to have, which units may have scout leaders from both genders, the planning can not have scout leaders with eighteen years old in charge of Explorer or Rover sections because the difference of ages is not enough, each unit should have scout leaders with attendance to all the activities of the association, the planning must consider problems existing between scout leaders, etc.

In the initial meeting between scout leaders, they decide that units with younger kids should have a scout leaders' team composed by three members. Moreover, the teams related to Cubs and Scouts should be mixed teams having scout leaders from both genders: male and female. The scout leaders' team associated to Explorers needs to have two people and do not require to be a mixed team.

Unit	**Team size**	**Age range**	**Mixed team**
Cubs	3	18-23 yo	True
Scouts	3	18-23 yo	True
Explorers	2	22-27 yo	False

Table 3.2: Unit characteristics in the example

With the details provided in the tables above, a possible solution that meets all the requirements is shown here.

Unit	**Scout lead.**	**Size**	**Mixed team**	**Age**	**Exp.**	**WC avl.**	**C avl.**
Cubs	Sofía María Antonio	OK *3/3*	OK *1 male, 2 female*	OK *18, 22, 19*	OK *2 true*	OK *2 true*	OK *2 true*
Scouts	Juan Carlos Javier Laura	OK *3/3*	OK *2 male, 1 female*	OK *21, 18, 23*	OK *2 true*	OK *2 true*	OK *2 true*
Explorers	Manuel Teresa	OK *2/2*	OK *1 male, 1 female*	OK *27, 24*	OK *2 true*	OK *2 true*	OK *1 true*

Table 3.3: Possible solution to the example problem

As the table of Figure 3.3 shows, the solution meets all the requirements so that it would be a feasible solution. The planning respects the units' size, the scout leaders' teams are mixed, the planning respects the age ranges, the planning has people with experience in all teams and available for the activities. However, the table does not show the information related to the banned scout leaders. Figure 3.1 defines that Javier can not stay in the

same team as Teresa and the same for Juan Carlos and María. The assignations of Figure 3.3 have respected all the restrictions covered. Finally, the example does not consider two restrictions for simplicity: the preferred units and preferred teammates for every scout leader. The system developed has been designed to consider all.

The example uses random data, but each case is completely different and the finding of the optimal solution is related to the input data. Some cases can not have a feasible solution and the association have to conform with the best solution found.

CHAPTER 4

Requirement Analysis

This chapter presents the requirement analysis, which is fundamental in software development. The methodology eases in identifying actors and use cases to help to define requirements. All this information applied to the project is detailed here.

4.1 Use cases

This chapter analyzes the problem characterized in the previous chapter to obtain the requirements that the system needs to meet expectations. Figure 4.1 shows the general diagram of the use cases and the actors involved. The following sections detail the actors and how to interact with the system to provide the final list of requisites.

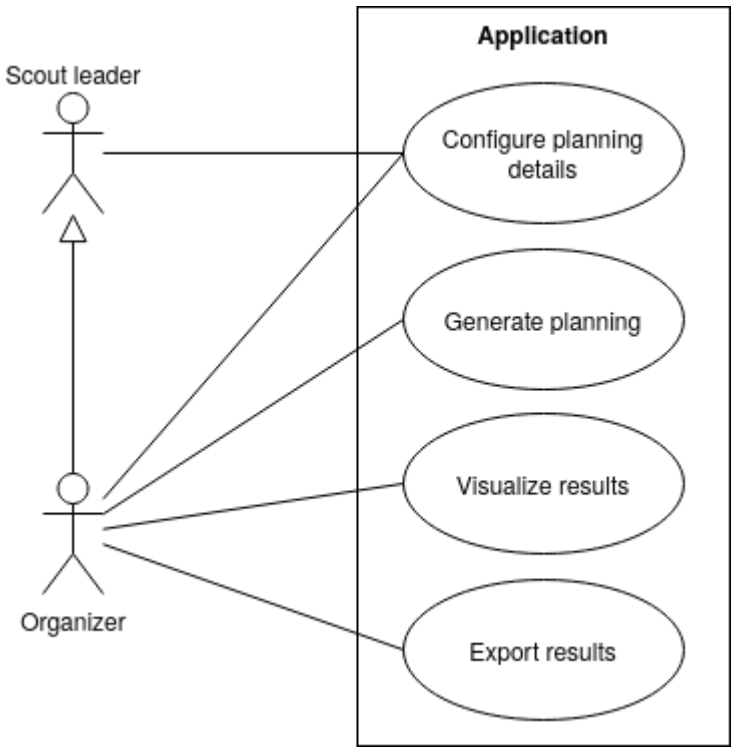

Figure 4.1: Use cases diagram

The system will be used by two actor roles, organizer and scout leader. Organizer is the main actor. He uses the system for I) the configuration of the details of the planning; ii) generate the planning, and iii) visualize the results, through a visual interface. In addition, he can export the results. Scout leaders only contribute to the planning configuration providing their personal data and preferences. Uses cases are further detailed in the following subsections.

- Configure planning details, explained in Sect. 4.1.2.
- Generate planning, explained in Sect. 4.1.3.
- Visualize results, explained in Sect. 4.1.4.
- Export results, explained in Sect. 4.1.5.

4.1.1 Actors

The problem involves two actors: Scout leaders and Organizer, that normally is the legal representative of the association. All their details are included in the following table.

Actor identifier	Role	Description
ACT-1	Organizer	The main user of the application. Commonly is the legal representative of the association and has the final say in every subject concerning the association's organization. Normally, he is an scout leader.
ACT-2	Scout leader	Prepares activities for the kids of the unit assigned with the other scout leaders of his team.

Table 4.1: Actor list

4.1.2 Configure the planning details use case

This use case is the first one to perform in the process. A planning has multiple parameters to configure and requires both actors. The use case contains in turn, two additional use cases.

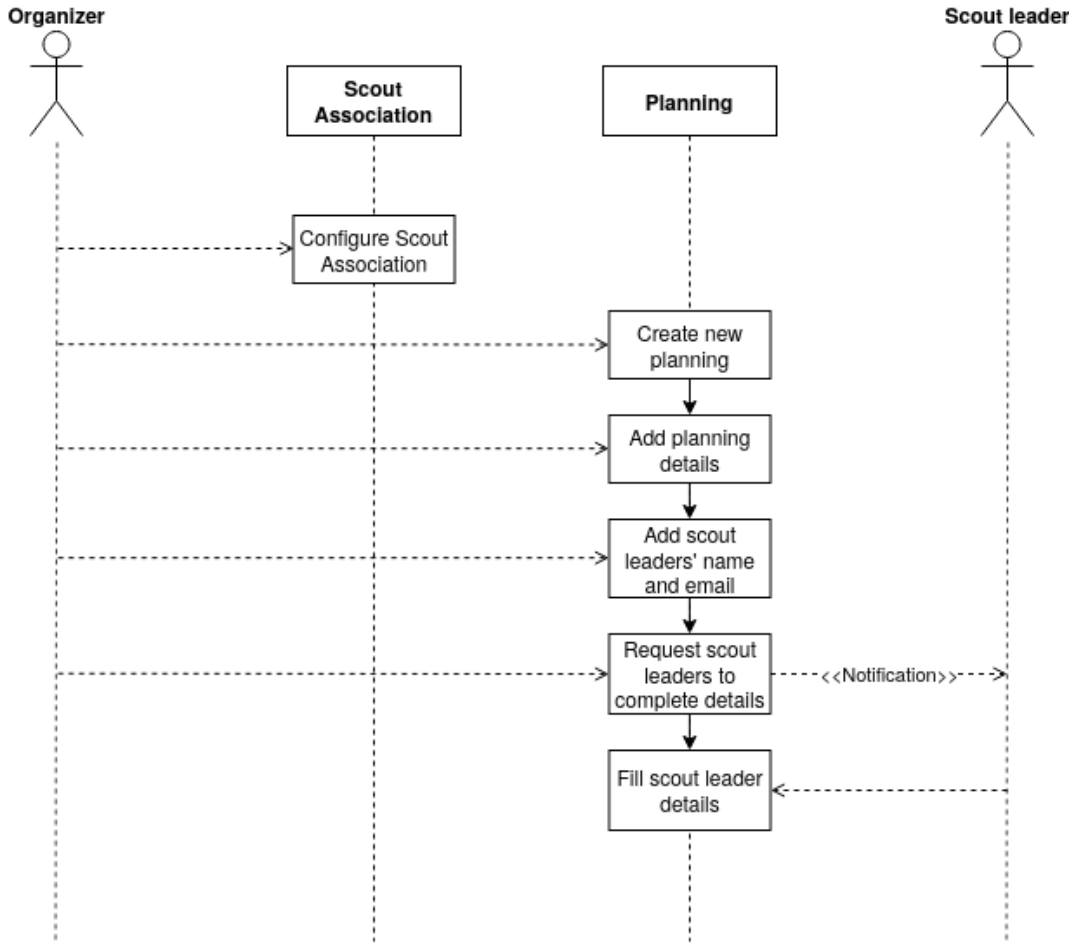

Figure 4.2: Configure the planning details sequence diagram

- *Configure the scout association parameters*, detailed in Sect. 4.1.2.1.
- *Configure the planning parameters*, detailed in Sect. 4.1.2.2.

4.1.2.1 Configure the scout association parameters

This use case is explained in Table 4.2.

Use Case Name	configure the scout association parameters		
Use Case ID	UC1.1		
Primary Actor	Organizer		
Pre-Condition	The application shows the home screen and the organizer visualizes the available options.		
Post-Condition	-		
Flow of Events		**Actor Input**	**System Response**
	1	The organizer presses *Create association* option.	The application shows the form for the introduction of the association's details.
	2	The organizer introduces the association's general details.	The application stores the association's information and redirects to the association view containing the details introduced.
	3	The organizer configures sections.	The application stores the sections' information and redirects to the association view.
	4	The organizer configures units.	The application stores the units' information and redirects to the association view.

Table 4.2: Configure the scout association parameters use case

4.1.2.2 Configure the planning parameters

This use case is explained in Table 4.3.

Use Case Name	configure the planning parameters		
Use Case ID	UC1.2		
Primary Actor	Organizer		
Pre-Condition	The organizer has configured the association parameters. The application shows the home screen and the organizer visualizes the available options.		
Post-Condition	-		
Flow of Events		**Actor Input**	**System Response**
	1	The organizer presses *Create planning* option.	The application shows the form for the introduction of the planning's details.
	2	The organizer introduces the planning general details.	The application stores the information and shows the planning view.
	3	The organizer adds scout leader names and emails.	The application stores the scout leader information and redirects to the planning view.
	4	The organizer presses *Send notifications to scout leaders* option.	The application generates requests and sends them to scout leaders.
	5	The scout leader presses in the link of the request.	The application loads the form.
	6	The scout leader fulfills the details requested in the form.	The application stores the details of the scout leader.

Table 4.3: Configure the planning parameters use case

4.1.3 Generate planning

After carrying out all the configurations, the application can generate the planning. The use case related to the generation of a planning is shown in Table 4.4.

Use Case Name	generate planning		
Use Case ID	UC2		
Primary Actor	Organizer		
Pre-Condition	All the configurations have been performed. The organizer is in the planning view.		
Post-Condition	-		
Flow of Events		**Actor Input**	**System Response**
	1	The organizer presses in the *Generate planning* button.	The application asks for confirmation.
	2	The organizer confirms.	The application starts generating the planning and redirects to the visualization view.

Table 4.4: Generate planning use case

4.1.4 Visualize results

Once the planning has finished solving, the application presents the results. The use case related to the visualization of the results of a planning is shown in Table 4.5.

Use Case Name	visualize results		
Use Case ID	UC3		
Primary Actor	Organizer		
Pre-Condition	The generation of the planning has finished. The organizer is in the home view.		
Post-Condition	-		
Flow of Events		**Actor Input**	**System Response**
	1	The organizer presses in the plannings section.	The application loads the organizer plannings.
	2	The organizer choose the planning to visualize.	The application loads the resolved planning view.

Table 4.5: Visualize results use case

4.1.5 Export results

After carrying out all the configurations, the application can generate the planning. The use case related to the generation of a planning is shown in Table 4.6.

Use Case Name	export results		
Use Case ID	UC3		
Primary Actor	Organizer		
Pre-Condition	The generation of the planning has finished. The organizer is in the planning view.		
Post-Condition	-		
Flow of Events		**Actor Input**	**System Response**
	1	The organizer presses the *Export planning* button.	The application presents the exportation formats.
	2	The organizer chooses the exportation format.	The application generates the file and downloads it.

Table 4.6: Export results use case

4.2 Requirements

After identifying the use cases relating to the kind of problems this project focuses on, the analysis provides the following requirements. Software engineering describes two types of requirements: Functional requirements and Non-functional requirements.

4.2.1 Functional requirements

Functional requirements define the basic system behavior. They specify the actions that the system must do or must not do. This section contains the functional requirements for this project.

- **FR1:** *The organizer* must be able to configure the association details: *association general details*, *sections* and *units*. Use case 4.1.2 provides this requirement.

- **FR2:** *The organizer* must be able to configure the planning parameters: *planning general parameters* and *scout leaders' name*. Use case 4.1.2 provides this requirement.
- **FR3:** *The organizer* should be able to request to scout leaders to complete their details. Use case 4.1.2 provides this requirement.
- **FR4:** *The scout leader* should receive the request to complete its details by email. Use case 4.1.2 provides this requirement.
- **FR5:** *The scout leader* must be able to configure its details. Use case 4.1.2 provides this requirement.
- **FR6:** *The organizer* must be able to generate the planning. Use case 4.1.3 provides this requirement.
- **FR7:** *The organizer* must be able to visualize the results of a generated planning. Use case 4.1.4 provides this requirement.
- **FR8:** *The organizer* may be able to export the results of a generated planning. Use case 4.1.5 provides this requirement.

4.2.2 Non-functional requirements

While functional requirements define what the system must do or what it must not do, non-functional requirements specify how the system should perform the actions. They do not affect the basic functionalities of the system. For the project, the system has to consider the following requirements.

- **NFR1:** Planning generation should provide a solution in a short period of time. Use case 4.1.3 provides this requirement.
- **NFR2:** The application directories and files should be easy to move/copy and execute in commonly used platforms. Use cases 4.1.2, 4.1.3, 4.1.4 and 4.1.5 provide this requirement.
- **NFR3:** The application must export the results in commonly used and generally accepted formats. Use case 4.1.5 provides this requirement.
- **NFR4:** Application screens may be simple and user-friendly. Use case 4.1.2 provides this requirement.

- **NFR5:** The application must request identification by username and password to configure, create, visualize and export plannings. Use cases 4.1.2, 4.1.3, 4.1.4 and 4.1.5 provide this requirement.
- **NFR6:** The application should have an API to allow interaction from other applications. Use cases 4.1.2, 4.1.3, 4.1.4 and 4.1.5 provide this requirement.

CHAPTER 5

Architecture

This chapter presents the architecture of the application developed in the master thesis. The architecture contains three main components: the intelligent system, the web server, and the persistence system.

5.1 General architecture

The project's main goal is to develop an intelligent system capable of distributing scout leaders into workgroups. The project has the requirement to develop a web application to allow non-technical people access to the intelligent system. Figure 5.1 shows a diagram of the architecture of the application developed and its components.

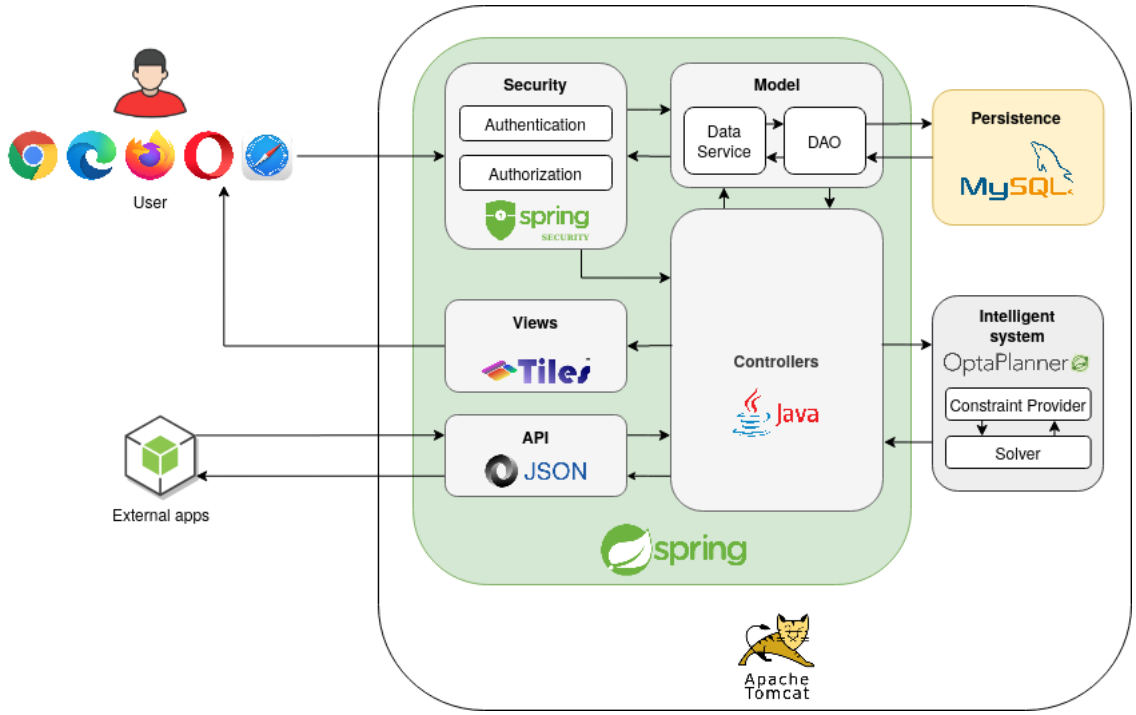

Figure 5.1: Architecture of the application diagram

As the diagram shows, the system has three important components hosted in the same machine that are:

- **Web server:** Structured following the Model-View-Controller pattern [6], is the component that manages the rest of the elements present in the architecture. The server provides authentication and authorization to the architecture, requests and retrieves information from the persistence system, generates the user interfaces and requests to perform operations to the intelligent system.

- **Intelligent system:** Is the key part of the architecture because it provides the business logic. Performs operations to solve users' problems in the context of the project and retrieves the solution. The intelligent system uses the OptaPlanner framework to allow the finding of solutions.

- **Persistence:** Allows the application to store and retrieve information by using a relational database with MySQL engine [54].

5.2 Intelligent system

5.2.1 Optaplanner

Optaplanner defines a common structure for the problems to solve. This structure has four important components [29]:

- **Planning entities:** Are the objects that get the assignation values to form a solution [32].
- **Planning solution:** Contains all the domains of the planning variables and the planning entities. This object is given to the solver to find the solution [34].
- **Constraints:** Define the requirements for generating a valid solution [36].
- **Solver Manager:** Solves the problem from its statement, configured in the planning solution and considering the constraints defined [28]. The Solver Manager contains the solving configuration.

5.2.1.1 Planning entities

Planning entities [32] are a key component in the architecture of Optaplanner problems. A planning entity has the following aspects to consider [29]:

- **Planing Entity:** Is a JavaBean (POJO) that changes during solving [32]. Normally, a planning problem has several planning entities, being all objects from one class. A planning problem can have more than one planning entity class but is uncommon. The annotation reserved for a planning entity class is *@PlanningEntity*.
- **Planning Variables:** Planning entities have one or more planning variables [35] that change during problem solving inside each planning entity. Moreover, planning variables have one or more defining properties [29], the immutable attributes describing each planning variable. The annotation for each planning variable is *@PlanningVariable*. Also, each planning variable needs to specify its domain, the candidate values that could be assigned.

- **Planing Id:** Is the identifier assigned to each planning entity [33]. The solver uses the identifier during assignments of planning variables to locate each planning entity. The tag *@PlanningId* identifies this variable.

5.2.1.2 Planning solution

OptaPlanner needs to define a class containing the parameters to allow the execution and finding a solution. A planning solution [34] initially contains the problem statement, and after performing the solving, the planning solution contains a possible solution for that problem. A possible solution does not have to be optimal or even feasible. The components for a solution and their characteristics are the following [29]:

- **Planning Solution:** A solution is mutable and changes during the execution [34]. For scalability reasons, the system manages only one instance of the solution class that is cloned to recall the one with the best score. The annotation tag associated with planning solutions is *@PlanningSolution*.
- **Planning Entity Collection Property:** Each planning solution can have more than one Planning Entity Collection Property [38]. As its name outlines, the property refers to the collection of planning entities that will be used. The tag associated to the Planning Entity Collection Properties for each problem is *@PlanningEntity-CollectionProperty*. All of the objects of the collection should be marked with the *@PlanningEntity* annotation.
- **Problem Fact Collection Property:** The problem facts used to generate a solution [40]. They do not change during solving, except if an event of type *ProblemFactChange* [42] is generated. As problem facts are part of the problem, they should not receive the annotation of *@PlanningEntity*. Constraints use the problem facts to create the solution. Each problem fact collection must have the annotation *@ProblemFactCollectionProperty*.
- **Value Range Provider:** The value range provider defines the domain of the variables [41]. It defines the relation between the planning variables and the possible values they can have. Planning variables share the identifier with the value range provider to make the relation.
- **Planning Score:** A planing solution must have exactly one planning score property [39]. The score allows to compare multiple solutions, being the one with the highest score the best one. The annotated attribute as the planning score holds that solution's

score, which can be `null` if the planning solution is not initialized. The solver modifies this property when the planning solution's score is calculated newly. The annotation the planning score should have is *@PlanningScore*.

5.2.2 Constraints

The constraints [36] are the key component of an OptaPlanner project because they define the correct or/and incorrect system's behavior. When an assignation does not comply with some criterion, the system can perform a penalization. Moreover, if some assignations improve performance, the system could also give a reward.

The framework defines multiple methods to declare constraints: using DRL or the recently developed Constraints Streams API [43]. Using DRL constraints implementation requires learning a different language, such as the one shown below.

Listing 5.1: Example of DRL constraint

```
rule "Don't assign Ann"
    when
        Shift(getEmployee().getName() == "Ann")
    then
        scoreHolder.addSoftConstraintMatch(kcontext, -1);
end
```

The Constraints Streams API [43] is based on Java 8 Streams and SQL [20], and has the advantage that users can write constraints in Java. Constraints Streams use functional programming to act over streams efficiently.

Listing 5.2: Example of Constraint Stream constraint

```
Constraint constraint = constraintFactory
        .from(Shift.class)
        .filter(shift -> shift.getEmployee().getName().equals("Ann"))
        .penalize("Don't assign Ann", HardSoftScore.ONE_SOFT);
```

All the constraints must provide a score if they are met. The score provided by each constraint have some properties [37]:

- **Score signum:** Positive or negative. To penalize or reward if some constraints are met. This allows the system to know the constraints to maximize and minimize.

- **Score level:** Simple or multilevel. To define multiple levels of constraints in a type of problem depending on their importance. The program will prioritize the compliance of the constraints with more importance, and once it has reached the best score on that level, the program tries to do the same with the next level.
- **Score weight:** To define the degree of penalization or reward if some constraint is met. This property allows ordering constraints by importance inside each level.

Depending on the number of levels to use by the score, OptaPlanner provides some classes that can be used directly. The user can also define a custom score implementing the interface `Score` that must implement methods to be `Comparable` [37].

- **Simple Score:** Just one score level. Depending on the type of score values to use in a problem, OptaPlanner supplies various classes: *SimpleScore* for integer values, *SimpleLongScore* for integers with a wide range of values, or *SimpleBigDecimalScore* for decimal values.
- **Hard Soft Score:** Two score levels. Hard score must be a positive number or zero to consider a solution feasible by the OptaPlanner framework. This type of score also has classes depending on the type of values that store the score like the previous case: *HardSoftScore*, *HardSoftLongScore*, and *HardSoftBigDecimalScore*.
- **Hard Medium Soft Score:** Three score levels. Like the previous case, OptaPlanner considers a solution feasible while the hard score is positive or zero. The implementation of this type of score contains only one class: *HardMediumSoftScore* for integer score values.

OptaPlanner implements another type of score more complex than the previous ones exposed, that is Pareto Scoring [37]. This type of score is used for more complex that deals with multi-objective optimization.

5.2.3 Solver Manager

The Solver Manager [28] is the component that initiates the process of finding a solution. This element reads the configuration that could be defined by multiple means, in an additional XML file [56], using the application's properties file (*application.properties*) under resources folder, or even declaratively calling methods from the Java API of the OptaPlanner framework.

The configuration of the solver contain the following aspects [29]:

- **Entities and solution:** The planning entity class [32] and the solution class [34].
- **Score function and constraints file:** The type of score class and the file containing the problem's constraints.
- **Termination configuration and optimization algorithms:** The configuration related to the execution's termination and optionally configure the optimization algorithms to use.

Once configured, the Solver Manager [28] has a method called `solve` that receives an object of the planning solution class. This method starts the execution of finding a solution to the problem. When the system finishes executing, `solve` returns a Solver Job, which is a kind of bundle. The bundle contains the planning solution solved with the planning entities fulfilled with the planning variables assigned and a UUID to identify that solution.

5.2.4 Application to the project

The intelligent system is based on an OptaPlanner program structure [16]. To comply with this structure, the program needs to define the elements explained in the previous section. Figure 5.2 shows the components of the intelligent system and the processes.

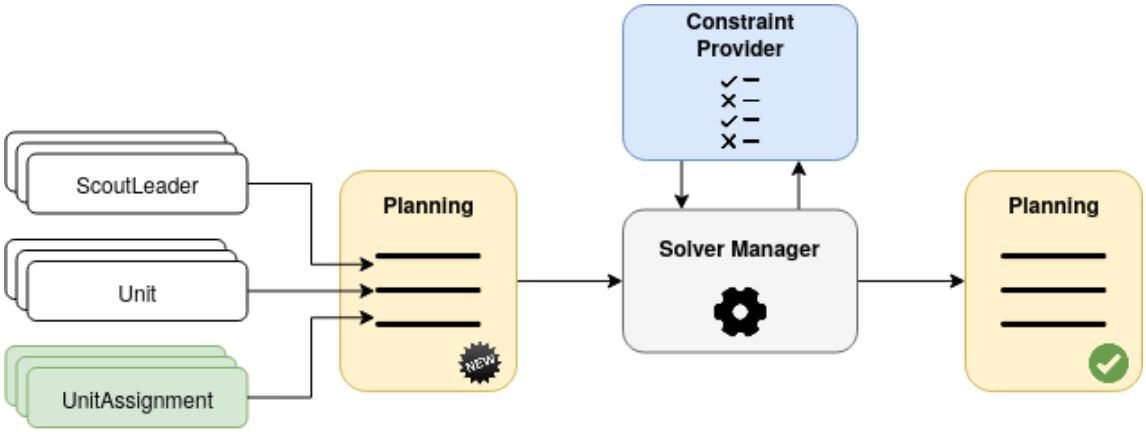

Figure 5.2: Intelligent system components diagram

The system first introduces the object entities *ScoutLeader* and *Unit* configured for the problem to an empty *Planning* object. Then, the same *Planning* object acquires the empty planning entities of the *UnitAssignment* class. The solver receives the configured *Planning* object and starts running to find a solution considering the constraint provider's constraints.

When the execution finishes, the solver produces another *Planning* object with the planning entities fulfilled.

5.2.5 Entities and solution

The intelligent system defines some components in order to perform scheduling operations. Figure 5.3 shows three important types of them: Planning Entities [32], Problem Facts [29] and Planning Solution [34].

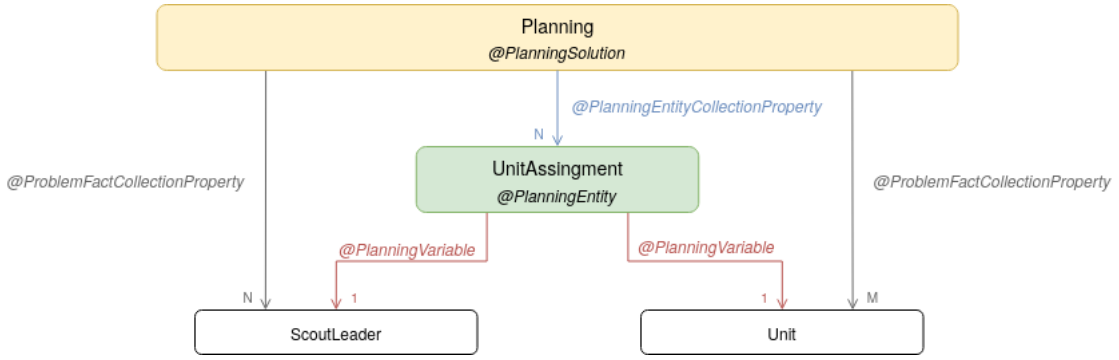

Figure 5.3: OptaPlanner elements diagram

5.2.5.1 Planning Entities

The Optaplanner components for planning entities explained in Sect. 5.2.1.1 applied to the project are detailed here.

- **Planning Entity:** For this project, the planning entity class is *UnitAssignment*.
- **Planning Variables:** In this case, the planning variables for each *UnitAssignment* are one *Unit* and one *ScoutLeader*.
- **Defining Properties:** For the case of *ScoutLeader*, its defining properties are:
  - Id[1]
  - Gender
  - Age
  - Experience

[1]Some constraints use the unique identifier to distinguish each ScoutLeader uniquely. Once a *ScoutLeader* gets an Id, it does not change during the solving process, so it is also considered a defining property.

- Camp attendance
- Weekend camps attendance
- Preferred units
- Preferred colleagues
- Banned colleagues

Similarly, defining properties for *Unit* are:

- Id[2]
- Section
- Number of scout leaders
- Age range
- Mixed unit

Listing 5.3: Planning entity: UnitAssignment

```
@PlanningEntity
public class UnitAssignment {

  @PlanningId
  private UUID id;

  @PlanningVariable(valueRangeProviderRefs = "scoutLeadersRange")
  private ScoutLeader scoutLeader;

  @PlanningVariable(valueRangeProviderRefs = "unitsRange")
  private Unit unit;

  ...
}
```

5.2.5.2 Planning Solution

The project implements the planning solution components with the following classes and attributes.

- **Planning Solution:** For this project, the planning solution class [34] is *Planning*.

[2]Same case as *ScoutLeader*.

- **Planning Entity Collection Property [38]:** For the project, the system has just one type of planning entities being all objects of the class *UnitAssignment*.

- **Problem Fact Collection Property:** For this case, the system considers two problem facts collections [40] the *ScoutLeader* list and the *Unit* list.

- **Value Range Provider [41]:** This case contains a relation between the *ScoutLeader* attribute in the planning entity (*UnitAssignment*) with the problem fact collection which contains all the possible objects of *ScoutLeader* class the attribute can have. Similar to this case, the problem fact collection of *Unit* is associated with the planning variable related in each *UnitAssignment* object.

- **Planning Score [39]:** In this case, as the constrains use three levels of score, the score is of the type *HardMediumSoftScore* [30], that include the the three levels required.

Listing 5.4: Planning solution: Planning

```
@PlanningSolution
public class Planning {

  @ProblemFactCollectionProperty
  @ValueRangeProvider(id = "scoutLeadersRange")
    private List<ScoutLeader> scoutLeaderList;

  @ProblemFactCollectionProperty
  @ValueRangeProvider(id = "unitsRange")
    private static List<Unit> unitList;

    @PlanningEntityCollectionProperty
    private List<UnidadAssignment> unitAssignmentList;

    @PlanningScore
    private HardMediumSoftScore score;
    ...
}
```

5.2.6 Constraint Provider

This component is the key component of this project, due to the fact that contains the constraints to provide solutions to the problem. The project uses the Constraints Streams API based on Java 8 Streams and SQL [20], instead of using DRL [57].

OptaPlanner provides an interface to declare constraints allowing the framework to identify the class that contains them. The `ConstraintProvider` interface [31] requires to implement the method `defineConstraints` which returns an array of Constraint objects.

Listing 5.5: Method defineConstraints

```
@Override
public Constraint[] defineConstraints(ConstraintFactory constraintFactory)
    {
    unitNumber = constraintFactory.from(Unit.class).groupBy(count());
    return new Constraint[] {

        // Hard constraints
        nonRepeatedScoutLeadersConflict(constraintFactory),
        scoutLeaderNumberConflict(constraintFactory),
        scoutLeaderAgeConflict(constraintFactory),
        bannedScoutLeadersConflict(constraintFactory),

        // Medium constraints
        mixedUnitConflict(constraintFactory),
        weekendCampsAttendanceConflict(constraintFactory),
        longTermCampsConflict(constraintFactory),
        unitExperienceConflict(constraintFactory),

        // Soft constraints
        preferredScoutLeadersConflict(constraintFactory),
        preferredUnitsReward(constraintFactory),
        preferredUnitsConflict(constraintFactory)
        };
}
```

Depending on the punctuation on the level of the score of each constraint provides, the system contains three types of constraints: Hard, Medium, and Soft [30]. The types contain specific constraints depending on their importance.

- **Hard constraints:** They must not be broken in any case because the assignment would not be valid. Hard constraints are all the ones related to the basic structure of a *Planning*. The problem requires to have four hard constraints:
  - No repeated scout leaders
  - Number of scout leaders per unit
  - Age of scout leaders in the unit
  - Banned colleagues in the same unit

- **Medium constraints:** They should not be broken. The compliance fully or partially of them would make better assignments. Some of the solutions where medium constraints are broken may not be valid, but the relevance compared to the previous group of constraints is lower. The application considers four medium constraints:
  - Mixed units
  - Weekend camps attendance in a unit
  - Camps attendance in a unit
  - Scout leaders with experience in the unit

- **Soft constraints:** The compliance with all of them would produce the best *Planning* possible, but it is extremely complex. This group manages the preferences of the scout leaders. This case requires to define two soft constraints.
  - Preferred colleagues in the same unit
  - Preferred units to be in

5.2.6.1 Hard Constraints

This group contains four important constraints related to the structural format of the *Planning*. This section details all the hard constraints developed.

No repeated scout leaders: By default, OptaPlanner tries to create the best combinations without taking care of the repetition of the elements assigned. This constraint declares that if a scout leader is assigned to one unit, the system can not assign it another time.

Figure 5.4: No repeated scout leaders constraint diagram

To do so, the system compares the unique identifier of each scout leader by pairing two of them. If this constraint is broken, the global score will add a penalization of one hard negative point in its value.

Listing 5.6: No repeated scout leaders constraint code

```
private Constraint nonRepeatedScoutLeadersConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.fromUniquePair(UnitAssignment.class, Joiners.
        equal(UnitAssignment::getScoutLeaderId))
            .penalize("Conflict with the scout leader's id",
                HardMediumSoftScore.ONE_HARD);
}
```

Number of scout leaders per unit: When configuring the scout association parameters, the user defines the number of scout leaders that units should have. If the quantity of scout leaders assigned to a unit is higher than the predefined number configured, the constraint would penalize the global score.

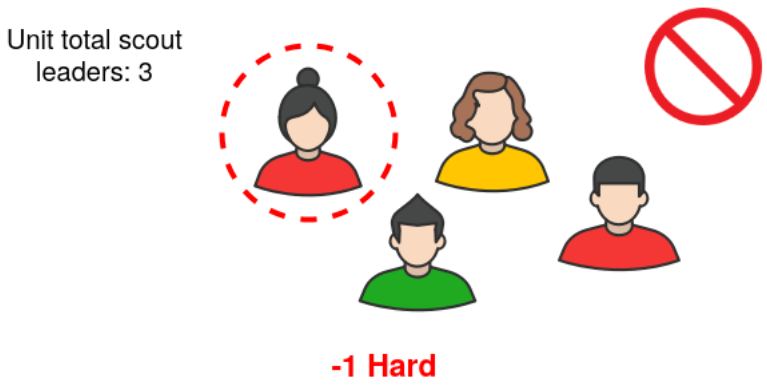

Figure 5.5: Number of scout leaders per unit constraint diagram

To perform this operation, the constraint groups all the assignments by unit, counting the elements per group. If the count's value is higher than the number of scout leaders predefined by unit, the global punctuation will penalize with one hard negative point.

Listing 5.7: Number of scout leaders per unit constraint code

```
private Constraint scoutLeaderNumberConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(Unit.class)
            .join(UnitAssignment.class, Joiners.equal(Unit::getId,
                UnitAssignment::getUnitId))
            .groupBy((unit, unitAssignment) -> unit, countBi())
            .filter((unit, unitAssignmentCount) -> unit.
                getScoutLeadersNumber() < unitAssignmentCount)
            .penalize("Conflict with the scout leaders' number in each unit
                ", HardMediumSoftScore.ONE_HARD);
}
```

Age of scout leaders in the unit: Each unit has associated a range of ages the scout leaders have to be within. This is important because younger scout leaders would find difficulties being in charge of older learners because they could be only one or two years difference. Moreover, if the system assigns older scout leaders to lower units, it may assign younger scout leaders to upper units, which is undesirable behavior.

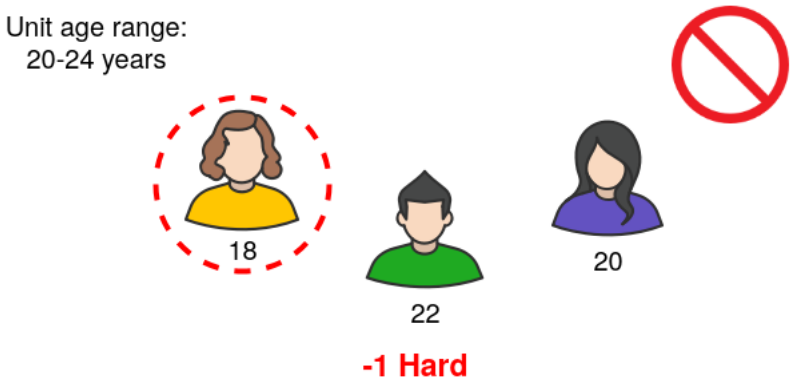

Figure 5.6: Age of scout leaders in the unit constraint diagram

This constraint starts comparing the scout leader age with the assignment's unit age range. If it is not included, the global punctuation will add one hard negative point.

Listing 5.8: Age of scout leaders in the unit constraint code

```
private Constraint scoutLeaderAgeConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class).filter(
            (unitAssignment) -> unitAssignment.getScoutLeaderAge() <
                unitAssignment.getUnitInitialAge()
                    || unitAssignment.getScoutLeaderAge() > unitAssignment.
                        getUnitFinalAge())
            .penalize("Conflict with the age of the scout leader",
                HardMediumSoftScore.ONE_HARD);
}
```

Banned colleagues in the same unit: A scout leader could have lots of reasons for not sharing the unit with others. An ideal environment may not have this type of issue, but problems happen between people and the system can not perform some combinations of scout leaders per unit. The constraint tries to avoid this behavior strongly.

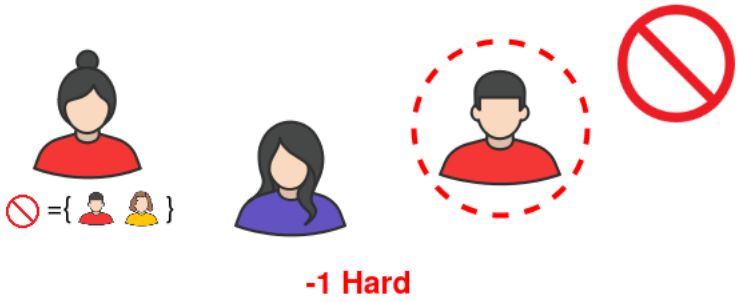

Figure 5.7: Banned colleagues in the same unit constraint diagram

To do so, the implementation of this constraint starts pairing *UnitAssignments* by the same unit. If one of the scout leaders in the pair has the other scout leader in its banned colleagues' list, the system must penalize the assignment. Similar to other cases, the penalization involves giving one hard negative point to the global punctuation.

Listing 5.9: Banned colleagues in the same unit constraint code

```
private Constraint bannedScoutLeadersConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.fromUniquePair(UnitAssignment.class, Joiners.
        equal(UnitAssignment::getUnitId))
            .filter((unitAssignment1, unitAssignment2) -> unitAssignment2.
                getScoutLeaderBannedScoutLeaders()
                    .contains(unitAssignment1.getScoutLeader())
                    || unitAssignment1.getScoutLeaderBannedScoutLeaders().
                        contains(unitAssignment2.getScoutLeader()))
            .penalize("Conflict with banned scout leaders",
                HardMediumSoftScore.ONE_HARD);
}
```

5.2.6.2 Medium Constraints

These constraints provide a better result once basic structural ones have been fulfilled. They are used to refine the result taking into account special aspects.

Mixed units: Some units may need to have scout leaders of both genders, male and female. This is mostly because of the ages of the kids. They might need to find a kind of father figure and mother figure in their leaders. Other reasons would be because kids have some needs, mostly biological, that could be solved by one gender in preference. The organizer would define which units require to have scout leaders from both genders.

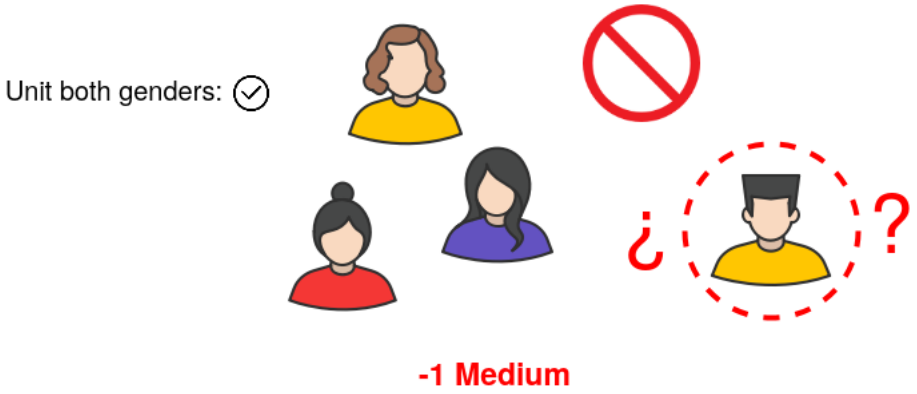

Figure 5.8: Mixed units constraint diagram

To allow having scout leaders from both genders in the units predefined, the constraint groups *UnitAssignments* by unit, counting the number of different genders that each unit has. If the unit needs to have scout leaders from both genders and the number of the counted different genders per unit is lower than two, the overall score suffers penalization. This case's penalization is different from the others explained previously, subtracting one medium negative point to the global punctuation.

Listing 5.10: Mixed units constraint code

```
private Constraint mixedUnitConflict(ConstraintFactory constraintFactory) {
    return constraintFactory.from(UnitAssignment.class)
            .groupBy(UnitAssignment::getUnit, countDistinct(UnitAssignment
                ::getScoutLeaderGender))
            .filter((unit, scoutLeaderGender) -> unit.isMixt() &&
                scoutLeadersGender < 2)
            .penalize("Conflict with mixed unit", HardMediumSoftScore.
                ONE_MEDIUM);
}
```

Weekend camps attendance in a unit: An important fact to do a valid *Planning* is distributing people who can attend weekend camps in all units. This allows each unit to have scout leaders available for this type of activity. If some unit does not have any

scout leader with weekend camps availability, the planning would not be valid. To consider satisfied this constraint, at least one of the scout leaders per unit needs to have positive attendance to weekend camps as a minimum requirement.

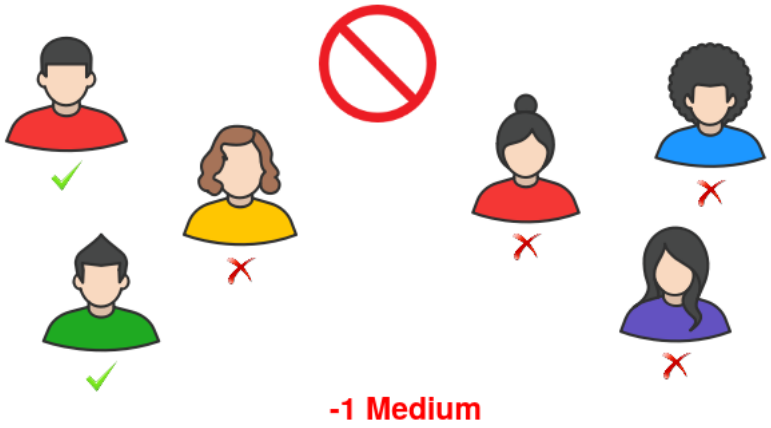

Figure 5.9: Weekend camps attendance constraint diagram

The constraint first groups *UnitAssignment* by units and by the attendance to this type of activity. Afterwards, the system groups newly the previous result by the attendance to weekend camps while counting each type's elements. The result provides the number of units that have scout leaders with attendance and without it. The last operation performed is to check if the number of attendance count is equal to the total number of units of the association. The constraint provider class contains an additional function performing the count and storing as an attribute of the class to know the number of units. This function is executed when the constraint provider is configured initially.

Listing 5.11: Weekend camps attendance in a unit constraint code

```
private Constraint weekendCampsAttendanceConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class)
                    .groupBy(UnitAssignment::getUnit, UnitAssignment::
                        isScoutLeaderWeekendCampsAttendance)
                    .groupBy((unit, isScoutLeaderWeekendCampsAttendance) ->
                         isScoutLeaderWeekendCampsAttendance,countBi())
                    .join(unitNumber)
                    .filter((isScoutLeaderWeekendCampsAttendance, count,
                        unitNumber) -> isScoutLeaderWeekendCampsAttendance
                        && count != unitNumber)
                    .penalize("Conflict with weekend camps attendance in a
                        unit", HardMediumSoftScore.ONE_MEDIUM);
}
```

Camps attendance in a unit: Similarly to the previous case, but now evaluating the attendance to long term camps. Commonly, this case is different from the previous one because employed people normally can not attend camps, or at least to all of them. As seen previously, the system should ensure that people who can attend long-term camps are correctly distributed in all the association units. As a minimum, the system requires assigning at least one scout leader per unit attending camps.

The code implementation is the same as the previous case, adapting variables and function calls.

Listing 5.12: Long term camps attendance in a unit constraint code

```
private Constraint longTermCampsConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class)
                    .groupBy(UnitAssignment::getUnit, UnitAssignment::
                        isScoutLeaderLongTermCampsAttendance)
                    .groupBy((unit, isScoutLeaderLongTermCampsAttendance)
                        -> isScoutLeaderLongTermCampAttendance,countBi())
                    .join(unitsNumber)
                    .filter((isScoutLeaderLongTermCampsAttendance, count,
                        unitsNumber) ->
                        isScoutLeaderLongTermCampsAttendance && count !=
                        unitsNumber)
                    .penalize("Conflict with long term camps attendance in
                        a unit", HardMediumSoftScore.ONE_MEDIUM);
}
```

Scout leaders with experience in the unit: Having a unit with all its scout leaders newly enrolled in the association can cause problems because of the work group's lack of knowledge. Every unit should have people able to teach new scout leaders and show them the unit's functioning, how to treat children and their parents, and solve any problem that could happen during the activities. This is similar to the cases of camps seen previously, where all the units require to have people with some specific characteristic. The system will try to set at least one scout leader with experience in each unit.

The implementation uses the same code as camps constraints but adapting functions and variables.

Listing 5.13: Scout leaders with experience in the unit constraint code

```
private Constraint unitExperienceConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class)
                    .groupBy(UnitAssignment::getUnit, UnitAssignment::
                        isScoutLeaderExperience)
                    .groupBy((unit, isScoutLeaderExperience) ->
                        isScoutLeaderExperience,countBi())
                    .join(unitsNumber)
                    .filter((isScoutLeaderExperience, count, unitsNumber)
                        -> isScoutLeaderExperience && count != unitsNumber)
                    .penalize("Conflict with experience in a unit",
                        HardMediumSoftScore.ONE_MEDIUM);
}
```

5.2.6.3 Soft Constraints

The group of soft constraints contains restrictions related to scout leaders' preferences. The system should respect constraints only when the maximum number of hard and medium constraints has been fulfilled. In this case, most of the constraints give positive punctuation to the overall score. This behavior gives a bigger reward if the system respect as many preferences as possible.

Preferred colleagues in the same unit: Scout leaders can select if they want any person to work with in the same unit. The idea is to insert one or two people in the list and the system will try to take them into account at the time of elaborating the planning. This constraint is more related to personal preferences than association requirements, so the non-compliance of this constraint will not break the feasibility of a possible solution.

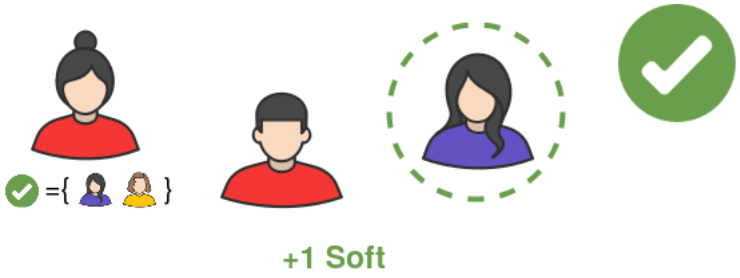

Figure 5.10: Preferred colleagues in the same unit constraint diagram

This constraint pairs *UnitAssignments* by units and checks if the preferred scout leader list of one scout leader of the pair contains the other scout leader. If this is met, instead of penalizing, the system rewards the soft overall score positively.

Listing 5.14: Preferred colleagues in the same unit constraint code

```
private Constraint preferredScoutLeadersConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class).join(UnitAssignment
        .class, Joiners.equal(UnitAssignment::getUnitId))
            .filter((unitAssignment1, unitAssignment2) -> unitAssignment2.
                getScoutLeaderPreferredColleagues().contains(
                unitAssignment1.getScoutLeader()))
            .reward("Reward with scout leader preferred colleagues",
                HardMediumSoftScore.ONE_SOFT);
}
```

Preferred units to be in: Similar to the previous case with preferred colleagues, this case handles units. Scout leaders can define an ordered list of the units present in the scout association. The list contains the units preferred in the firsts positions and the units where the scout leader does not want to be in the last positions. Newly, a scout leader could prefer to stay in a determinate unit, but if other higher priority constraints are not met, this preference will not be respected.

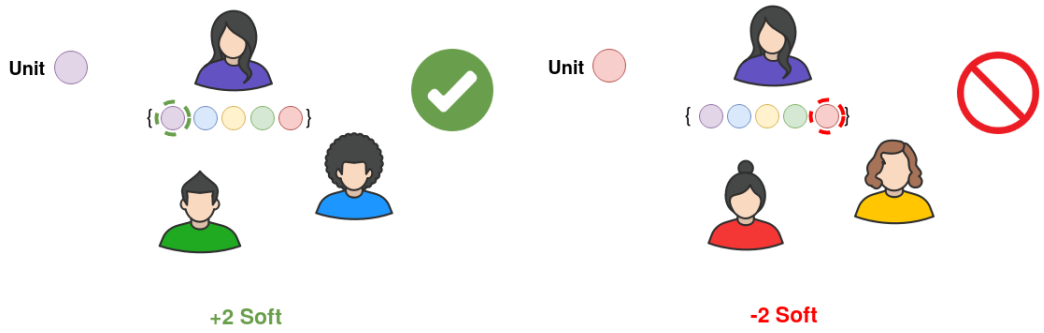

Figure 5.11: Preferred units to be in constraints diagram

To perform this operation, the *ConstraintProvider* contains two constraints implemented. Both of them look for the unit's position assigned in the preferred units list for a scout leader. If the list's position is the first or second one, the constraint assigns a reward of two or one soft points respectively to the overall score. However, if the list's position is the last or the penultimate one, instead of rewarding, the system penalizes negatively with two soft points.

Listing 5.15: Preferred units to be in by a scout leader constraints reward code

```
private Constraint preferredUnitsReward(ConstraintFactory constraintFactory
    ) {
    return constraintFactory.from(UnitAssignment.class)
            .filter((unitAssignment) -> {
                int position = unitAssignment.getPreferredUnitMultiplierPos
                    ();
                return position == 1 || position == 2;
            })
            .reward("Reward with preferred units", HardMediumSoftScore.
                ONE_SOFT, UnitAssignment::getPreferredUnitMultiplierPos);
}
```

Listing 5.16: Preferred units to be in by a scout leader constraints penalization code

```
private Constraint preferredUnitsConflict(ConstraintFactory
    constraintFactory) {
    return constraintFactory.from(UnitAssignment.class)
            .filter((unitAssignment) -> {
                int position = unitAssignment.getPreferredUnitMultiplierNeg
                    ();
                return position == 2;
            })
            .penalize("Conflict with preferred units", HardMediumSoftScore.
                ONE_SOFT, UnitAssignment::getPreferredUnitMultiplierNeg);
}
```

To know the list's position and the corresponding reward or penalization points, each object of the class *UnitAssignment* contains two additional methods. One of them returns the positive points to assign depending on the position: if the list's position of the unit is the first, the function returns two. If the list's position of the unit is the second one, the function returns one. In any other case the function returns zero.

The other function has similar but negative behavior. If the list's position of the unit is the last or the penultimate one, it returns two. In any other case, the function returns zero. The constraints definitions uses both functions to know if the constraints have to be activated and as multipliers on the negatives and positives points.

5.2.7 Solver Manager

The initial way OptaPlanner proposes to create the Solver Manager [28] is by using tags that automatically inject the component with the configuration described in the *application.properties* file. Moreover, OptaPlanner scans all the application components to find the tags defining all the OptaPlanner elements.

Listing 5.17: Solver Manager

```
@Inject
SolverManager<Planning, UUID> solverManager;
```

The Solver Manager initial configuration that the OptaPlanner framework proposes has some problems when using other additional frameworks. For instance; by using the Spring framework in the web application, some incompatibilities make this manner really complex or even impossible to create the Solver Manager this way.

To solve this problem, the easiest manner is to create the Solver Manager programmatically. This way requires to create two attributes: the Solver Config [44] and the Solver Manager Config [45]. The first one contains the configuration previously registered in the *application.properties* file. This case requires to indicate elements, such as the solution class and the entities, that were not needed in the previous manner of implementation. The code used to do so is the following.

Furthermore, the configuration of the Solver Manager using the Java API requires to declare the type of the definition of constraints. The solver can manage constraints could stored in an XML file [56] or by using the Constraint Provider interface. Also, the configuration should indicate the maximum time of execution, because if not, OptaPlanner executes indefinitely.

Listing 5.18: Solver Manager configuration

```
SolverConfig solverConfig = new SolverConfig()
        .withSolutionClass(Planning.class)
        .withEntityClasses(UnidadAssignment.class)
        .withScoreDirectorFactory(
                new ScoreDirectorFactoryConfig()
                    .withConstraintProviderClass(
                            PlanningConstraintProvider.class))
        .withTerminationConfig(new TerminationConfig()
                .withSecondsSpentLimit(60L));
```

```
SolverManager<Planning, UUID> solverManager = SolverManager.create(
    solverConfig, new SolverManagerConfig());
```

The code above reflects all the characteristics explained about the Solver Manager configuration explained in the previous paragraphs. It is the code developed for the web application. Once the Solver Config has all the important parameters configured, the problem's specific Solver Manager can be created. The solver manager's signature includes the solution class and the type of identifier for problems that will be submitted.

5.3 Web server

5.3.1 Spring Framework

The Spring Framework [46] eases developers to create applications. The framework provides tools to create easily web applications. The framework has some important components described in this section to understand the application implementation.

5.3.1.1 Beans

The framework bases the performance on a concept called Beans [55]. These are the objects that form the backbone of an application, and the Spring IoC Container is responsible for managing them. This allows the application classes to use objects from other classes without having to create them explicitly. The Spring IoC Container creates these objects when the application starts running.

To allow Spring IoC Container to identify the classes to create an object to inject them to others, the Spring Framework needs components to have annotations. Some of these annotations, and the most important ones, are:

- **Configuration:** Each configuration file must be annotated with the *@Configuration* annotation.
- **Controller:** Used to identify the controllers of the application (*@Controller*).
- **Service:** The service tag (*@Service*) indicates components that hold the business logic.

- **Repository:** Related to persistence components and oriented to catch persistence specific exceptions. The annotation for this class is *@Repository*.

When some class needs to have an object such as some repository or service, the object does not have to be created and tagged with the annotation *@Autowired*. Moreover, the class this object pertains must have one of the previous tags exposed related to Spring application components.

5.3.1.2 MVC Pattern

Using the Spring Framework implies applying the *Model-View-Controller* pattern [6] directly. This pattern distributes functions into the three components of the pattern. Each component represents a layer depending on the functions that perform: persistence, presentation and logic.

- **Models:** Include the entities and how to store and recover them from the persistence system. The persistent system interaction is produced normally by using Data Services and DAOs.

- **Views:** Include the presentation layer and all the components of web pages. The controllers render the views after performing business logic actions.

- **Controllers:** Cover the business logic of the application. Ask models to store and recover information from the persistence system, perform actions with the information and generate a response to the client that commonly is a view.

Models

Models are responsible for data storage and characterization. Java is an object-oriented language, so every type of data is considered as objects of different classes. Web applications manage three components relating to data characterization and storage that are:

- **Entities:** An entity is a class containing the attributes and methods related to a concept in the application.

- **Data Repository Service:** Is the class in charge of providing data retrieved from the storage and adapting the structures to the application needs.

- **DAO:** The objects responsible for contacting the database are *Data Access Objects* [19]. They provide CRUD operations (*Create*, *Read*, *Update* and *Delete*) to applications. The *Data Repository Service* calls DAOs when applications need to store and recover data in the storing system.

While using the Spring Framework, some of the components must have an annotation of the ones exposed in Sect. 5.3.1.1. In this case, the *Data Repository Service* needs to have the annotation *@Service* because it holds the methods to store and recover data. Furthermore, all DAOs must have the annotation *@Repository*, because they implement the methods purely to contact the storing system.

Views

The views are in charge of the presentation of information. They provide user interfaces to allow users to interact with the application. Dynamic web applications can present different information depending on the context, the user, and so many variables.

Views are JSP files [65] normally in Java. Moreover, to implement dynamic web application views, Java provides annotations called JSTL [58]. The Spring Framework also contains annotation to provide more functionalities.

To avoid developers have to repeat common code present on multiple views, views are split into components. Some common components on multiple views are the navbar, menus, application logo, login information, or footer. Apache has developed a module called Apache Tiles [9] that developers commonly use along with the Spring Framework to ease the distribution of code related to the views into multiple components.

The Apache Tiles module has a definition file (*tiles.xml*) with the components of each view to render. The file contains the name of the basic templates that include tags indicating the exact location of each of the components which compound the view. When a controller asks to render some view, Apache Tiles loads the proper components and injects them into the basic template to generate the view.

Listing 5.19: Example of tilex.xml

```
<tiles-definitions>
  <definition name="template" template="/WEB-INF/views/template.jsp">
    <put-attribute name="head" value="/WEB-INF/views/head.jsp" />
    <put-attribute name="header" value="/WEB-INF/views/header.jsp" />
    <put-attribute name="sidebar" value="/WEB-INF/views/sidebar.jsp" />
    <put-attribute name="footer" value="/WEB-INF/views/footer.jsp" />
```

```
  </definition>
  <definition name="home" extends="template">
    <put-attribute name="content" value="/WEB-INF/views/home/home.jsp" />
  </definition>
</tiles-definitions>
```

The XML code [56] above contains a simple example of the definition file of Apache Tiles. The file contains a basic template called *template* that imports all the components (head, header, sidebar and footer) except the content that will depend on the views to render. The other component (*home*) specifies the content of the view related to the home of the application. When the controller asks to render the home view, Apache Tiles loads the *template* component including the common elements and the content of the *home* definition.

Controllers

The controllers are the most important part of a web application, due to the fact that they contain the business logic of the application. An application can have multiple controllers in charge of performing multiple functions. The Spring Framework provide some tools to develop controllers in a more easier manner. Each controller class must be tagged as *@Controller* to allow the Spring Framework to identify the controller classes of the application.

Controllers contain multiple methods in charge of handling client requests. Traditionally, a controller should manage two types of requests: *GET* for requesting resources from the server and *POST* for creating and modifying resources. The methods inside a controller class should be able to manage both type of requests. The Spring Framework provides the annotation *@RequestMapping* to indicate the properties of the request that maps to the method which holds that request. The main properties of a request are the URI and the type.

Listing 5.20: Example of controller class method or servlet

```
@RequestMapping(value = {"/home"}, method = RequestMethod.GET)
public ModelAndView home(HttpServletRequest req) {
    ModelAndView mav = new ModelAndView("home");
    return mav;
}
```

The code above represents a simple servlet present in a controller class that loads a view called *home*, as the one explained in the previous section. As detailed before, the *@RequestMapping* tag specifies that the method holds *GET* requests with the URI */home*.

Five simple lines of code that perform lots of actions internally, this is one of the main advantages of using the Spring Framework.

5.3.1.3 Spring Security

Spring Security [48] is the Spring framework to manage authentication and access-control in an application. The framework provides different methods and classes that could be modified to adapt them to the application. Some of the customizable options are: the way of acquiring users from the storage, the definition of authorized sites for users, or the type of password encoder to use. Spring Security requires to implement the following modules:

- **Configuration:** The configuration of the authentication and authorization. The application should have a class implementing the `WebSecurityConfigurerAdapter` interface [49]. The class has to contain methods to describe how to retrieve users for comparison and the protected URIs that require login, and the type of users that can access them.
- **User Details Service:** This service is similar to the Data Repository Service because the User Details Service implements how to retrieve users from the storage. Like in the configuration, the class that contains the service has to implement the `UserDetailsService` interface [52].
- **User Details:** For Spring Security, the user definition should contain some methods to perform comparisons and permit access to determined URIs. So the user class in the applications that use Spring Security should implement the `UserDetails` interface [51].

Listing 5.21: UserDetails interface

```
public interface UserDetails extends Serializable {

  Collection<? extends GrantedAuthority> getAuthorities();
  String getPassword();
  String getUsername();
  boolean isAccountNonExpired();
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();

}
```

One important concept the framework defines are user roles or *authorities* [50]. Each role allows users to perform specific actions in the application. When some user tries to access a URL, the system checks its roles and the ones required to access there. If the user does not have enough permissions, he will not be able to access the mentioned URL. However, if the user can access that URL, the request is passed to the corresponding controller to process the actions to take.

5.3.2 Application to the project

The web server uses the Java Dynamic Web application framework. The structure of the development follows the Mobile-View-Controller pattern that the use of Spring MVC facilitates.

5.3.2.1 Models

Models represent data and how to acquire them from the storing system. The application performs these operations using three types of components explained in this section: *Entities*, *Data Repository Service* and *DAOs*.

Entities

As mention before, an entity is a class that represents a concept. The class contains its attributes and methods. The project contains the following seven entities with their relations, represented using *Unified Modeling Language* [11].

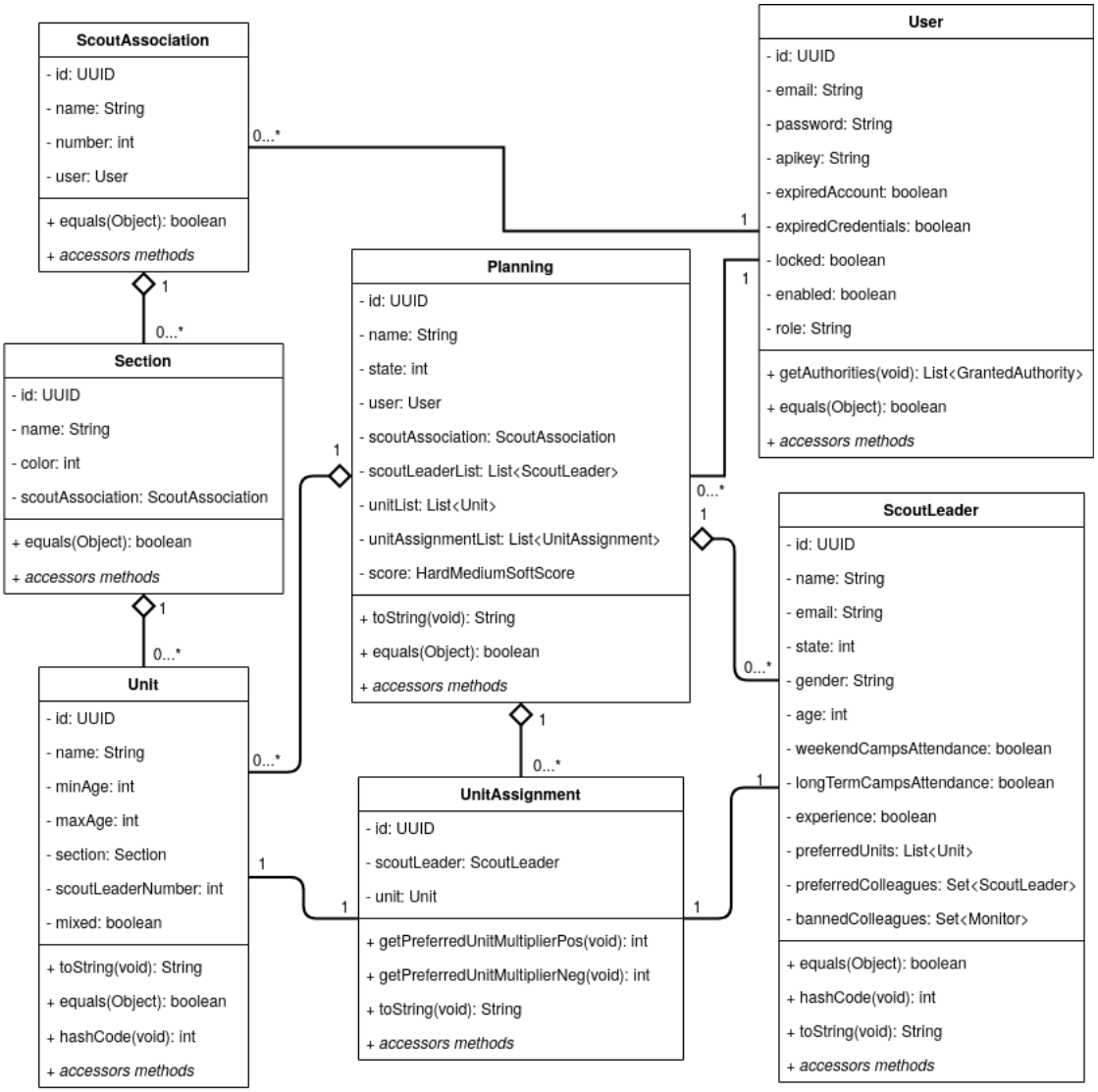

Figure 5.12: UML diagram

User

The application has users that perform the functions. The user class implements the `UserDetails` interface [51] provided by Spring Security [48] to implement authentication and authorization. The attributes of this class are the following:

- **Id:** The unique identifier of the user.
- **Email:** The email address of the user.
- **Password:** Encrypted password of each user. Spring Security performs encryption.
- **Api key:** The key to carry out authentication when using the application's API.

- **Expired account:** Boolean flag that represents if an account has expired. Actually, the application does not consider cases for account expiration.
- **Expired credentials:** Another boolean variable that indicates whether an account's credentials are valid.
- **Locked:** The system administrator can decide to lock some user for so many reasons: misbehavior, security incidents, etc. This boolean flag represents if a user is locked or not.
- **Enabled:** Like the previous case, the enabled flag identifies whether a user account is enabled or disabled. A user account could be disabled by the lack of use of the application.
- **Role:** The role a user has. Initially, the application considers two roles: Common user and administrator. The only difference is that an administrator can perform actions over user accounts.

Scout Association

The *ScoutAssociation* class represents the concept of the scout association. The class is really simple because it contains just four attributes.

- **Id:** The unique identifier of the scout association.
- **Name:** The name of the scout association.
- **Number:** Each scout association in Spain has a number. The number contains three digits.
- **User:** The user that created the scout association in the application.

Section

A section is a division of children of a range of ages. Similar to the *ScoutAssociation* entity, the class contains just four attributes.

- **Id:** The unique identifier of the section.
- **Name:** The proper name of the section.
- **Color:** Each section in Spain have a color for multiple purposes. Some of the purposes are: to help people identify the section a kid pertains and as a resource for the scout association's activities.

- **Scout Association:** The section belongs to one scout association.

Planning

A *Planning*, is the main objective to solve this problem. The attributes that describe a *Planning* are the followings.

- **Id:** The unique identifier of each planning.
- **Name:** The name of the planning.
- **State:** The planning state determines whether the planning is in a configuring phase, in the solving process, or solved.
- **User:** The user owner of the planning.
- **Scout Association:** The scout association that contains the units this planning has to assign.
- **Scout leader list:** Contains all the scout leaders participating in the planning and their details and preferences to elaborate the solution. This list is a Problem Fact Collection Property, of the ones mentioned in Sect. 5.2.5.2.
- **Unit list:** This collection lists the units and the configured parameters. The unit list is the other Problem Fact Collection Property.
- **Unit Assignment list:** List containing the relations between scout leaders and units. These are the assignations that the system has performed between each scout leader and unit. This list changes continuously when solving the planning, to evaluate different assignations until reaching the best planning. This attribute is the Planning Entity Collection Property, seen in Sect. 5.2.5.2.
- **Score:** This parameter hosts the *HardMediumSoft* score presented in Sect. 5.2.5.2.

Scout Leader

Scout leaders are the people to distribute in groups (*units*). To describe them, this problem uses the following characteristics.

- **Id:** The unique identifier of each scout leader.
- **Name:** The person's real name or a nickname used to identify him for operations carried out by users manually in the system.

- **Email:** To communicate with the scout leader, each one has to provide its email address.
- **State:** Similar to the *Planning* case, the attribute that determines whether a scout leader has completed its details or not.
- **Gender:** The scout leader gender (male or female).
- **Age:** The actual numerical age.
- **Experience:** Attribute defining whether the scout leader has worked previously in the association or is newly enrolled. This concept could be abstract, but the idea is to consider whether it is the first time for a scout leader to perform its responsibilities or has a long trajectory in the association as a relevant fact in group scheduling.
- **Camp attendance:** Determines whether the scout leader can attend the several camps performed by the scout association during the scholar year.
- **Weekend camps attendance:** Like the previous attribute, this variable specifies whether a scout leader can attend specifically to weekend camps.
- **Preferred units:** An ordered list containing the association units shorted, where starting positions are for the favorite units where the scout leader wants to be and final ones for disliked units, the ones where the scout leader may feel unpleasant.
- **Preferred colleagues:** Set that contains the preferred colleagues. The ones the scout leader wants to work with, but not excluding the rest of them.
- **Banned colleagues:** Similar to preferred colleagues, but in this case, each of the scout leaders contained here can not stay in the same work group with this one. The reasons why a scout leader can include others in this set could be various ones: personal problems, have been engaged before, or simply personality incompatibilities.

Unit

The system generates assignations between scout leaders and units. To perform the assignations, each unit has its own properties that the system should consider.

- **Id:** The unique identifier of each unit.
- **Name:** The name of the unit.
- **Section:** The unit's section.

- **Number of scout leaders:** Requirement of the total number of scout leaders to assign to the unit. The total number of scout leaders for all units must be equal to the total number of available scout leaders.
- **Age range:** Represented by two attributes: `minAge` and `maxAge`. This unit should contain scout leaders whose age may be within the age range defined. This parameter has significant relevance, especially for units containing older children. Usually, young scout leaders look after younger kids, and older ones take care of older children. This behavior is because younger scout leaders usually have one or two years more than the oldest children in an association, so they can not be their responsibles.
- **Mixed:** Because of the children's ages, some units require to have people of both genders to comply with their needs. The attribute is a boolean variable indicating whether to have scout leaders from both genders or is not relevant. Setting this attribute to false does not necessarily mean that a unit will have scout leaders exclusively from one gender in the unit.

UnitAssignment

This class is the Planning Entity described in Sect. 5.2.5.1. When the system finishes the finding of a solution, each *UnitAssignment* match a scout leader with a unit in a specific planning. So *UnitAssignment* class contains the solution relations in a planning. The details of the class are the following:

- **Id:** The unique identifier of each UnitAssignment.
- **Scout Leader:** The scout leader of the assignment.
- **Unit:** The unit assigned to the scout leader.

This class contains two additional methods that the intelligent system uses for the preferred units constraint detailed in Sect. 5.2.6.3. The methods evaluate the assignation and return an integer number that the constraint uses.

Data Repository Service

The service connecting to repositories containing the data related to the application. The Data Repository Service includes all the methods for the creation, recovery, edition and deletion of data from the storage system. This class provides centralization of all the possible sources of data, so that they can be called without the need of importing all the sources each

time. Normally, this component has two elements: an interface (`DataRepositoryService`) and its implementation (`DataRepositoryServiceImpl`).

Controllers have an instance of the interface when the application is initialized and use it when they require to perform one of the operations mentioned before. Due to the design of the storage system used, the data repository service needs to parse retrieved data into objects for the application, the rest of the methods (create, update and delete) are called directly from the different DAOs. If the storage system changes, or multiple ones are used depending on the type of data, the implementation of the methods will change but always complying with the interface.

DAOs

They are purely the connections with the storing system. Their main use is to abstract the application from the way the information is stored. For the project, the data storage system is a MySQL server executed in the same machine as the application. The language to interact with the database is SQL, so *Data Access Objects* [19] contain methods translating Java objects attributes into SQL statements to perform operations.

Like the Data Repository Service, each DAO contains two components: the interface and its implementation. This also allows changing the storing system without modifying all the application components that use DAOs. For instance, if developers decide to move from MySQL to MongoDB, the only action in the code will be to adapt DAOs to the new storing system, but the rest of the application would remain unchanged.

The application contains a DAO class associated with each table in the database, mostly corresponding to each entity. The project contains has the following ten DAOs:

- DaoScoutAssociation
- DaoScoutLeader
- DaoPreferredScoutLeaders
- DaoBannedScoutLeaders
- DaoPlanning
- DaoSection
- DaoUnit
- DaoUnitAssignment

- DaoPreferredUnits
- DaoUser

In comparison with the seven entities presented, the application contains three more DAOs. This is because of the need to store the *ScoutLeader* attributes `preferredColleagues`, `bannedColleagues` and `preferredUnits`. These attributes are collections of existing objects that need to be stored creating N:M relations. When the system asks to perform a specific *ScoutLeader* operation, the four DAOs are involved: *DaoScoutLeader*, *DaoPreferredScoutLeaders*, *DaoBannedScoutLeaders* and *DaoPreferredUnits*.

5.3.2.2 Controllers

The application contains four controllers in charge of the business logic. By having multiple controllers, the system distributes responsibilities and functions into all of them.

- **Main Controller:** Performs operations related to users sessions, user registration, rendering of the index page and manage of errors.
- ***Scout Association* Controller:** Covers operations related with the shaping of the structure of a scout association.
- ***Planning* Controller:** Allows the creation and modification of a *Planning*, adding and modifying scout leaders involved on a determined *Planning*, solving a *Planning* and visualizing of the results. Moreover, the controller includes servlets to export a *Planning* in different formats.
- ***API* Controller:** Responsible for the API, this controller contains the servlets for performing operations directly using the API. The details of the controller are in Sect. 5.3.4.

Main Controller

The Main Controller contains servlets to perform multiple general operations necessary for the application. The controller implements the following functions:

- **Log in:** A servlet maps the *GET* operation to supply the login form for non-authenticated users.
- **Sign up:** Two servlets perform these operations:

  - One provides the form view when the application receives a *GET* request.
  - The other receives *POST* requests and checks if the data of the user is valid. If the information is correct, the system stores it in the database creating a new user.

- **Error:** A servlet renders the error view when the system receives a request for the error page. If the application turns into an error, the *web.xml* file redirects to this servlet by using a *GET* request.

Listing 5.22: Error redirecting in web.xml

```
...
<error-page>
    <location>/error</location>
</error-page>
...
```

Scout Association Controller

The controller contains all the operations for shaping a scout association. A scout association has sections, and each section includes units. This controller performs operations related to these entities.

- **Scout Association operations:** The controller implements CRUD operations by using multiple servlets:
  - **Create Scout Association:** Is the only function which has two servlets associated: *GET* servlet rendering the create form and the servlet for *POST* requests that validates the data and stores it in the database.
  - **Read Scout Association:** This servlet is essential because it needs to load all the information of a scout association. After checking that the user is the owner of the scout association of the request, the servlet loads the scout association's sections and units. Finally, the servlet passes data to the view.
  - **Update and Delete Scout Association:** Each function has its own servlet for responding to *POST* operations. Firstly, the servlets check the validity of the request's information and the scout association's owner. The owner should be the session's user and, after the validations, the servlet performs the update or deletion.

- **Section operations:** Like the previous case, the controller implements CRUD operations for this entity. As the scout association's view contains sections, the implementation does not need to create an additional servlet to show the section's information. The controller contains several servlets for the mapping of *POST* operations because the scout association's view contains modals with forms to perform operations over these entities. As other cases, before carrying out actions over data, servlets perform validity and owners of the information checks.
- **Unit operations:** Like section's operations, the controller holds servlets to perform operations over units. Section's and unit's operations have only two differences: one is the volume of information required to provide for a unit, which is relatively broader. The other difference is the validity checks. In this case, units belong to sections, sections to a scout association, and a scout association to a user. For this case, the servlet needs to check all parents until arriving at the scout association's owner that should be the session's user. If one of these checks fails or checks on the forms' information, the method will not continue executing, and the servlet will redirect to the error page.

Planning Controller

After shaping the scout association, is the time for configuring the *Planning* parameters. To do so, the *Planning* controller contains the following functions.

- **Planning CRUD methods:** As other entities, the controller has methods to create, read, update and delete this entity. As happens with the scout association entity, two servlets respond to *GET* and *POST* requests respectively for the create operation. One of them for providing the form and the other for creating the entity and storing it in the database if it is valid. For the rest of the operations, the controller contains one servlet per each one. The read method considers the state of the *Planning*, if the planning is in an initial configuration status or the user has solved it yet. In the last case, the servlet loads the view for the solved planning and its list of *Unitassignments*.
- **Scout leader's operations:** Each planning has scout leaders, so the controller includes the scout leader's CRUD methods. The life cycle of a scout leader has some critical considerations.
  - **Create scout leader:** The organizer of the association creates *ScoutLeader* entities, one per scout leader who participates in the planning. The organizer completes the name and email for each scout leader. This method will create a *ScoutLeader* object with this data and store it in the database.

  - **Read scout leader:** As the view of the planning contains all the participant scout leaders, the controller does not need to implement this function.
  - **Update scout leader:** The controller includes three types of update methods:
    * **Modify data:** the application considers a method to modify the scout leader name and email that the organizer executes if he needs to update these data.
    * **Fill details:** Initially, each scout leader completes its details, but the organizer can fulfill or edit them too. Some of these details are: gender, age, experience, camp attendance, Etc. Moreover, another servlet provides the form to allow the scout leaders to fulfill their details. The controller implements an additional servlet to modify this data after fulfilling it. If the organizer fills the scout leader's details, he can perform the operation directly by a modal embedded in the planning view.
  - **Delete scout leader:** The controller implemens a servlet to delete a scout leader from a specific planning.

- **Email sending:** The application contains a servlet in charge of sending emails when the organizer requests, to allow scout leaders to introduce its details. The server has a template that contains the gaps specifically for the scout leader's information and the planning data. The servlet fulfills this information and sends the emails to the scout leaders' addresses. The application uses the module *StringTemplate* [24] for the generation of dynamic templates and its fulfillment.

Listing 5.23: Email sending to each scout leader

```
List<Monitor> monitores = dataRepositoryService.
    readMonitoresFromPlanningSimple(parrilla);
ServletContext context = req.getServletContext();
String pathToEmailTemplate = context.getRealPath("/WEB-INF/views/jsp/
    emailTemplate.stg");
STGroup mailTemplateGroup = new STGroupFile(pathToEmailTemplate, '$', '$');
for(Monitor monitor: monitores) {
    ST mailTemplate = mailTemplateGroup.getInstanceOf("monitorFillTemplate
        ");
    mailTemplate.add("parrilla_name", parrilla.getNombre());
    mailTemplate.add("monitor_name", monitor.getNombre());
    String monitor_url = "http://localhost:8080/TFM/monitor/" + monitor.
        getId() + "/fill";
    mailTemplate.add("monitor_url", monitor_url);
```

```
    String mailTemplateRendered = mailTemplate.render();
    String monitor_email = monitor.getEmail();
    String mailSubject = "plan.x: " + parrilla.getNombre() + " completar
        datos de monitor";
    GenericUtils.sendMail(monitor_email, mailSubject, mailTemplateRendered)
        ;
}
```

- **Planning solving:** When all the scout leaders have introduced their details, the organizer can solve the planning. The controller contains a servlet that, firstly, imports the unit list of this scout association from the database and the scout leaders' list with their details. Secondly, the servlet adds these lists to the planning object and finally uses the intelligent system's solver to find a solution. When the solver finishes this operation, the servlet stores the assignments in the database, updates the planning status to *SOLVED* and finally, redirects the user to visualize the results.

- **Export planning:** Finally, with the planning solved, the organizer can export the results in different formats: HTML, PDF, and JSON. Each format has associated a different *GET* servlet that generates the file with the planning information. When the organizer requests to export the results in HTML, the file that obtains contains a table with the units and scout leaders assigned. The generation of this file is similar to the behavior for the creation of the email template explained before using also, in this case, the *StringTemplate* module [24]. The application first creates the HTML code that then converts to a PDF file to generate the PDF file, for this transformation the applicaiton uses the *html2pdf* converter module [18]. For the JSON format, the servlet generates a structured JSON [17] code with the *UnitAssignment* and the information about the units and scout leaders of each assignment.

5.3.2.3 Views

The application does not have any huge amount of views, because the information managed does not require to elaborate them. The views are split into components by using Apache Tiles [9], so that components can be reused to produce similar views.

The application contains two basic templates used to generate the rest of views. One of the templates covers the log in and sign up pages and a more complex one the views supplied when the user is authenticated.

Login page

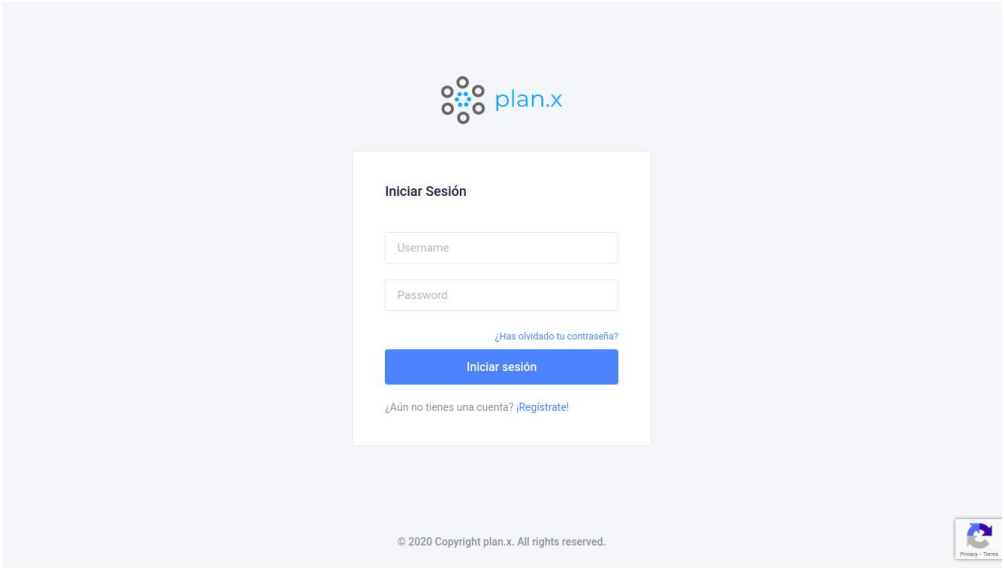

Figure 5.13: Log in view

The login page contains a basic form to introduce the username and password of the user. For the application, the username is the user's email address that he submit when registering in the application. The page is the first one shown if the session does not contain a logged-in user. To use the application, users must log in using the login form. If a user tries to access some URI that needs to log in, the application redirects him to this page.

Figure 5.13 shows in the lower-right corner the icon of reCAPTCHA [14], a Google service that protects from spam and abuse a web site. The service calculates a score that determines whether a person or a machine uses the application. This protects websites from undesirable bots that can be malicious. Also, the service avoids brute force attacks in the login form, preventing the form's sending.

Sign up page:

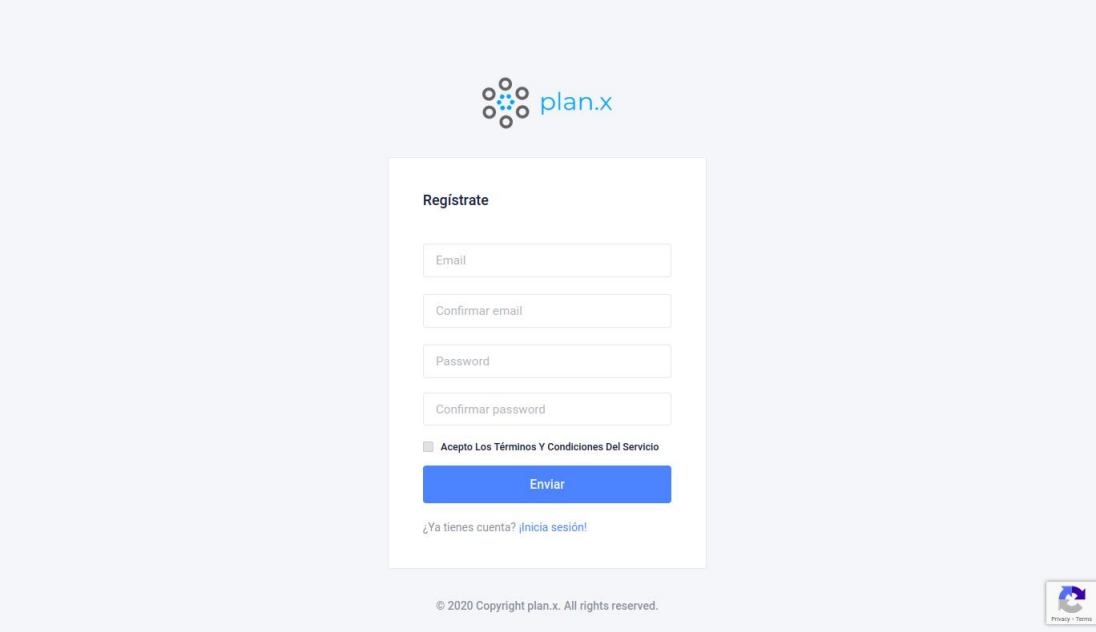

Figure 5.14: Sign up view

The signup page is similar to the login page. This page contains a form where the user specifies his email address and the password that he wants to use for the application. The user should indicate this information twice in the corresponding fields to check that the information is correct. Furthermore, when registering, the user has to accept the Terms and Conditions of the application to submit the form.

To prevent bots from creating a vast amount of users, this form also includes the re-CAPTCHA [14] service. The login form and the signup form are the only ones that include this tool to ensure that physical people use the application.

Create Scout Association page:

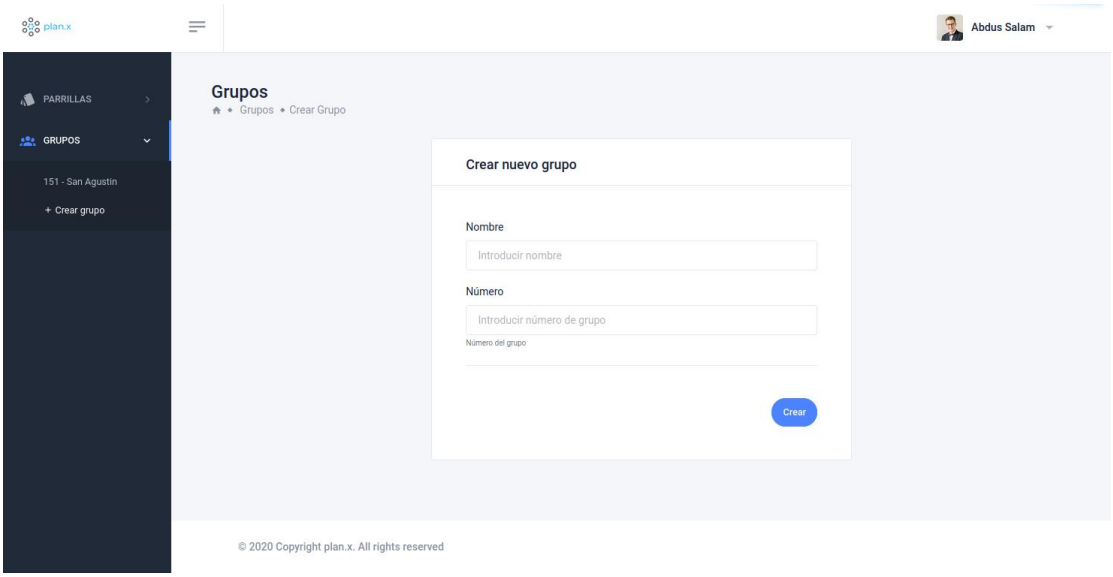

Figure 5.15: Create Scout Association view

The Create Scout Association view contains a form where the user configures the first details of the scout association. In particular, these information is the name and number of the scout association.

The Create Planing page is precisely the same but with other fields in the form, the ones required to complete the planning's necessary details.

Scout Association page:

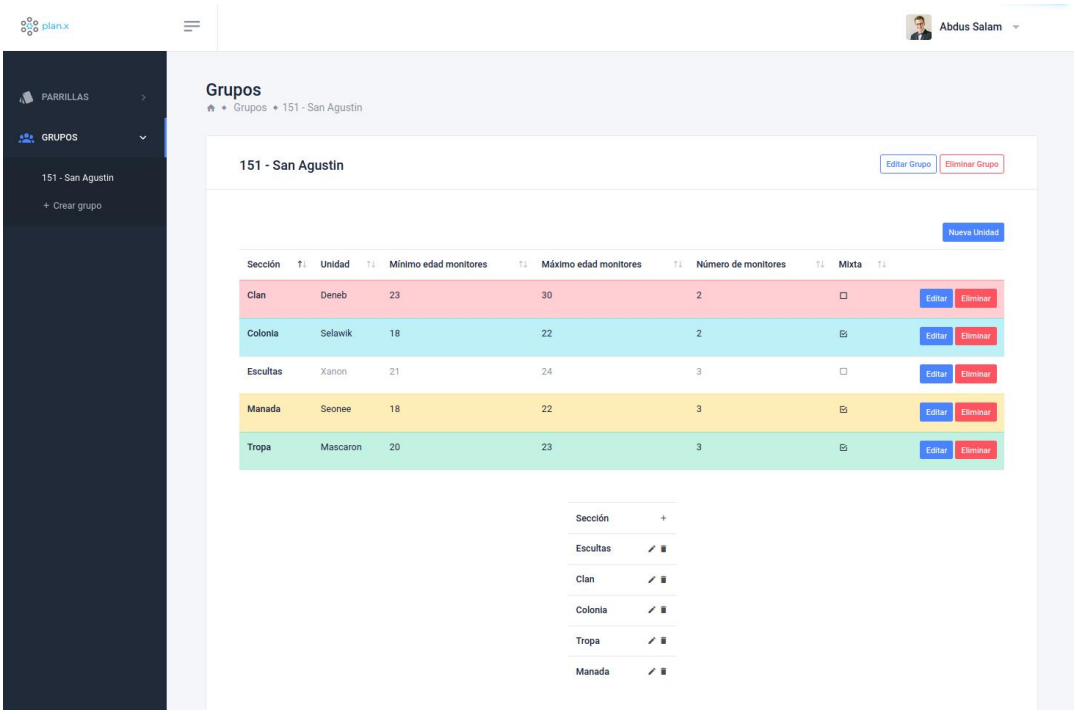

Figure 5.16: Scout Association view

This view contains all the details of a scout association. Here the user can create, edit, and delete all the sections, units, and even the scout association. Figure 5.16 shows a scout association that contains all its details fulfilled. The first table contains the list of units and all their details. The second one lists all the sections of a scout association. When a user configures a section, he has to define the color of the section. The colors highlight units of the same sections so that users can quickly identify all the section units.

To perform operations over each component of a scout association, the user can click on the view's corresponding buttons. Each button displays a modal with a form like the one shown in Figure 5.17.

Edit Unit in Scout Association page:

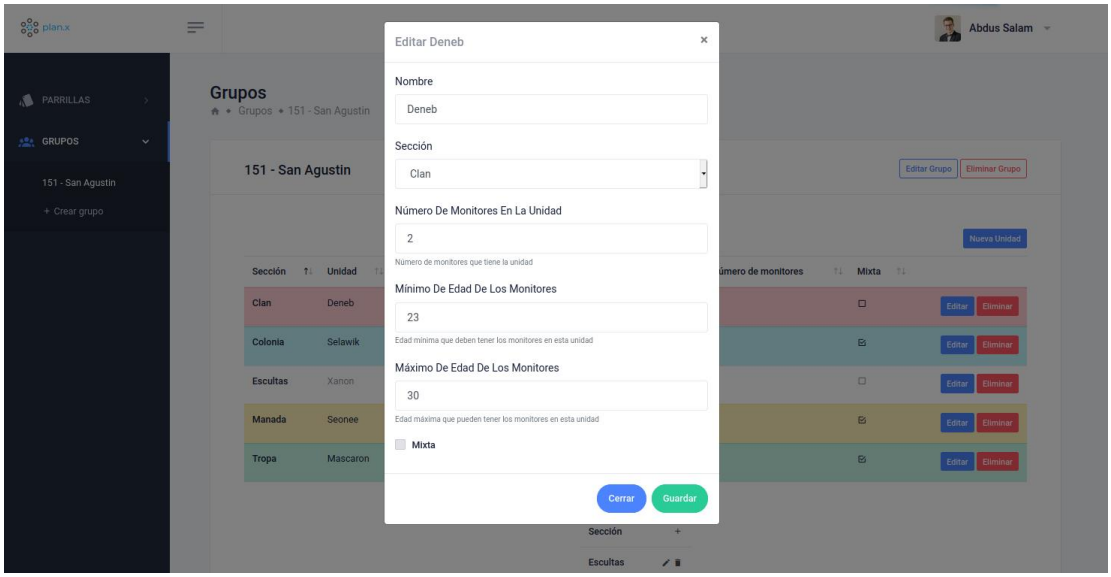

Figure 5.17: Edit Unit view

Depending on the function to perform, the form of the modal contains some fields or others. The edit fields contain the item's actual data to modify, such as in Figure 5.17, which shows the edit form for a unit. The creation form of an item is the same as the edit form but with all the fields blank to allow the user to introduce the data that each item requires. Moreover, the delete modal does not contain a form like the previous ones exposed but asks the user to confirm the action.

The application uses modals to carry out the actions to create, edit, and delete all the items that the system manages. The only exception is with the creation of a scout association and with the creation of a planning.

Planning page:

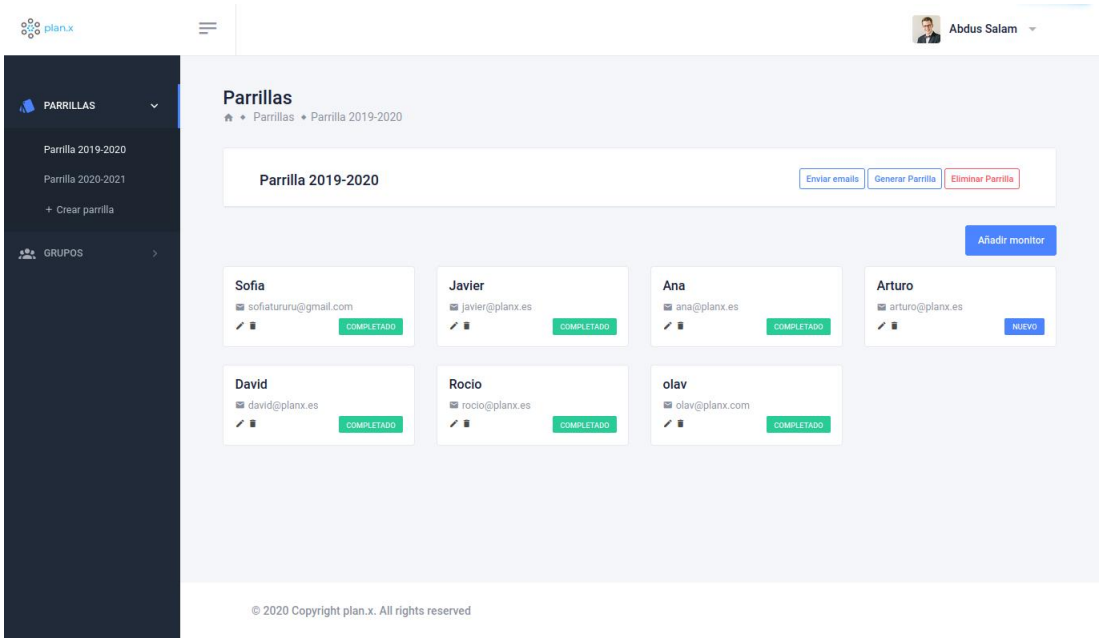

Figure 5.18: Planning view

The planning page has two views, one for solved plannings and the other for the ones in the configuration phase. Figure 5.18 shows the view of a non solved planning. This view includes the planning details that are the scout association and the scout leaders that participate in the planning. Like in the Scout Association page, the page contains buttons to display the modals to perform actions over the planning and scout leaders. The view displays two additional buttons: one to send emails to the scout leaders to ask them to complete their details and the other to ask the system to generate the planning.

Another important fact is that when a scout leader submits its details, the application updates the state to **completed**. The view presents each scout leader's state so that the organizer can see who has fulfilled the form or not. This view allows users to edit each scout leader's details, the name and the email address, and the details that the scout leader submitted.

Scout leader details form:

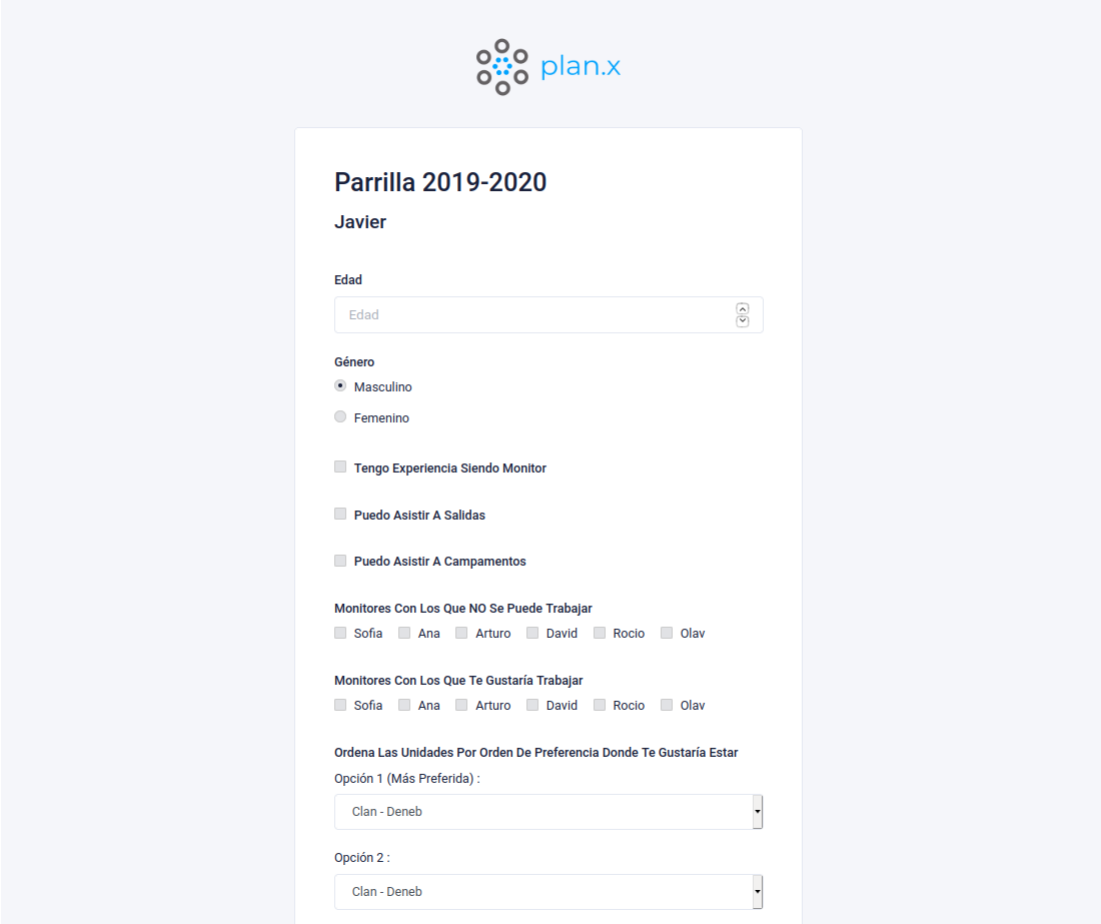

Figure 5.19: Scout leader details form view

This view represents the form that scout leaders receive by email when the organizer requests them to complete their details. As Figure 5.19 shows, the form contains all the specific fields required to build a planning. The form fields are mainly checkboxes, radio buttons, and option selection fields, not requiring the scout leaders to write extensive texts. The only exception is with the age field, which requires the scout leader to introduce a number. This is important because it speeds up the fulfillment of the form, recovering all the required data. When the scout leader has introduced all the details, he presses the submit button contained at the end of the form.

Solved planning page:

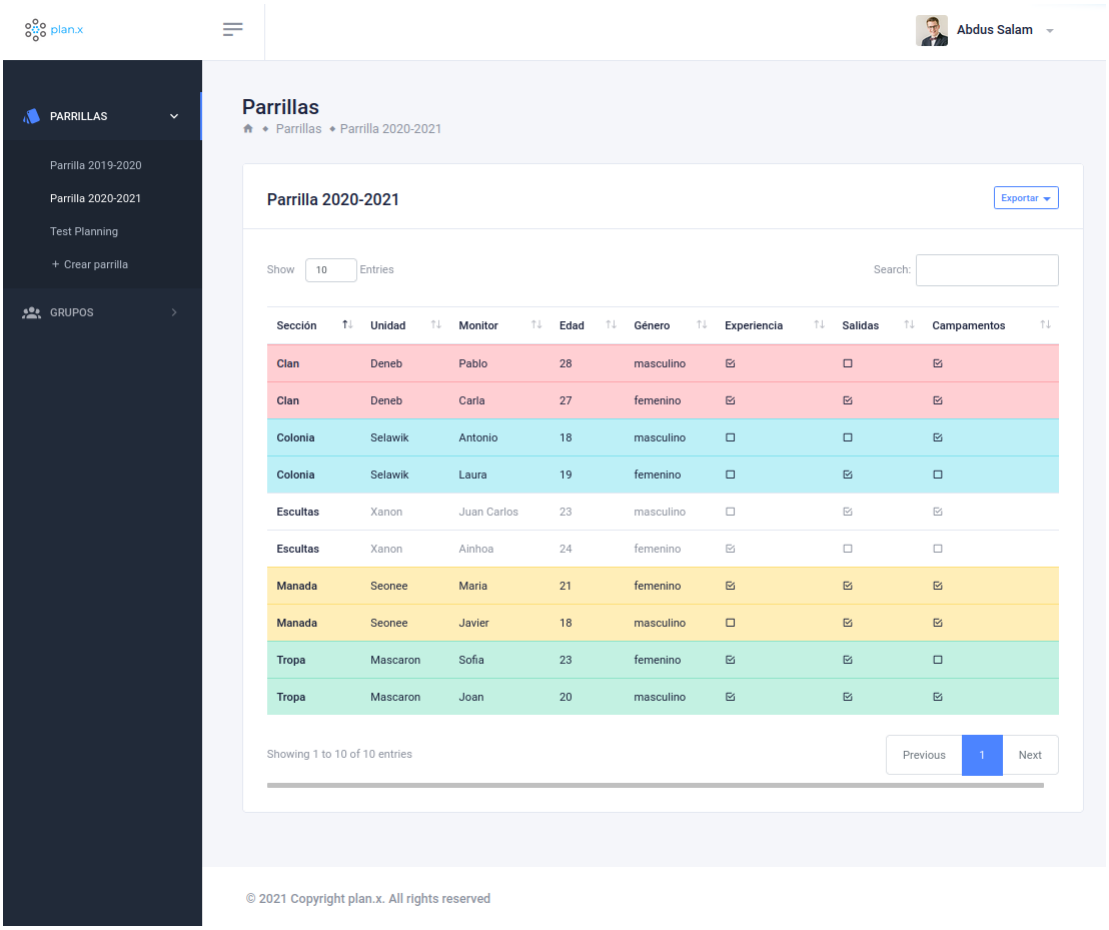

Figure 5.20: Solved planning view

When the intelligent system has finished solving the planning, the application presents the results. The organizer presses in the planning's name to access them, like when configuring the planning. The application checks the state of the planning, and if the planning is solved, loads the solved planning view shown in Figure 5.20.

Figure 5.20 contains a table with all the scout leaders and the unit assigned. Furthermore, the view includes some of the scout leader's details to allow the organizer to check visually if units comply with all the requirements. Like in the Scout Association page, the table contains each unit's sections' colors to ease the organizer with the visualization. Furthermore, in the table's upper right corner, the view shows the button for exporting the planning results into the formats explained.

Error page:

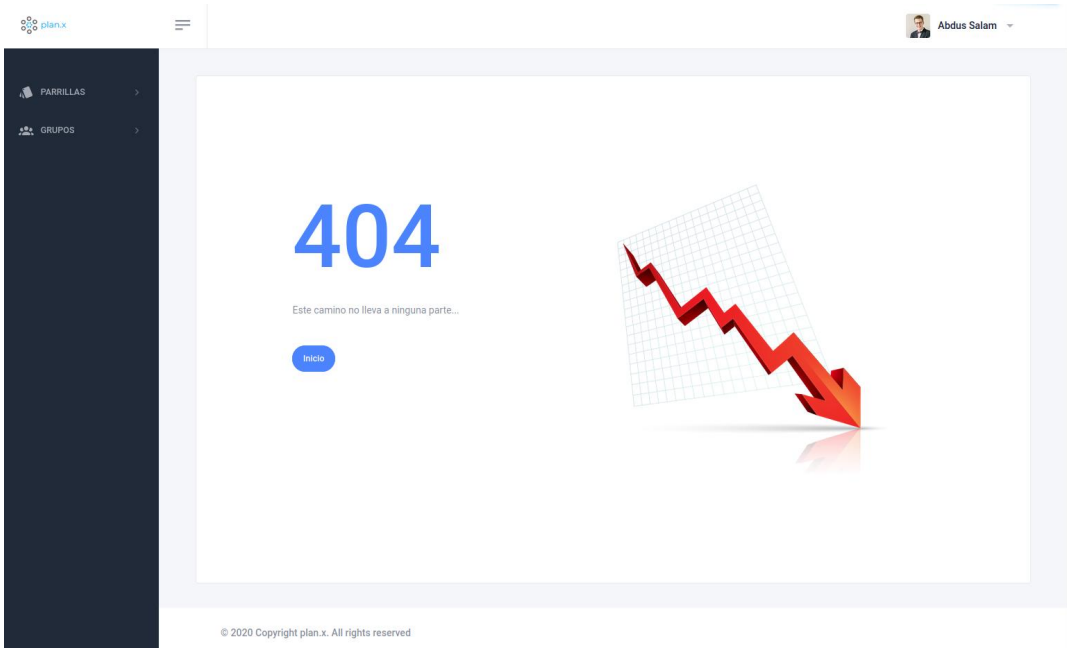

Figure 5.21: Error view

The application redirects users to the error page when they try to perform actions that the application does not contemplate. For instance, when a user tries to access a URI different from the ones configured.

5.3.3 Authentication and authorization

The application uses the Spring Security framework [48] to easily manage authentication and authorization without having to develop any additional system. The authentication requires to implement three classes containing some special methods.

Configuration

Spring Security framework [48] requires creating a new configuration file (`WebSecurityConfig`) related exclusively to this module's configuration. This class has to extend the abstract class `WebSecurityConfigurerAdapter` [49] to override the methods of Spring Security for authentication and authorization. Some aspects that should be configured here are the password encoder used, the URIs that require authentication, and which user roles can access.

For the project, the password encoder chosen is `BCryptPasswordEncoder` [53] which has some advantages over other encoders, such as the generation of random *salts* for every password. The output of the encoder is a String with the password hash and the *salt*. However, the encoder provides different hashes even if the password is the same. An example of this algorithm's output is shown here. For this case, the password encoded is *1234*.

Listing 5.24: Example of password encoded using BCrypt encoder

```
$2a$10$4NmOhfTSJ66LRnaXJAJNt.iRK9LQs/XWchk8blq3kG258fLQZzKB.
```

Another essential subject to configure here is the set of URIs that require login to access and the redirection to the login form instead. Nevertheless, the framework requires to define the opposite, the URIs that do not require authentication. Moreover, the module enables to describe the login form to use if the application contains a customized one. This project contains the login form is under the */login* URI. Finally, the configuration should define the site to redirect after the user logs out, in this case to the login form.

Listing 5.25: Security configuration

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/", "/signup", "/assets/**",
                  "/monitor/*/fill", "/monitor/*/fill/edit")
                    .permitAll()
```

```
            .anyRequest()
            .authenticated()
            .and()
          .formLogin()
            .loginPage("/login")
            .permitAll().defaultSuccessUrl("/index",false)
            .and()
          .logout()
              .logoutSuccessUrl("/login")
              .permitAll();
}
```

Furthermore, Spring Security should know how to authenticate users. The module provides multiple options to store users, such as keeping them in memory or using a database. For this project, the application stores user information in the MySQL database, so Spring Security's configuration to retrieve users is the following.

Listing 5.26: Code used to specify how to retrieve users from the storage

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws
    Exception {
    auth.userDetailsService(userDetailsService).passwordEncoder(
        passwordEncoder());
}
```

The code above specifies to Spring Security where the User Details Service is defined and the password encoder to use, in this case, the encoder described before.

User Details Service

The *UserDetailsService* [52] is an interface whose implementation should include the method `loadUserByUsername(final String username)`. The project stores users in the database, so the application should retrieve users from there. The method searches for a user by the given user name and returns the user details if it is found.

Listing 5.27: Code used to look for users in the storage

```
@Transactional(readOnly=true)
@Override
public UserDetails loadUserByUsername( final String username)
    throws UsernameNotFoundException {
```

```
  User user = dataRepositoryService.findByUserName(username);
  if (user != null) {
      return user;
    }
  throw new UsernameNotFoundException("User '" + username + "' not found");
}
```

Spring Security uses the user details object to perform the comparisons by the username and password provided in the login form. If the result of the comparisons is valid, the username is stored in the session.

User Details

Finally, the entity user has to implement the interface `UserDetails` [51] which obliges to develop methods relating to user description. Spring Security calls these methods to know the user's username, password, roles, and state.

The implementation of these methods in the project is straightforward. They only return the values of the user object's attributes, so they could be considered *getters*. However, the *User* class should contain a special method: `getAuthorities` that should return a collection with the user's roles.

Listing 5.28: Code used to return the list of roles of a user

```
@Override
public List<GrantedAuthority> getAuthorities() {
  List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
  authorities.add(new SimpleGrantedAuthority(role));
  return authorities;
}
```

The project considers just two roles: *USER* and *ADMIN*. The only difference is that an *ADMIN* user can manage other users, but an administrator is also a user. So the implementation considers that users will have one role.

5.3.4 API

The application needs to define a way to allow connections from other systems to perform tasks. Other applications from the scout associations may use this system to generate plannings and directly introduce the data to their systems automatically. The connector for this side of the application is the API defined.

Basically, the API allows users to perform the same functions explained in previous sections but changing the interface. In this case, the application receives data using a structured language (JSON [8]) instead of providing them using the forms shown in Sect. 5.3.2.3. The following code creates a scout association with three sections and one unit per section.

Listing 5.29: JSON code to create a scout association using the API

```
{
    "grupo":
    {
        "numero": 334,
        "nombre": "Proel",
        "secciones": [
        {
            "color": 2,
            "nombre": "Tropa",
            "unidades": [
            {
                "nombre": "Mascaron",
                "edad_inicial": 20,
                "edad_final": 25,
                "numero_monitores": 3,
                "mixta": true
            }]
        },
        {
            "color": 0,
            "nombre": "Colonia",
            "unidades": [
            {
                "nombre": "Destellos",
                "edad_inicial": 18,
                "edad_final": 23,
                "numero_monitores": 3,
                "mixta": true
            }]
        }]
    }
}
```

To allow a user to perform operations using the API, he has assigned an API key that he has to provide within each request. Moreover, the request should contain the code with

the operation data to perform in a variable called *data*. Furthermore, the URI present in the request determines the operation to carry on. For example, if using *curl* to perform operations, the request to create the scout association shown previously should have the following form.

Listing 5.30: Example of curl command to use the API

```
curl -d "apikey=134aebf4-1e6e-4439-aa25-3c5397a9b180" -d 'data=JSON_CODE' -
    X POST http://localhost:8080/TFM/api/grupo
```

Where *JSON_CODE* contains the JSON code with the scout association's data in this case.

Before doing any actions, the application checks if the request contains a valid *apikey*. All the system associates all the operations requested to the API with the user owner of the *apikey* provided. The responsible for performing the functions of the API is the *Api-Controller*, that implements the necessary servlets. The functions that a user can perform by using the API are the followings:

- **Create and update scout association:** Unlike when configuring the scout association with the user interface, this time, the *POST* request has to contain all the details of the scout association, the sections, and the units. The application checks whether each element of the JSON code of the request (scout association, section, or unit) contains the *id* attribute. If the elements contain the *id* attribute with a valid identification number, the system updates the elements and creates new ones if the attribute is not present. Before carrying on any action, the servlet checks the validity of the data received. As a result, if the operations are correct, the servlet returns the *id* of the scout association created or updated.

- **Create and update planning:** Similar to the previous case, users can create and update plannings by providing the required data. In this case, the name of the planning, *id* of the scout association and the scout leaders' data: name and email. Like in the previous case, if the elements contain a valid *id* attribute, instead of creating a new one, the application updates the element's information. Newly, before performing any action, the servlet checks the validity of the data of the *POST* request. The servlet returns the *id* of the planning created or updated if all the operations are correct.

- **Send emails to scout leaders:** The application can receive a *GET* request to send the emails to scout leaders to ask them to complete their details. This request does

not need to include any additional data under the *data* variable because the URI contains the *id* of the planning with the scout leader data implicitly.

- **Solve planning:** When the scout leaders have completed their details, the user can ask to solve the planning. The *GET* request associated with this case is similar to the one of sending emails and does not require to submit any additional data under the *data* variable. When the application finishes generating the planning, the servlet sends *OK* to the client that requested the operation.
- **Read planning:** Finally, clients can request the data of a planning by sending a *GET* request to the API that will return the data using JSON code. The code returned contains the *UnitAssignment* of a planning with the unit's information and the scout leader. This request does not require the *data* variable.

As detailed previously, the create/update requests send to the API are *POST* requests. The system needs to have disabled the CSRF control to map these requests with the corresponding servlet, because this type of control only applies when using cookies. While the web interface uses sessions and cookies, the API does not require them to work, so this type of control is unnecessary. To disable the CSRF control for the API request, the `configure` method seen on Sect. 5.3.3 has to include the following code.

Listing 5.31: Code to disable CSRF control over API requests

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    ...

    http.csrf().ignoringAntMatchers("/api/**");
}
```

5.4 Persistence

5.4.1 MySQL

A MySQL system [54] uses the Client-Server architecture like many other applications. The server stores data and the clients access the data connecting to the server. A server contains one or more databases and multiple users that can access databases. To perform actions on a server's databases, clients need to have valid credentials of a user with permissions in the database they would like to act.

Commonly an application contains one database with multiple tables and one user with permissions to operate in the database. In a simple development, the server that hosts the application corresponds to a client that connects to the MySQL server. For broader applications, the number of clients can be higher as the number of application servers grows.

As mentioned before, each database contains multiple tables with columns and rows. Columns indicate the data structure that the table stores and rows contain the data to save. While MySQL is a relational database, each row must have the values subject to the columns that the table defines. This means that rows cannot have more or fewer fields than the ones of the columns. Additionally, each row must have a unique key that identifies its data properly. Besides, tables can have relations with others through foreign keys.

5.4.2 Application to the project

The application needs to implement a persistence system to store the managed data. When the application stops, the data that the application was managing should be able in the next execution. The project uses a MySQL server that stores the structured information of the application.

The MySQL server runs on the same machine as the server that hosts the application. The application employs Java as the primary language, and Java uses object-oriented programming, so the MySQL server should store objects. To do so, each of the entities seen on Sect. 5.3.2.1 contains a table that saves their attributes' values. An entity corresponds to a table, and an attribute of the entity corresponds to a column in the table. Finally, an object of an entity matches a row in the table.

A class's attributes may require particular storage actions if they are not simple types of data. For instance, if an attribute is an object from another entity, the database tables should contain a relation between them. The relations can be 1:1 1:N or N:M, depending

on the number of objects a specific attribute of an entity can have. Figure 5.22 contains the database scheme with each table and the relations for the application of the project.

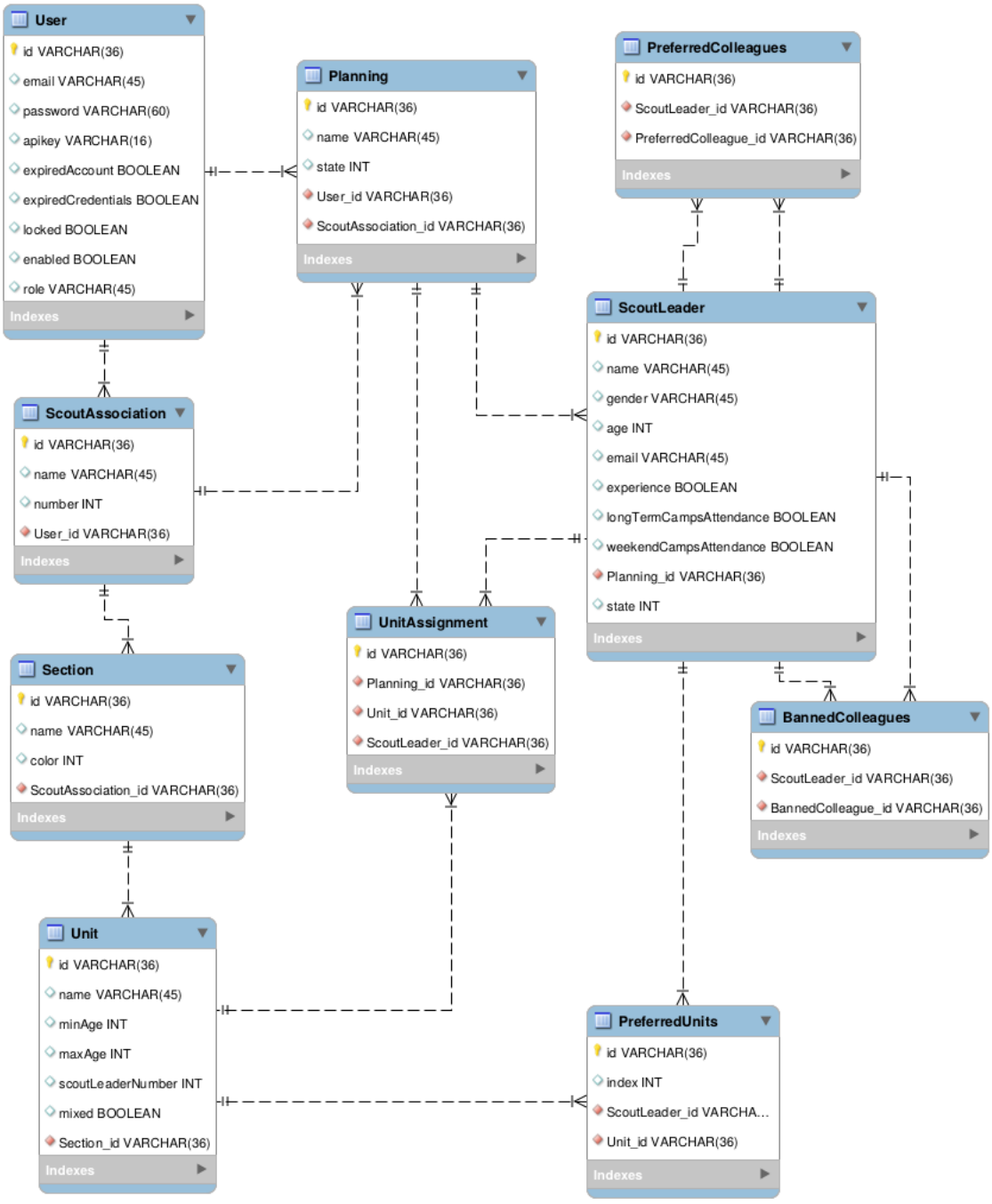

Figure 5.22: Database scheme diagram

The figure contains ten tables in comparison with the seven entities that the application manages. This is because the *ScoutLeader* entity contains three collections of objects: `preferredUnits`, `preferredColleagues` and `bannedColleagues`. Each of the col-

lections has an associated table with the same name that stores the id of the *ScoutLeader* object and each of the ids of the entities of each lists. So when the application saves a *ScoutLeader* object, the persistence system updates four tables.

As the figure shows, each of the entities contains a table with the values of their attributes. One important attribute is the id, where the application uses unique identifiers of type UUID generated randomly. The database does not create this ids like when they are integers auto-incremented, but generates an exception if a client asks to store an item with the same id of another in the same table. The application is responsible for generating the ids and catching the exceptions of the database system.

Most of the relations Figure 5.22 contains are of the type *belongs to*. All of them are 1:N[3] relations.

- A *ScoutAssociation* belongs to a *User* and a *User* can have N *ScoutAssociation*.
- A *Section* belongs to a *ScoutAssociation* and a *ScoutAssociation* can have N *Section*.
- A *Unit* belongs to a *Section* and a *Section* can have N *Unit*.
- A *UnitAssignment* belongs to a *Unit* and a *Unit* can have N *UnitAssignment*.
- A *Planning* belongs to a *User* and a *User* can have N *Planning*.
- A *Planning* belongs to a *ScoutAssociation* and a *ScoutAssociation* can have N *Planning*.
- A *UnitAssignment* belongs to a *Planning* and a *Planning* can have N *UnitAssignment*.
- A *ScoutLeader* belongs to a *Planning* and a *Planning* can have N *ScoutLeader*.
- A *UnitAssignment* belongs to a *ScoutLeader* and a *ScoutLeader* can have N *UnitAssignment*.

Also, the collections mentioned previously apply relations more complex than the previous ones. The tables with the same names as the collections implement M:N[4]. relations, that are the following:

- A *ScoutLeader* can have N *ScoutLeader* and a *ScoutLeader* can have M *ScoutLeader*, both by the *preferredColleagues* relation.

[3]Here N represents multiplicity, not an specific value
[4]Here M represents multiplicity, not an specific value

- A *ScoutLeader* can have N *ScoutLeader* and a *ScoutLeader* can have M *ScoutLeader*, both by the *bannedColleagues* relation.
- A *ScoutLeader* can have N *Units* and a *Unit* can have M *ScoutLeader*, both by the *preferredUnits* relation.

CHAPTER 6

Case study

This chapter presents a case study that contains a problem introduced to the application and executed to find a possible solution. The first section provides the statement, then the application receives the configuration of the problem to solve, and finally, the last section covers the analysis of the results that the system supplies.

6.1 Problem

Similar to the example seen in Chapter 3, this chapter defines a problem, but in this case, the application solves it directly. The difficulties to solve the problem manually considering all the variables and constraints would require lots of hours or even days, but the application gives a solution in a minute. Check a solution is really easy, just by examining that the solution provided complies with all the requirements.

For the example, the scout association has the five sections of reference, seen in Chapter 3: Beavers (B), Cubs (C), Scouts (S), Explorers (E), and Rovers (R). Each section has just one unit, and the scout association has ninety kids distributed in the units shown in Table 6.1. The units have the following requirements for the scout leaders, based on the children's ages and characteristics.

Unit	**Children's age range**	**Number of kids**	**Team size**	**Scout leaders' age range**	**Mixed team**
Beavers	6-7 y.o.	15	3	18-23 y.o.	True
Cubs	8-10 y.o.	25	4	18-23 y.o.	True
Scouts	11-13 y.o.	20	3	20-24 y.o.	True
Explorers	14-16 y.o.	17	3	22-25 y.o.	True
Rovers	17-20 y.o.	13	2	23-27 y.o.	False

Table 6.1: Unit characteristics in the case study

As shown in Table 6.1, the scout leaders team size, and the number of kids are positively correlated. Furthermore, the total number of scout leaders is equal to the sum of all the scout leaders' teams' sizes. Moreover, all the units require scout leaders from both genders except the Rovers section because the children here are between seventeen and twenty years old. The children's maturity at these ages does not require both a man and a woman as a referent figure.

Table 6.2 shows the fifteen members' complete set of characteristics that compose the scout leaders team. The table contains some abbreviations to include all the details in the same space. Like in Chapter 3, the availability is shown in terms of: WC meaning weekend camps attendance and C meaning camps attendance. Also, the scout leaders have a reference Ai where A is a letter, and i is a number that both the banned and preferred colleagues columns use. Finally, the preferred units column shows each unit's initial letter

ordered from left to right, being the one on the left the more preferred and the one right the least preferred one.

Id	Name	Gender	Age	Exp.	Avl.	Banned people	Preferred people	Preferred units
M1	Miguel	Male	18	No	WC, C	-	J1,J2	B,C,S,E,R
M2	Mario	Male	19	Yes	WC, C	-	N	C,B,S,E,R
S1	Sofía	Female	18	No	-	-	M1	C,B,S,E,R
M3	María	Female	22	Yes	WC, C	J1	J3	B,C,S,E,R
J1	Juan	Male	21	No	C	M3	A1,J2	C,B,S,E,R
M4	Manolo	Male	27	Yes	WC	-	-	R,E,S,C,B
L	Luis	Male	24	Yes	WC, C	-	-	E,R,S,C,B
J2	Jorge	Male	19	No	WC, C	A1	M1	B,C,S,E,R
A1	Aitana	Female	20	No	WC	-	-	C,B,S,E,R
S2	Sara	Female	23	Yes	WC, C	-	N	S,B,C,E,R
M5	Marta	Female	25	Yes	WC	S2	-	E,R,S,C,B
N	Natalia	Female	22	Yes	-	-	-	S,C,B,E,R
J3	Joan	Male	23	Yes	WC, C	-	L	S,E,C,B,R
A2	Ana	Female	26	Yes	C	M5	M4	R,E,S,C,B
J4	Jaime	Male	24	Yes	WC, C	-	S2,M5	E,R,S,C,B

Table 6.2: Scout leaders characteristics for the case study

The number of scout leaders and their characteristics make really complicated to find a solution to the problem. The number of dimensions to consider is vast, but the application developed will produce the optimal solution. This solution will try a vast number of combinations until finding the best result, attending to the constraints explained in section 5.2.6.

6.2 Application configuration

The first step to solve the problem is to introduce the problem specific characteristics in the corresponding options. At first, the application needs the user to detail the scout association parameters, that are the ones of Figure 6.1.

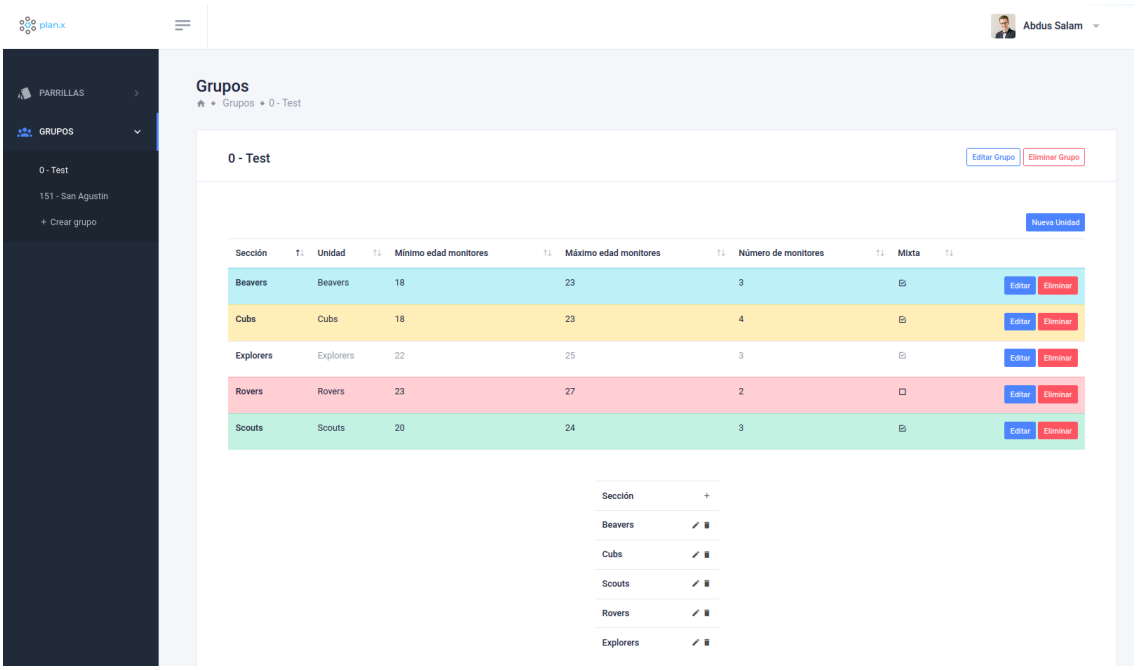

Figure 6.1: Scout association for the case study

Figure 6.1 contains the scout association defined in the previous section introduced in the application. For the example, the scout association name selected is *Test* and the number of the scout association is *0*. Besides, sections and units have the same names because the association has only one unit per section.

After configuring the scout association details, the next step is to introduce the details of the planning. Similar to the scout association, the name for this planning is *Test Planning*. The application receives the names and the emails of all the scout leaders of Figure 6.2. This time, the organizer has to introduce the details of each scout leader manually. This behavior is not the one users will have to face when using the application because each scout leader will have to introduce his own details. However, in some cases, the organizer would need to modify some specific data relating to scout leaders through the planning view.

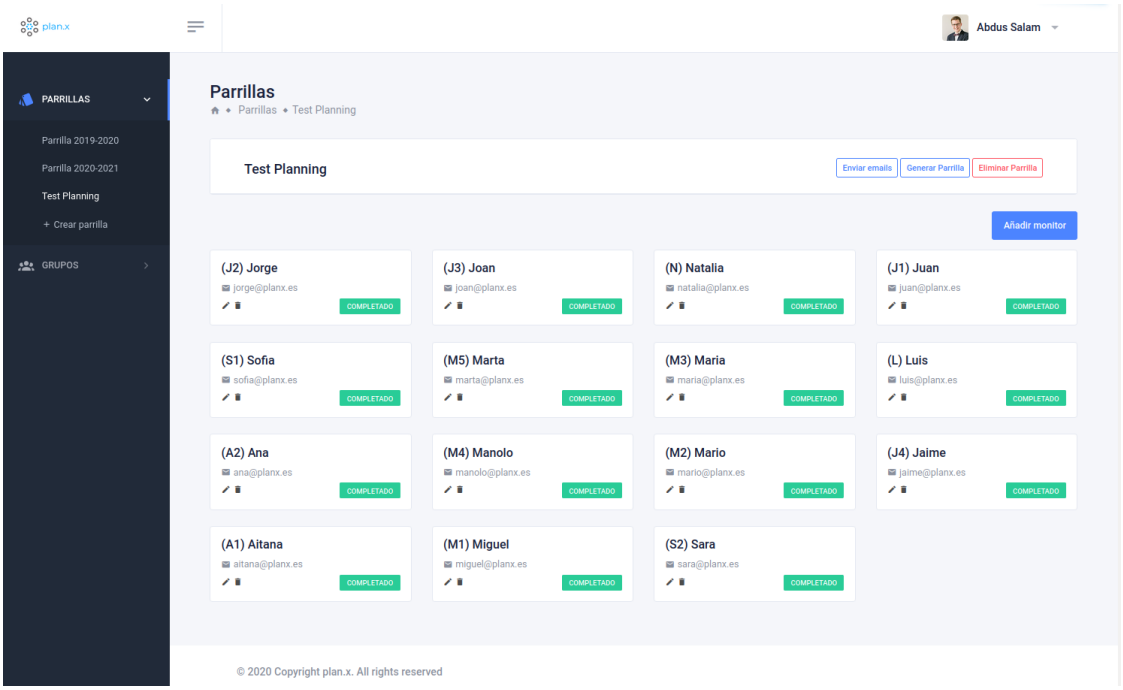

Figure 6.2: Example planning configuration

6.3 Solution

After performing the scout association and the planning configurations, the application can generate the solution. To do so, the organizer should press the *Generate planning* button, the application starts executing and returns the following result. Figure 6.3 contains the list of scout leaders and the units assigned after one minute of execution.

The view allows the organizer to rapidly check the compliance of some requirements, such as if units contain scout leaders from both genders, the presence of people with experience in the units, or attendance requirements to the scout association's activities during the year. However, the page does not include the preferred and banned colleagues and the ordered list of preferred units to allow some privacy over scout leaders' preferences.

The score of the optimal solution presented that the system gives through the debug console is *0 hard / 0 medium / 32 soft*. This means:

- The system has respected all the defined **Hard Constraints**:
  - The planning does not contain repeated scout leaders.
  - The planning respects the number of scout leaders defined per unit.

  - The planning respects the age range in each unit.
  - The planning does not contain banned colleagues in the same unit for any scout leader.

- The system has covered all the defined **Medium Constraints**:
  - The planning respects mixed units.
  - The planning considers the availability of scout leaders in weekend camps attendance in each unit.
  - The planning considers the availability of scout leaders in camps attendance in each unit.
  - The planning includes scout leaders with experience in each unit.

- The system has taken into account some of the preferences of the scout leaders relating to **Soft Constraints**:
  - The planning tries to create combinations taking into account the preferred colleagues of some scout leaders.
  - The planning tries to respect the order of preferred units for some scout leaders.

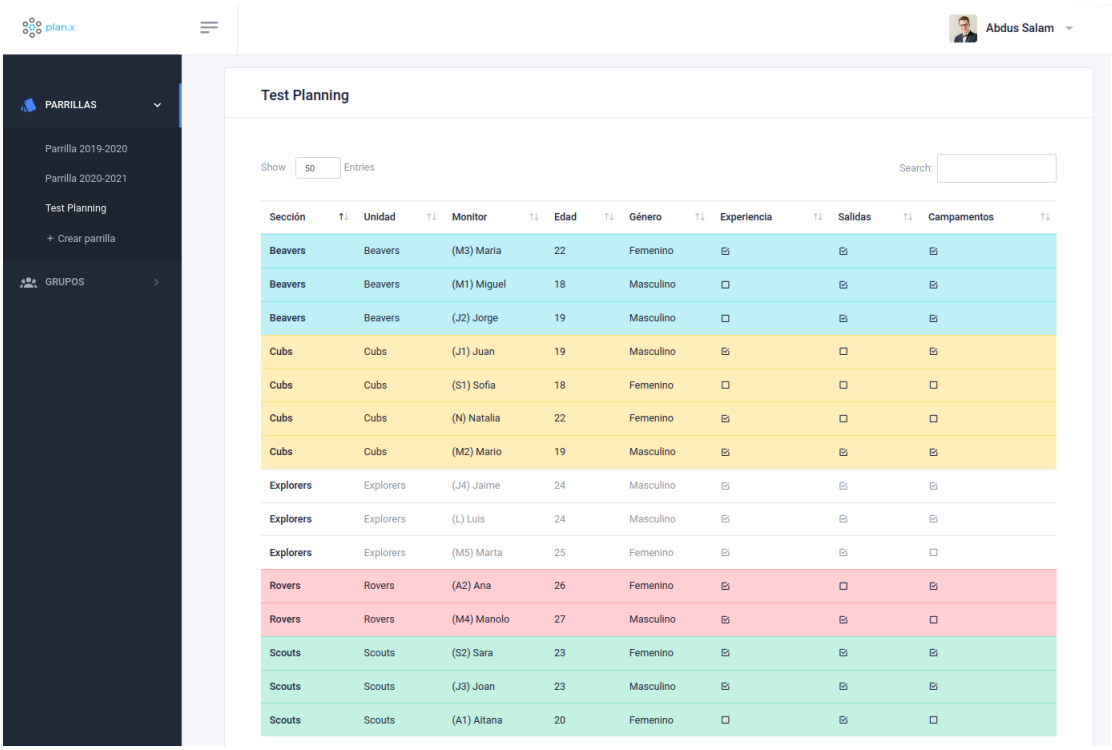

Figure 6.3: Case study solved planning

Table 6.3 lists the assignations and some of the basic checks related to requirements that a planning should have.

Unit	Scout lead.	Size	Mixed team	Age	Exp.	WC avl.	C avl.
Beavers	(M1) Miguel (M3) Maria (J2) Jorge	OK *3/3*	OK *2 male, 1 female*	OK *18, 22, 19*	OK *1 true*	OK *3 true*	OK *3 true*
Cubs	(J1) Juan (S1) Sofia (N) Natalia (M2) Mario	OK *4/4*	OK *2 male, 2 female*	OK *19, 18, 22, 19*	OK *3 true*	OK *1 true*	OK *2 true*
Scouts	(S2) Sara (J3) Joan (A1) Aitana	OK *3/3*	OK *1 male, 2 female*	OK *23, 23, 20*	OK *2 true*	OK *3 true*	OK *2 true*
Explorers	(J4) Jaime (L) Luis (M5) Marta	OK *3/3*	OK *2 male, 1 female*	OK *24, 24, 25*	OK *3 true*	OK *3 true*	OK *2 true*
Rovers	(A2) Ana (M4) Manolo	OK *2/2*	OK *1 male, 1 female*	OK *26, 27*	OK *2 true*	OK *1 true*	OK *1 true*

Table 6.3: Case study solution requirement check (1/2)
WC avl.: Weekend camps availability, C avl.: Camps availability

As the table shows, all the units comply with the requirements of size, mixed scout leaders' team, age, experience, and availability for the scout association's activities. Furthermore, while the *Rovers* unit does not require to be a mixed unit, the application has assigned scout leaders from both genders.

Some details to consider with the planning are that, for instance, the availability for the *Cubs* unit is not so high compared to other units. While the planning contains three units

with a 100% of attendance to weekend camps, the *Cubs* unit has only a 25%. This behavior might be because of the other constraints that Table 6.3 does not show. To understand why the system has taken these decisions, Table 6.4 analyzes the rest of the scout leaders' requirements.

Unit	Scout lead.	Banned people	Preferred people	Preferred units position
Beavers	(M1) Miguel	(-)	OK (J1,J2)	OK (1)
	(M3) Maria	OK (J1)	- (J3)	OK (1)
	(J2) Jorge	OK (A1)	OK (M1)	OK (1)
Cubs	(J1) Juan	OK (M3)	- (A1,J2)	OK (1)
	(S1) Sofia	(-)	- (M1)	OK (1)
	(N) Natalia	(-)	(-)	OK (2)
	(M2) Mario	(-)	OK (N)	OK (1)
Scouts	(S2) Sara	(-)	- (N)	OK (1)
	(J3) Joan	(-)	- (L)	OK (1)
	(A1) Aitana	(-)	(-)	- (3)
Explorers	(J4) Jaime	(-)	OK (S2,M5)	OK (1)
	(L) Luis	(-)	(-)	OK (1)
	(M5) Marta	OK (S2)	(-)	OK (1)
Rovers	(A2) Ana	OK (M5)	OK (M4)	OK (1)
	(M4) Manolo	(-)	(-)	OK (1)

Table 6.4: Case study solution requirement check (2/2)

The table shows the compliance of the planning with the last requirements and preferences of scout leaders. The system has respected all the banned colleagues defined by scout leaders when creating the planning, something that the score has revealed previously. Moreover, the application has tried to assign in the same unit colleagues marked as preferred ones by others, achieving this purpose in five cases from the ten scout leaders that define preferred colleagues. Finally, the system has behaved really good assigning each

scout leader in the unit contained in the first position of his list for the 86.6% of the cases. One scout leader is in the unit content in the second position and another in the third one. The configuration of the constraints defines that assigning scout leaders in units content in the first or second position is the best behavior to have. The third position would be an acceptable performance and assigning in the least positions, lousy functioning.

The case study shows how the system behaves when using data with characteristics similar to the real data that the system will receive. As analyzed before, the system behaves really well for the data introduced. The system prioritizes the most critical constraints and attempts to generate the best suitable planning respecting scout leader preferences to the extent possible. The application tries to guarantee the minimums in some aspects such as experience or attendance to activities and afterward considers the preferences. An important thing to highlight is the dependence of the system's behavior on the input data, which means that the planning would not comply with the minimums in some cases. For instance, if the total number of girls for a planning is less than the number of units and all the units require scout leaders from both genders, the system can not comply with this requirement. So the performance of the system depends mostly on the quality of the data introduced.

CHAPTER 7

Conclusions

This chapter describes the achieved goals done by the master thesis, the conclusions extracted and the thoughts about the future work.

7.1 Achieved Goals

The project has developed a solution to a common problem in scout associations. The process to obtain the solution has covered multiple phases. The achieved goals for this process are the following ones:

- **Study of the problematic:** The first phase consisted of understanding the problematic to extract the common aspects that all the plannings have. An in-depth analysis of scout associations provides the main characteristics that all plannings should have. These details contain the minimum requirements to generalize the problem and establish the constraints that ensure the plannings' minimum requirements. The constraints express the characteristics of the units and scout leaders.

- **Analyze the requirements:** After understanding the problematic correctly, the following step was to perform a requirement analysis to extract the characteristics that the application should have. The requirement analysis contains the use cases and the actors that the application has to consider. The use cases express the functions that the application has to perform, allowing users to consider the application useful for the objectives proposed. The requirement analysis provided the mandatory, desirable, and optional requirements for the development of the system.

- **Design the system's architecture:** The design phase started with the requirements known to elaborate the application's structure, the entities, and the rest of the system's components. The MVC pattern helps with this phase because it structures the functions in three components. The model defines the entities with their characteristics, the views present the information to the users, and the controllers hold all the system's logic. These structures help resolve how to represent the information, how users should interact, and what the application has to do with the information when users select one option. Furthermore, this phase includes the design of the database schema that stores the information between sessions.

- **Implement the application:** When the application's design has provided all the schemes and the architecture, the implementation phase started. The implementation was modular, first implementing the basic entities and the intelligent system and performing unit tests until reaching the desired behavior. The following module developed was the web application that allows people to interact with the intelligent system without requiring technical knowledge.

- **Test the development:** With the full system implemented, the next step was to test the full development with all the modules connected. The activities here were to identify errors and solve them until reaching the solution explained in the master thesis.

- **Analyze the system's results:** This document includes an example of one of the tests performed and an analysis of the results that the application has provided. The example shows the application's performance and the good results that the system provides for this case with synthetic data, very similar to the ones that users will introduce.

7.2 Conclusions

The project result is a web application capable of generating valid plannings of scout leader's teams within some minutes. The development of the application has been the methodology used to fulfill the goals proposed in the project. This section reviews the objectives proposed and whether the project has worked on them or not.

To acquire more in-depth knowledge of intelligent systems based on constraint programming. After investigating this subject, the project uses this type of technology to elaborate the results. For the problem to solve, this type of intelligent systems is the one that fits the best, providing good results without much difficulty once understood the basic concepts. To provide the best results, the application uses constraint programming with the OptaPlanner framework that provides a suite of configuration options to allow users customization for their programs. Users should try multiple options of configuration to obtain the best results for the problems to solve.

To obtain greater ability in problem description and requirements gathering. The development of an application requires understanding deeply and analyzing the problem to solve to extract all the characteristics to prepare for the design phase. A good analysis provides the requirements and all the functionalities that the system needs to be useful. This project has covered this aspect at the time of characterizing the problem and performing the requirement analysis. These aspects have determined the design of the application and the components to implement.

To apply functional programming techniques for the development of efficient programs working with data flows. Applications should be efficient so that users can find in them alternatives to perform tasks. The project uses functional programming in the constraints that work with data flows so that they can rapidly evaluate their compliance. The problems that this application solves have a significant number of variables to consider and the system should behave adequately even if the data grows. Functional programming requires a change in the way of programming that creates confusion until fully understanding the considerations to have. The OptaPlanner framework provides various classes and methods that aid with the development process in this aspect.

To broaden knowledge in web applications deployment for their main components: client and server. Nowadays, developers have implemented lots of technologies and frameworks to help with the development of web applications. Practically anyone can implement a web application just by understanding some basic concepts and sometimes without requiring special knowledge in programming. This project uses the Spring framework and its multiple modules for the development of the web application. Furthermore, the application uses Java as its primary language and dynamic web application techniques and methods.

As mentioned, the project develops widely all the goals proposed with the implementation of the application and the previous work related to analysis and design.

To conclude, the project shows a real example of using new technologies for solving problems that people actually figure out manually. The new developments in technology provide tools to aid in the digital transformation easing users to perform tasks efficiently. The intelligent systems have supposed a revolution in the ways of living because they provide solutions to problems that require *human intelligence* and, in some cases, exceeding the human abilities because they can predict some aspects of the future.

On the other hand, the increasing use of web applications has allowed companies to develop technologies that improve the user experience. Each day the number of new services published on the Internet grows, and users can perform new activities with all their devices and in any location. These are two of the advantages of web applications that motivate the development of this field. Society demands more applications ever with these two characteristics, so engineers should take these requests into account when creating new programs.

Moreover, NGOs are entities that dedicate their time and budget to perform social activities, and most of their personnel are volunteers so that they spend their time preparing the activities. Innovation and development activities typically take second place in these

associations' objectives, so the digital transformation is slower than in other entities. These associations appreciate the contributions that people make to facilitate their social work. With this project, scout associations receive an application that eases elaborating plannings, so they will not have to spend a considerable amount of time in this fact.

7.3 Future work

The project has covered the initial version of the application, but future application updates will contain additional functionalities improving user experience. Some of these improvements are:

- Implement advanced configuration options for elaborating plannings so that the organizer can select the algorithms, the time of execution, the constraint penalizations or rewards, etc. This option will give customization to allow more technological users to generate better plannings.
- Generate multiple plannings with different scout association configurations and reusing the details of scout leaders from other plannings. With this functionality, the organizer can have multiple alternatives for a planning without asking scout leaders to complete their details each time.
- Allow scout leaders to vote for one planning and generate newer ones if none of them receive a majority. This functionality will implicate more scout leaders, giving them new responsibilities.
- Create an administrative panel to allow administrators to perform actions over users through a user interface.
- Improve the API capabilities so that other connected applications can perform more options directly.

APPENDIX A

Impact of the project

This appendix contains the social, economic, environmental and ethical impact that the project could have.

A.1 Social Impact

This project's primary goal is to provide a solution to a problem that resides in scout associations. The thesis supplies an alternative to ease scout associations in creating yearly plannings. Traditionally, people do a planning manually, spending lots of hours due to the high number of considerations to take into account. Moreover, elaborating a planning in a traditional manner involves lots of arguments between scout leaders because of the relevance of assignations.

The advantages of using the application are really high: firstly to avoid the requirement of performing a meeting for the elaboration, secondly to prevent arguments between scout leaders and third to obtain a good solution to a complicated problem in just a minute.

A.2 Economic Impact

Scout associations are NGOs, so their main objective is not to obtain an economic profit from their activities. Because of this reason, the economic impact is collateral: the meeting to approve the planning of the year will last in less time, so the consumption of electrical energy and allowances would reduce.

However, with the application, a free alternative arises, and scout associations do not need to consider other options like subcontracting the elaboration of a planning to external companies. Although the economic impact is collateral, scout associations would reduce costs by using the application.

A.3 Environmental Impact

The project does not provide any significant change from an environmental point. The only subject implicated is the energy consumption spent on the development and the maintenance of the system, if scout associations would like to keep the system active. When the intelligent system module runs to find a solution, the server resources required increase, so the energy consumption is higher. Moreover, the fabrication of the servers' physical components where the application runs also has implications in the environmental impact.

If the use of the application grows, the maintainers can decide to run the system in cloud services. The energy consumption to maintain these services is higher than that used in personal computers or small servers, so the environmental impact becomes affected.

A.4 Ethical Impact

The ethical impact of the project resides in the personal data introduced by scout leaders to create plannings. If the people involved in creating the planning use these data for other purposes, the scout association could have problems. For instance, if the organizer uses the permissions to fulfill or edit the information of the scout leaders to know the people that do not want to stay with others or if he modifies these data before creating the planning without the approval of the scout leaders to create plannings more favorable for him. Moreover, if the application suffers a data leak, scout leaders' personal information would be exposed.

APPENDIX B

Project Budget

This appendix covers the costs that the project requires. This costs include human resources, physical assets and licenses.

B.1 Human resources

This section estimates the total cost of the human resources required to elaborate the project. The project requires a Telecommunications Engineer with knowledge and experience in Java application development.

The calculus of the time to elaborate the project is based on the ECTS assigned to the elaboration of the Master thesis. The Master thesis involves 30 ECTS, where 1 ECTS represents 30 hours of student work. So the total time assigned to the project ascends to 900 hours of work. Considering that working hours for a person are 8 per business day and that a month has 22 business days, an engineer would spend between 5 and 6 months to elaborate the project.

The average salary of a junior Telecommunications Engineer with the knowledge and experience required for this subject is 24.000€yearly. So the human resources costs for the project will ascend to 12.000€. These costs do not include the maintenance of the application once the development has finished.

B.2 Physical assets and services

The only physical asset required to elaborate the project is a personal computer. The minimum specifications that the personal computer needs to have are the following:

- **Processor:** AMD A9-9425, dual-core
- **RAM:** 8GB
- **Hard Disk:** 512GB SSD

A computer with these specifications has cost over 400€at the beginning of 2019. Moreover, to search for information, the personal computer needs to connect to the Internet, and with the situation produced by COVID-19, this requirement becomes crucial. The restrictions and the risks that society is experiencing make fundamental to connect to the Internet to allow people to telework without going out from home.

B.3 Licenses

All the modules and frameworks that the development uses are open source, so they do not imply additional costs over the project's total budget.

APPENDIX C

Functional programming

This is the description of the appendix.

C.1 λ Calculus

The functional programming paradigm is based on λ calculus, consisting on a single transformation rule and a single function definition scheme. Its considered to be the *smallest universal programming language of the world* and it is an approach more related to software than to hardware [26].

The most relevant concept in λ calculus is the *expression*. That is defined as:

<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>

So an application could be defined as a set of expressions, where an expression could be a variable, a function or even an application. Expressions are evaluated from the left to the right, so expressions could be chained. An example of a function is the identity function really common in multiple programming languages.

$$\lambda x.x$$

Where λx defines the arguments, in this case there is just one (x), and the expression at the right part of the dot is the function definition. To apply this function to an expression, the function should be located at the left side of the expression.

$$(\lambda x.x)y$$

This expression is used to apply the identity function defined previously to variable y. As seen before, expressions could be concatenated to be evaluated from left to right in order to produce more complex operations.

As it can be seen, this language is really useful and opens lots of possibilities in order to develop some kind of programs. Some of its characteristics are:

- Simplicity, as it can perform operations without having to develop a large amount of lines of code and because of its syntax.
- Practical, as with not so much code it can become a practical programming language. Some programs can be defined using λ calculus and the amount of code is shorter than its equivalent in other high-level languages.

- One-line universal program, because powerful programs can be developed in just one line.
- Represents data with functions, so small vocabulary notation is used but enough to carry out complex needed functions.
- Solves halt problem by using types, so λ calculus problems can be ensured to always halt in some point.
- Developed code is probably correct, because of the relation existing between λ calculus and foundations of mathematics. The advanced types used for programing make bugs impossible to express, so every syntactically correct program is also semantically correct.

Usually λ calculus is expressed in a reduced notation. This makes an abstraction over mathematical expressions and makes notation even easier to understand. An example using this reduction:

```
(+ (* 5 6) (* 8 3) )
   (+ 30 (* 8 3) )
      (+ 30 24 )
         = 54
```

That expression will return 54 as the result of evaluating, firstly inner parentheses expressions (products) will be solved and finally, the outer parentheses expression will return the last result of evaluation. Languages which implement this kind of notation, usually include some built-in functions to allow performing more complex operation in a natively manner.

C.2 Functional programming and Imperative programming comparison

Traditional programming, or what is commonly known as programming is called imperative programming. Basically, the developer types software specifying the steps the computer should carry out to reach the defined goals. Sometimes, this is known as algorithmic programming while developers are centered on developing the steps of a program in an efficient, sequential and ordered manner. With functional programming, attention should be focused

on functions or methods and its composition. Users define carefully the parameters and the result returned, so programs are formed by a set of functions normally called by each other. Imperative code usually treats with statements, pieces of code which performs some actions. Functional programming code uses expressions, pieces of code which evaluates to some value by a combination of function calls, values and operators. Table C.1 shows a comparison of both, imperative and functional techniques [13].

	Imperative programming	**Functional programming**
Programmer focus	The way to perform tasks (algorithms) and to track changes in state	The information desired and the transformations required
State changes	Important	Non-existent
Order of execution	Important	Low importance
Primary flow control	Loops, conditionals, and function or method calls	Function calls, including recursion
Primary manipulation unit	Instances of structures or classes	Functions as first-class objects and data collections.

Table C.1: Functional programming and imperative programming comparison

C.3 Transition from Object Oriented Programming to Functional Programming

Object Oriented Programming languages are commonly taught as the initials of developing code because of the simplicity of concepts in this type of programming. While this kind of languages are been taught, programmers minds become suited to this way of thinking. To switch from object oriented programming to developing in a pure functional style, a transition has to be made on how traditional software developers think and their approach to development.

Object oriented programming problem solving design class hierarchies, focus on proper encapsulation and think in terms of class contracts. The behavior and state of object types

are key concepts and in order to address this, languages provide structures such as classes, interfaces, inheritance, and polymorphism.

In the other side, functional programming base problems in the evaluation of pure functional transformations of data collections as seen before. Programs avoid to use states, mutable data and emphasizes the use of pure functions.

For the purpose of making the transition easier for traditional developers, common used programming languages support both imperative and functional programming approaches. A developer can choose which approach is suitable for an application, but typically an application uses a combination of both approaches depending on their components, its characteristics and requirements.

C.4 Disadvantages of functional programming

Other sections cover the main functional programming characteristics and advantages each of them provide to software. Now disadvantages will be exposed in order to what characteristics are been lost while coding this way [1].

- The combination of immutable variables with recursion could perform a reduction in performance.
- Code could become difficult to read because of the use of pure functions in some cases when treating with large functions.
- The integration of pure functions may become hard with the rest of the application, while coding them is considered to be easier. I/O operations might be included here as penalized also because of pure functions.
- Functional programming style can become laborious because of the use of recursivity instead of loops or working with streams and other structures detailed before.

C.5 Applications of functional programming

Traditionally, functional programming has been used for academic purposes rather than commercial software development, but nowadays tendency is changing as properties are being discovered to solve problems present in modern technology. Associated with this, functional programmers are difficult to be found as this paradigm of programming was not commonly used before.

Several programming languages are designed for the development of functional programming software such as Clojure, Erlang, F#, Haskell and Racket, widely used for developing a variety of commercial and industrial applications. Organizations as Facebook make use of Erlang using functional programming paradigm to manage the data of 1.5 billion users in WhatsApp. Also it uses Haskell in their anti-spam filtering system.

This paradigm is widely employed in applications design to work with concurrency or parallelism and for resolving mathematical computations, traditionally in programs working with large amounts of data.

Bibliography

[1] Akhil Bhadwal. *Functional Programming: Concepts, Advantages, Disadvantages, and Applications*, 2020 (accessed November 01, 2020). URL: `https://hackr.io/blog/functional-programming`.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1987.

[3] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, 76(5):349 – 375, 2011.

[4] Ting Chen and Steven S. Skiena. *Trie-based data structures for sequence assembly*. Springer, Berlin, Heidelberg, 2005.

[5] Binildas Christudas. curl and postman. In *Practical Microservices Architectural Patterns*, pages 847–855. Apress, Berkeley, CA, 6 2019.

[6] John Deacon. *Model-View-Controller (MVC) Architecture*. JOHN DEACON Computer Systems Development, Consulting and Training, 2009.

[7] Eric Elliott. *Master the JavaScript Interview: What is Functional Programming?*, 2017 (accessed October 29, 2020). URL: `https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0`.

[8] Fernando Suárez Martín Ugarte Felipe Pezoa, Juan L. Reutter and Domagoj Vrgoč. Foundations of json schema. *WWW '16: Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, 4 2016.

[9] The Apache Software Foundation. *Apache Tiles^{TM}*, 2017 (accessed October 06, 2020). URL: `https://tiles.apache.org/framework/index.html`.

[10] The Apache Software Foundation. *Apache Tomcat®*, 2020 (accessed October 03, 2020). URL: `http://tomcat.apache.org/`.

[11] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, 2004.

[12] Peter van Beek Francesca Rossi and Toby Walsh. *Handbook of Constraint Programming*, volume 2. Elsevier Science, 2006.

[13] Microsoft .NET Fundamentals. *Functional programming vs. imperative programming (LINQ to XML)*, 2015 (accessed October 30, 2020). URL: `https://docs.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming`.

[14] Google. *What is reCAPTCHA?*, 2020 (accessed November 15, 2020). URL: https://developers.google.com/recaptcha/.

[15] Red Hat. *OptaPlanner User Guide*, 2020 (accessed June 16, 2020). URL: https://docs.optaplanner.org/7.45.0.Final/optaplanner-docs/html_single/index.html.

[16] Red Hat. *What is OptaPlanner?*, 2020 (accessed June 16, 2020). URL: https://www.optaplanner.org/.

[17] Sean Leary. *JSONObject*, 2020 (accessed November 10, 2020). URL: https://github.com/stleary/JSON-java.

[18] Bruno Lowagie. *Create PDF from HTML with pdfHTML*. iText Software, 2017.

[19] Gustavo A. Oliva Maurício F. Aniche and Marco A. Gerosa. Are the methods in your data access objects (daos) in the right place? a preliminary study. In *2014 Sixth International Workshop on Managing Technical Debt*. IEEE, 9 2014.

[20] Jim Melton and Alan R. Simon. *SQL: 1999: Understanding Relational Language Components*. Morgan Kaufmann Publishers, 2002.

[21] Yanina Muradas. *Conoce qué es Spring Framework y por qué usarlo*, 2018 (accessed October 05, 2020). URL: https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/.

[22] Google OR-Tools. *Constraint Optimization*, 2020 (accessed October 25, 2020). URL: https://developers.google.com/optimization/cp.

[23] Pankaj. *Java Web Application Tutorial for Beginners*, 2020 (accessed October 01, 2020). URL: https://www.journaldev.com/1854/java-web-application-tutorial-for-beginners.

[24] Terence Parr. *String Template*, 2020 (accessed November 02, 2020). URL: https://www.stringtemplate.org/.

[25] Enric Rodríguez-Carbonell. *Introduction to Constraint Programming*, 2020 (accessed October 30, 2020). URL: https://www.cs.upc.edu/~erodri/webpage/cps//theory/cp/intro/slides.pdf.

[26] Raúl Rojas. *A Tutorial Introduction to the Lambda Calculus*, 1997 (accessed November 07, 2020). URL: https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf.

[27] Scout.org. *World Scout Foundation*, 2020 (accessed June 01, 2020). URL: https://worldscoutfoundation.org/home.

[28] Pivotal Software. *Interface SolverManager*, 2020 (accessed July 02, 2020). URL: https://docs.optaplanner.org/7.33.0.Final/optaplanner-javadoc/org/optaplanner/core/api/solver/SolverManager.html.

[29] Pivotal Software. *Planner Configuration*, 2020 (accessed June 15, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-docs/html_single/#plannerConfiguration.

[30] Pivotal Software. *Class HardMediumSoftScore*, 2020 (accessed June 18, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/score/buildin/hardmediumsoft/HardMediumSoftScore.html.

[31] Pivotal Software. *Interface ConstraintProvider*, 2020 (accessed June 18, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/score/stream/ConstraintProvider.html.

[32] Pivotal Software. *Annotation Type PlanningEntity*, 2020 (accessed June 20, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/entity/PlanningEntity.html.

[33] Pivotal Software. *Annotation Type PlanningId*, 2020 (accessed June 20, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/lookup/PlanningId.html.

[34] Pivotal Software. *Annotation Type PlanningSolution*, 2020 (accessed June 20, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/solution/PlanningSolution.html.

[35] Pivotal Software. *Annotation Type PlanningVariable*, 2020 (accessed June 20, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/variable/PlanningVariable.html.

[36] Pivotal Software. *Interface Constraint*, 2020 (accessed June 20, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/score/stream/Constraint.html.

[37] Pivotal Software. *Score Calculation*, 2020 (accessed June 22, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-docs/html_single/#scoreCalculation.

[38] Pivotal Software. *Annotation Type PlanningEntityCollectionProperty*, 2020 (accessed June 23, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/solution/PlanningEntityCollectionProperty.html.

[39] Pivotal Software. *Annotation Type PlanningScore*, 2020 (accessed June 23, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/solution/PlanningScore.html.

[40] Pivotal Software. *Annotation Type ProblemFactCollectionProperty*, 2020 (accessed June 23, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/solution/drools/ProblemFactCollectionProperty.html.

[41] Pivotal Software. *Annotation Type ValueRangeProvider*, 2020 (accessed June 23, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/api/domain/valuerange/ValueRangeProvider.html.

[42] Pivotal Software. *Interface ProblemFactChange*, 2020 (accessed June 25, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/impl/solver/ProblemFactChange.html.

[43] Pivotal Software. *Constraint Streams - Modern Java constraints without the Drools Rule Language*, 2020 (accessed June 27, 2020). URL: https://www.optaplanner.org/blog/2020/04/07/ConstraintStreams.html.

[44] Pivotal Software. *Class SolverConfig*, 2020 (accessed November 01, 2020). URL: https://docs.optaplanner.org/7.28.0.Final/optaplanner-javadoc/org/optaplanner/core/config/solver/SolverConfig.html.

[45] Pivotal Software. *Class SolverManagerConfig*, 2020 (accessed November 01, 2020). URL: https://docs.optaplanner.org/7.38.0.Final/optaplanner-javadoc/org/optaplanner/core/config/solver/SolverManagerConfig.html.

[46] Pivotal Software. *Spring Framework*, 2020 (accessed October 01, 2020). URL: https://spring.io/.

[47] Pivotal Software. *Spring Web MVC*, 2020 (accessed October 05, 2020). URL: https://spring.io/projects/spring-security.

[48] Pivotal Software. *Spring Security*, 2020 (accessed October 20, 2020). URL: https://spring.io/projects/spring-security.

[49] Pivotal Software. *Class WebSecurityConfigurerAdapter*, 2020 (accessed October 29, 2020). URL: https://docs.spring.io/spring-security/site/docs/5.4.0/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html.

[50] Pivotal Software. *Authorization*, 2020 (accessed October 30, 2020). URL: https://docs.spring.io/spring-security/site/docs/5.4.0/reference/html5/#servlet-authorization.

[51] Pivotal Software. *Interface UserDetails*, 2020 (accessed October 30, 2020). URL: https://docs.spring.io/spring-security/site/docs/5.4.0/api/org/springframework/security/core/userdetails/UserDetails.html.

[52] Pivotal Software. *Interface UserDetailsService*, 2020 (accessed October 30, 2020). URL: https://docs.spring.io/spring-security/site/docs/5.4.0/api/org/springframework/security/core/userdetails/UserDetailsService.html.

[53] Pivotal Software. *Class bcryptpasswordencoder*, 2020 (accessed October 31, 2020). URL: https://docs.spring.io/spring-security/site/docs/5.4.0/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html.

[54] Steve Suehring. *MySQL Bible*. Wiley Publishing, Inc, 2002.

[55] Nguyen Nam Thai. *What is a Spring Bean?*, 2020 (accessed October 05, 2020). URL: https://www.baeldung.com/spring-bean.

[56] Jean Paoli Tim Bray and C. M. Sperberg-McQueen. *Extensible Markup Language(XML) 1.0*. W3C (MIT, INRIA, Keio), 1998.

[57] TutorialsPoint. *Drools - Rule Syntax*, 2020 (accessed June 19, 2020). URL: https://www.tutorialspoint.com/drools/drools_rule_syntax.htm.

[58] TutorialsPoint. *JSP - Standard Tag Library (JSTL) Tutorial*, 2020 (accessed October 10, 2020). URL: https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm.

[59] Aleksa Vukotic and James Goodwill. *Apache Tomcat 7*. Apress, Berkeley, CA, 2011.

[60] Philip Wadler. *Monads for functional programming*. Springer, Berlin, Heidelberg, 1995.

[61] Desarrollo web. *Conceptos básicos: definición de web app y ejemplos*, 2019 (accessed November 08, 2020). URL: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-una-web-app-y-que-clases-hay/.

[62] CIO Wiki. *Client Server Architecture*, 2020 (accessed November 11, 2020). URL: https://cio-wiki.org/wiki/Client_Server_Architecture.

[63] Wikipedia. *Web application*, 2020 (accessed November 08, 2020). URL: https://en.wikipedia.org/wiki/Web_application.

[64] Wikipedia. *Client–server model*, 2020 (accessed November 10, 2020). URL: https://en.wikipedia.org/wiki/Client%E2%80%93server_model.

[65] Wikipedia. *Jakarta Server Pages*, 2020 (accessed October 10, 2020). URL: https://en.wikipedia.org/wiki/Jakarta_Server_Pages.

[66] Wikipedia. *Constraint programming*, 2020 (accessed October 30, 2020). URL: https://en.wikipedia.org/wiki/Constraint_programming.

[67] David Yang. *The 9 Best Programming Languages to Learn in 2020*, 2020 (accessed November 15, 2020). URL: https://www.fullstackacademy.com/blog/nine-best-programming-languages-to-learn.

[68] Ángel Robledano. *Qué es MySQL: Características y ventajas*, 2019 (accessed October 10, 2020). URL: https://openwebinars.net/blog/que-es-mysql.