Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Čestmír Houška

Tournament Management System

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. David Obdržálek

Studijní program: Informatika, programování

2010

Děkuji panu RNDr. Davidu Obdržálkovi za ochotu být vedoucím této práce, jakož i za podnětné rady a připomínky k programování aplikace.

Také bych rád poděkoval Mgr. Ondřeji Luksovi za poskytnutí dat ze soutěže Eurobot 2010 k testování.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Čestmír Houška

Contents

Název práce: Tournament Management System
Autor: Čestmír Houška
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Obdržálek
e-mail vedoucího: david.obdrzalek@mff.cuni.cz

Abstrakt: V této práci jsou analyzována specifika různých turnajových systémů a navržen objektový datový model, pomocí kterého lze tyto turnajové systémy příhodně modelovat. Součástí práce je dále aplikace pro správu turnajů, ve které je tento model využit. Důraz byl kladen především na otevřený charakter a také na rozšiřitelnost a univerzálnost aplikace. Další důležitou součástí práce je případová studie, která ukazuje, jak může být aplikace využita v praxi. Tato případová studie je provedena na skutečných datech z robotické soutěže Eurobot 2010.

Klíčová slova: správa turnajů, soutěže, zapisování výsledků, software

Title: Tournament Management System
Author: Čestmír Houška
Department: Department of Software Engineering
Supervisor: RNDr. David Obdržálek
Supervisor's e-mail address: david.obdrzalek@mff.cuni.cz

Abstract: In this work the specifics of various tournament systems are analyzed and an object data model is proposed that can be used to model these tournament systems conveniently. A tournament management application that implements this model comprises another part of the work. Emphasis was given to the open nature and to the extensibility and universality of the application. Another important part of the work is a case study that shows how the application can be used in praxis. The case study is made using the real data from a robotic competition Eurobot 2010.

Keywords: tournament management, competitions, score-keeping, software

Chapter 1

Introduction

One of the reasons why sports exist is that people like to compete with each other and measure their performance against other contestants. That is why every sport includes a scoring system that somehow mirrors the performance of the contestants. However, as Jack Richards and Danny Hill correctly argue in [2], the interesting data from sports and competitions is not only the scores and who won and who lost. Statistics and records are as much important as the scores, especially to couches, more knowledgeable fans, media, but also to wide public. The task of collecting the statistics then becomes even more difficult when we want to manage a whole tournament.

The task of keeping the scores and other statistical data was very tedious in the past and required a lot of people. Personal computers made this task potentially much easier, but to my best knowledge, no tournament management system exists that is free and open on one hand and easy to use and universal on the other.

1.1 Motivation

During Eurobot 2008 Czech cup, I was responsible for keeping the scores and presenting them to the audience. This was done using a computer to make the presentation of the scores easier. For more information on the Eurobot competition, see [5].

The application that was used back then (and also in the following and several preceding years) was a rather complicated spreadsheet in Microsoft Excel. It was not very trivial for anyone to understand the principles of the spreadsheet as there were a lot of complicated formulas, cross-references, hidden values and other workarounds that prevented one to grasp the overall concept of the file. This solution was the outcome of incremental programming efforts of several people over several years of lifetime of the spreadsheet. When a problems arose in the application, it was hacked with an ad-hoc workaround instead of being solved and this hack was carried on to the next version, making it even more complicated.

Moreover, during the use of the score-keeping system, an unexpected situation or two arose almost every year that called for editing some automatically updated values by hand with the possibility of breaking chains of inter-referencing cells and causing a cascade of problems later, which sometimes did indeed happen. Nothing is then more unnerving than delegates from the affected teams, complaining about the wrong score of their teams while you are already trying to fix the problem.

More issues included misuse of the system due to incomplete understanding of how it worked. This caused problems similar to those already described — frustrated contestants, confused audience and embarrassed organizers.

Presentation of the scores was also tedious as it consisted of doing the following procedure every time the scorekeeper changed the scores and wanted to show them: one had to locate the right tab, where the scores were presented in the desired format, switch the spreadsheet application to full-screen view, hide the mouse pointer so that it would be invisible on the projector, unfreeze the projector and freeze it back so that the audience would not see the scorekeeper writing down the scores. Of course, this was not very user-friendly and sometimes one forgot to do a step in this sequence, causing the presentation to look unprofessional.

To address the previously mentioned problems, I began to think about a more robust computer system that would solve them and this work is the result.

1.2 Structure of the work

The rest of the text is organized as follows: Chapter 2 analyzes the requirements on the system, describes several existing applications that solve similar problems and specifies the data model design that will be used in the system. Chapter 3 contains details about my implementation of the design. Chapter 4 describes the application from a user's point of view. Chapter 5 shows a case study that uses the application to manage matches in the Eurobot 2010 competition. Chapter 6 concludes the achievements of this work. The appendix chapters describe the incorporated Lua scripting API, explain unclear terminology and list the contents of the accompanying CD.

Chapter 2

Design and analysis

This chapter analyzes the requirements on the application and looks at existing software. Also, characteristics of various tournament types are observed and a data model with a means to manipulate the data is provided that will suit these characteristics. At the end of the chapter, the model is compared with the initial requirements.

2.1 Requirements

As a result of my analysis that considered the issues that were experienced with the solutions mentioned in section 1.1, following requirements arose:

Universality

The system has to be designed in such a way that will allow its use in a wide variety of tournaments and competitions. Parts of the application should be script-able and editable by the user so that it can be modified to the specific needs of the given event. Should a particular employment of the system show that the system misses a feature important for that employment, the design of the system should make it easy to add support for the feature in a future version.

Modularity

This requirement is actually a means of achieving universality. By making the application modular (together with open design), we will allow its users to change parts of it for something that they already use or design a specific module that will suit their needs perfectly.

Usability

Use of the system should not be limited to programmers or people proficient with information technology. This will allow the referees and scorekeepers to use the application without help from a tournament system administrator. The typical user of the application understands the rules of the game and wants to use the system to record anything that the rules allow without having to learn to program.

Auditability

Audit in this case does not mean an external third party scrutiny of the system, but an action performed by the system administrator that leads to identification of a problem. The system should log all important actions to allow the administrator to find out how the problem was created.

Recoverability

After identifying the problem, the administrator — or maybe even the user — has to solve it somehow. This should not be very difficult and the system should be prepared for such interventions.

Open design

In case of more serious problems, one sometimes has to modify the underlying data manually. This would be impossible if the data format would be proprietary or poorly documented. The data should thus be easy to access manually — that is without the mediation of the application — and it should be presented in a format that is easily editable. The system itself should also be well documented and its source code available. Open source code will allow other people to build upon or extend the application with new features in the future.

Multi-platform use

Small hobbyist tournaments or competitions usually do not have several computers at their disposal and they may not have the time, resources or interest in buying or installing a specific hardware or software. The system should therefore be compilable with and able to run on most of the major operating systems.

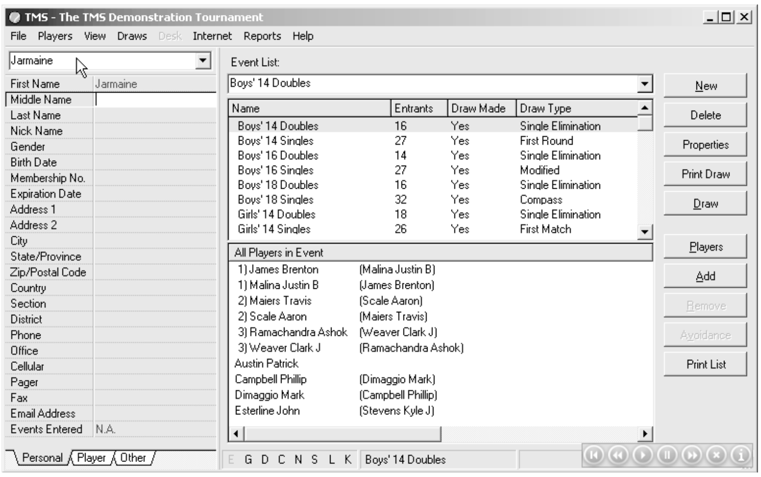
Cost

Because we did not want to spend money for such a specific software as a score-keeping system when it gets used just once a year and because it could show that the newly acquired application misses an important feature, we understand that other small tournament organizers face the same restrictions. A cost-free open-source application is a solution to this problem.

2.2 Existing applications

To better illustrate the variety of tournament managements systems available on the world wide web, I will describe a few examples in detail. The choice of the individual applications is not based on any rigid criterion, instead I chose the applications that stood out the most because of their position in web searches, good presentation and professional look and feel or features that make them representatives for a certain group of applications.

TMS — Tournament Management System (Tennis)



Jarmaine

First Name	Jarmaine
Middle Name	
Last Name	
Nick Name	
Gender	
Birth Date	
Membership No.	
Expiration Date	
Address 1	
Address 2	
City	
State/Province	
Zip/Postal Code	
Country	
Section	
District	
Phone	
Office	
Cellular	
Pager	
Fax	
Email Address	
Events Entered	N.A.

Event List: Boys' 14 Doubles

Name	Entrants	Draw Made	Draw Type
Boys' 14 Doubles	16	Yes	Single Elimination
Boys' 14 Singles	27	Yes	First Round
Boys' 16 Doubles	14	Yes	Single Elimination
Boys' 16 Singles	27	Yes	Modified
Boys' 18 Doubles	16	Yes	Single Elimination
Boys' 18 Singles	32	Yes	Compass
Girls' 14 Doubles	18	Yes	Single Elimination
Girls' 14 Singles	26	Yes	First Match

All Players in Event	
1) James Brenton	(Malina Justin B)
1) Malina Justin B	(James Brenton)
2) Maiers Travis	(Scale Aaron)
2) Scale Aaron	(Maiers Travis)
3) Ramachandra Ashok	(Weaver Clark J)
3) Weaver Clark J	(Ramachandra Ashok)
Austin Patrick	
Campbell Phillip	(Dimaggio Mark)
Dimaggio Mark	(Campbell Phillip)
Esterline John	(Stevens Kyle J)

Boys' 14 Doubles

Figure 2.1: *Data input GUI of Tournament Management System*

T.M.S. Tournament Management System [16] is a good representative of the commercial tournament management software. The graphical user interface looks neat and professional (see Figure 2.1) and the application offers a lot of interesting features like site scheduling, automatic or manual draw making, sheet printing and Internet publishing. The official website claims that the United States Tennis Association has purchased a nationwide license for this system.

The system goes a long way in simplifying the usability, which is due to its commercial nature, but let's look at how it would comply with my other criteria.

Universality and modularity are not fulfilled in this application, because it is intended for use in tennis tournaments and there is one modification that allows it to be used for racquetball. One might argue that it is a specialized application and it is not intended to be universal or modular, but for my application I chose different goals.

Error detection and recovery is not needed as in our case because the application does not automatically count score and other statistics. In case of assistance, one could also call the support hot-line (which is available with nearly all of the commercial solutions).

Regarding the application itself, it is not multi-platform and it is closed-source. It is also not free — users have to pay annual subscription fees and also pay for publishing the tournaments.

STRONGVON

Strongvon [15] is another example of a commercial tournament management system. It is strongly oriented on Internet publishing and registration, but also includes bracketing tools. However, according to their own website, the application can save only wins/losses and cannot display scores or other statistics.

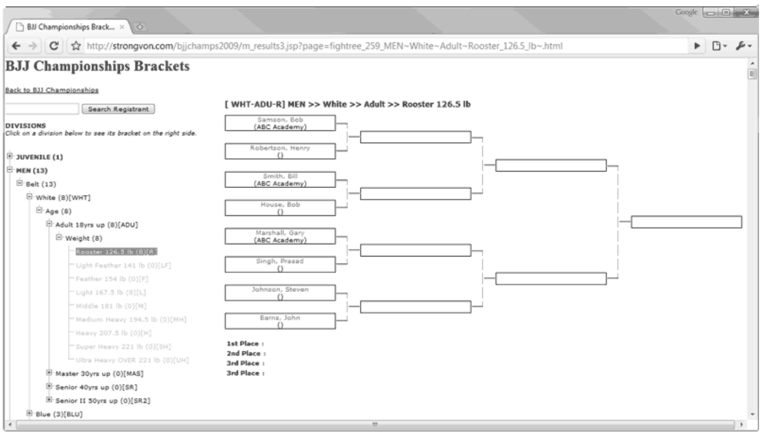

BJJ Championships Brack...
http://strongvon.com/bjjchamps2009/m_results3.jsp?page=fightree_259_MEN~White~Adult~Rooster_126.5_lb~.html
BJJ Championships Brackets
Back to BJJ Championships
Search Registrant
DIVISIONS
Click on a division below to see its bracket on the right side.
JUVENILE (1)
MEN (13)
Belt (13)
White (8)[WHT]
Age (8)
Adult 18yrs up (8)[ADU]
Weight (8)
Rooster 126.5 lb (8)[R]
Light Feather 141 lb (0)[LF]
Feather 154 lb (0)[F]
Light 167.5 lb (0)[L]
Middle 181 lb (0)[M]
Medium Heavy 194.5 lb (0)[MH]
Heavy 207.5 lb (0)[H]
Super Heavy 221 lb (0)[SH]
Ultra Heavy OVER 221 lb (0)[UH]
Master 30yrs up (0)[MAS]
Senior 40yrs up (0)[SR]
Senior II 50yrs up (0)[SR2]
Blue (3)[BLU]
[WHT-ADU-R] MEN >> White >> Adult >> Rooster 126.5 lb
Samson, Bob (ABC Academy)
Robertson, Henry ()
Smith, Bill (ABC Academy)
House, Bob ()
Marshall, Gary (ABC Academy)
Singh, Prasad ()
Johnson, Steven ()
Barns, John ()
1st Place :
2nd Place :
3rd Place :
3rd Place :

Figure 2.2: *Online bracket view in Strongvon*

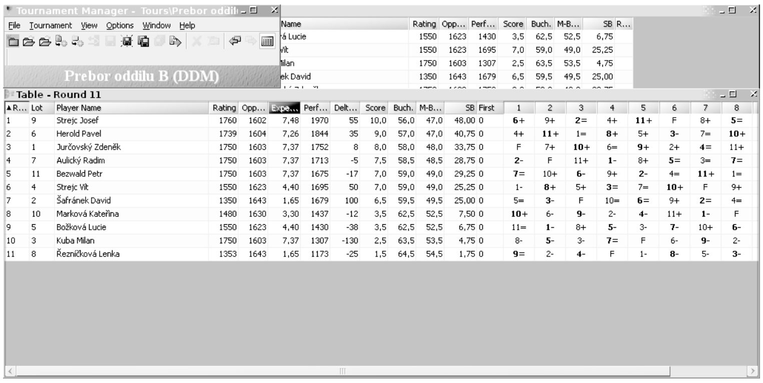
The program's features include automatically grouping competitors into groups based on various criteria, registration of the competitors online or printing tournament brackets and schedules.

As for my criteria, Strongvon seems (and claims) to be relatively universal, although they mention martial arts several times on the website. However, it is not modular and thus not customizable. Strongvon's source is closed and the use of the application is not free. Similarly to previous application, Strongvon utilizes the pay-per-use scheme to charge its users.

David Šafránek's Tournament Manager

David Šafránek's Tournament Manager [3] is a specialized tournament manager for chess tournaments that features swiss system or round-robin tournaments, managing multiple tournaments simultaneously or a lot of chess-related statistics. However, it lacks the more advanced usability related features of the commercial applications like printing out the draws, Internet publishing or online registration.

If we look at the criteria, universality is obviously not fulfilled, modularity neither. Nevertheless, the application is relatively usable and it is free, although not open-source. It also does not run on multiple platforms.



Name	Rating	Opp…	Perf…	Score	Buch.	M-B…	SB	R…
á Lucie	1550	1623	1430	3,5	62,5	52,5	6,75	
Vít	1550	1623	1695	7,0	59,0	49,0	25,25	
ilan	1750	1603	1307	2,5	63,5	53,5	4,75	
ek David	1350	1643	1679	6,5	59,5	49,5	25,00	

Table - Round 11

▲R…	Lot	Player Name	Rating	Opp…	Expe…	Perf…	Delt…	Score	Buch.	M-B…	SB	First	1	2	3	4	5	6	7	8
1	9	Strejc Josef	1760	1602	7,48	1970	55	10,0	56,0	47,0	48,00	0	**6+**	9+	**2=**	4+	**11+**	F	8+	**5=**
2	6	Herold Pavel	1739	1604	7,26	1844	35	9,0	57,0	47,0	40,75	0	4+	**11+**	1=	**8+**	5+	**3-**	7=	**10+**
3	1	Jurčovský Zdeněk	1750	1603	7,37	1752	8	8,0	58,0	48,0	33,75	0	F	7+	**10+**	6=	**9+**	2+	**4=**	11+
4	7	Aulický Radim	1750	1603	7,37	1713	-5	7,5	58,5	48,5	28,75	0	**2-**	F	11+	**1-**	8+	**5=**	3=	**7=**
5	11	Bezwald Petr	1750	1603	7,37	1675	-17	7,0	59,0	49,0	29,25	0	**7=**	10+	**6-**	9+	**2-**	4=	**11+**	1=
6	4	Strejc Vít	1550	1623	4,40	1695	50	7,0	59,0	49,0	25,25	0	1-	**8+**	5+	**3=**	7=	**10+**	F	9+
7	2	Šafránek David	1350	1643	1,65	1679	100	6,5	59,5	49,5	25,00	0	5=	**3-**	F	10=	**6=**	9+	**2=**	4=
8	10	Marková Kateřina	1480	1630	3,30	1437	-12	3,5	62,5	52,5	7,50	0	**10+**	6-	**9-**	2-	**4-**	11+	**1-**	F
9	5	Božková Lucie	1550	1623	4,40	1430	-38	3,5	62,5	52,5	6,75	0	11=	**1-**	8+	**5-**	3-	**7-**	10+	**6-**
10	3	Kuba Milan	1750	1603	7,37	1307	-130	2,5	63,5	53,5	4,75	0	8-	**5-**	3-	**7=**	F	6-	**9-**	2-
11	8	Řezníčková Lenka	1353	1643	1,65	1173	-25	1,5	64,5	54,5	1,75	0	**9=**	2-	**4-**	F	1-	**8-**	5-	**3-**

Figure 2.3: *David Šafránek's Tournament Manager*

Conclusion

Research shows that the supply of computerized tournament management solutions on the market is relatively vast. Nevertheless, most of these tools are online and commercial and the rest is either narrowly specialized, too restrictive or lacking important features. Moreover, some of the commercial solutions tend to utilize a pay-per-usage approach, requiring the user to pay a predefined amount of money not only for every tournament, but also for every contestant that enters the competition. Together with limitations on the number of teams and contestants and other restrictions, these systems will not be an alternative for many of their potential users.

2.3 Data model

The current design makes the application adaptable to various types of sports and competitions by providing a means to define behavior of the system in particular situations. It also tries to be as general as possible while keeping in mind the common characteristics of most competitions. The latter is important to differentiate the system from a general-purpose database application. I observed these characteristics to be able to derive properties of the underlying data model of the system.

Object analysis

The data of my system is object-oriented in that it will be able to combine related data into objects and provide polymorphism and basic inheritance — so that for example all types of contestants will the share some properties.

I will illustrate the need for types in my object system on the case of matches. Generally speaking, a tournament is a series of events that is either homogeneous or heterogeneous. An example of a homogeneous tournament is virtually any professional sports league, be it association football, ice hockey, baseball or a variety of other competitive sports. On the other hand, decathlon, triathlon, athletics and similar tournaments all consist of more types of contests and are thus heterogeneous. The existence of heterogeneous tournaments implies for the application the possibility to define more types of matches and be able to use them in one tournament.

However, what both homogeneous and heterogeneous tournaments share is that they have a certain structure. They consist of single matches that can be grouped together in rounds, tables and similar group-like configurations. Matches that consist of multiple rounds and require the teams to win a predefined number of times to win the entire match (playoff matches in the National Hockey League in the USA but also other playoff matches in various sports all over the world) can also be considered a group of matches. Some matches are played twice with the teams switching home and away roles, for example in association football. All these examples lead to the need of a grouping feature in the system — matches have to be group-able and the whole group has to be able to determine winners of the group in a similar way as a match does. In fact, the groups themselves can also be included in further groups, which leads to a tree-like hierarchical system of match groups and matches.

So far, I considered only games that feature two contestants and have a winner and a loser. But there can also be matches with more contestants, as for example in racing competitions. Most card games can also accommodate — if not require — more than two players. Similarly, sometimes we do not want to know only the winner of the match, but we also want to determine positions of the other contestants. These examples led me to use a variable number of contestants and winners or in other words inputs and outputs of the matches and match groups.

Players or teams (I have called them *actors* since my early analysis of the system, although *contestants* would be a more appropriate term), matches and match groups (called *tables* in some places of the source code) are enough object types to model entities in the most basic scoring systems, but sometimes the user will need to define another object type to represent other real-world entities such as venues, referees or track checkpoints as well as imaginary helper objects to store various data. The application has to allow the user to do so by adding a third type of objects besides contestants, matches and match groups. I chose to call them simply *objects*.

Object model

As was revealed in the above analysis, my system will work with four types of objects: actors, matches, tables and objects. The user will define derived types (or sub-types) from these basic types and will be able to create and delete instances of these sub-types. The instances will contain as member variables simple data such as strings, numbers or boolean values, but also references to other instances.

The object model will also need to include variable-size arrays to accommodate matches of variable contestant number, as per my analysis. These arrays will however be useful in other cases such as list of team's players, list of goals or similar events in a match, list of checkpoints of a racing track and so on.

Apart from instances of four types of objects, the object will also need to support global variables to allow the user to save data that is relevant to the tournament as a whole.

Data back-end

The described object model is just an abstraction over a back-end data storage. In order to fulfill the openness requirement, the data model needs to be implemented either by plain text files or a freely available database management system. The implementation details of the abstraction should be documented, so that an advanced user or administrator, who wishes to work with the data without the help of the tournament management application, is able to do so.

Object model extension

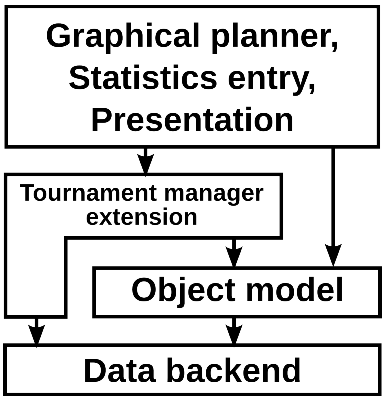


Figure 2.4: *Data model of my system*

The object model, as I described it, only serves as an abstraction over the data back-end. But in the application, I will need to have certain pre-defined globals, for example an array of actors that will contain references to all contestants. For the purposes of the graphical application, the instances will also need to have automatically created implicit variables that will contain various graphics-related data, for example the position of the object on the screen. There will also be connections — a facility to connect outputs of matches and match groups with inputs of other matches and match groups.

All these requirements (apart from connections) can be realized in our object model, but it would be impractical to create the needed variables manually every time a new tournament is created or an object is instantiated. The tournament manager extension to the object model will do that for us. As for the connections, they will be stored in the data back-end separately from the object model. They cannot be built on top of the model without difficulties because they connect various object types (matches or match groups).

Communication patterns

The application will not communicate directly with the data back-end. Instead, it will do that with the help of either the tournament extension to the object model or directly the object model. Figure 2.4 describes communication patterns inside the data model and between the data model and the rest of the system.

2.4 Event system

The model, as I have presented it so far, can describe the tournament's state at a given time. However, we also need to record changes somehow. For this, I have designed an event system, which consists of a set of user-defined events that can be executed to modify the data model. The time in my tournament management system will be discretized and an event will be a transition between two time intervals that represent two states of the data model.

An event can happen on demand from a user such as a referee or scorekeeper or it can be delivered to the application from an automated sensor somewhere in the field. Upon the arrival of an event, the application will parse its arguments and run the event handling routine that will have been defined in a scripting language by the administrator before starting the system. Events will be logged and it will be possible to reconstruct the whole tournament just by reading this log.

Short example

To clarify the model a little bit, let's observe the change of data by events on a simple example. For the purpose of the example, I will show only the part of the data that is changing.

There will be an object of base type *match* with an identifier *"match1"*. Subtype of this object is not so important, we only know that it contains two integer variables — *home* and *away* — that will keep the score of the match.

For the purpose of modifying the score, there will be an event named *"score"* that will have three arguments. These arguments will be named *"match"*, *"home"* and *"away"* and their types will be *match*, *integer* and *integer* respectively. Follows a pseudo-code description of the event. Note that the actual implementation might look different, based on the used scripting language.

```
event score(match match, integer home, integer away) {
  match.home := home
  match.away := away
}
```

This is how the data might look after a few invocations of this event. Do not get confused by mixing of views of the data with scripting language pseudo-code — odd lines show current state of data, even lines are pseudo-code that is about to be executed by the scripting language interpret.

```
match1{home = 0; away = 0}
  score(&match1, 1, 0); //Home team just shot a goal
match1{home = 1; away = 0}
  score(&match1, 1, 1); //Away team shoots one as well
match1{home = 1; away = 1}
  //And so forth
```

Ordering group outputs

In order to be able to implement qualification tables, two extra arrays of actors and a special ordering event will be a part of each match group. The arrays will be named input and output ordering arrays and they will be connectible by connections inside the match group. When an actor in the input ordering array changes, the ordering event for the given match group type will be executed and, will determine the correct ordering of the actors and will set the output ordering array correspondingly. That way, order of the outgoing actors will not have to be defined by the connections only, but will be modifiable by an event. The ordering event will also be runnable manually.

Thus, to create a qualification table, one would just add the qualification matches inside the match group and when all of the matches are finished (or in any time in between — ideally after a single match is finished), the ordering event would be executed to sort the table's players according to their performance in the matches.

2.5 Modules

For more complex uses, one application will not suffice. A setup, where there are several people working simultaneously on several computers and inputting various statistical data, is easily imaginable. Simply running more instances of the tournament management application would be an option, but since all of them would have to share the same data back-end, this could bring concurrency problems like reading a temporary value of a variable or overwriting a value that has just been written by another user. Put into terms of the data model abstraction, transitions between the states must be atomic and the event handler execution must be serialized.

To allow serialization of the events, the system has to be divided into several applications that communicate with a central data host application. This central application provides the data model abstraction over the data back-end and is accessible via a dynamically linked library that is provided with the system. All applications that need to communicate with the central module then have to be linked against this library. The communication mechanism should allow the applications to run either all on one machine or to be distributed over the network for easier collaboration of multiple users. A basic tournament planner and event executor applications will be available without programming anything, so that users without too specific needs will be able to use the system.

2.6 Design vs. requirements

The design fulfills all of my requirements mentioned at the beginning of this chapter.

The system will be universal for several reasons. The underlying data model is general enough to suit a wide variety of tournaments and it can be modified by defining own sub-types and own events. Modularity will be satisfied due to network nature of the whole system and also because the presentation modules will be able to directly read the database without having to communicate with the data model.

Usability cannot be hindered anyhow by the design — the implementation of the graphical front-end of the application will decide, whether the application will be intuitive or not. Auditability will be satisfied due to various logging facilities, mainly the event log. Recoverability is ensured by the nature of the underlying data back-end, because the whole database schema can be periodically backed up. Also, in the case of less severe problems, one might be able to modify the data directly because of the open nature of the data model. Which leads us to the requirement on open-design. Apart from the open and documented format of the data model, this will be ensured by the open-source nature of the code, although this is an implementation detail, as well as the multi-platform nature of the used tools and the zero cost of the application for the users.

Chapter 3

Implementation

This chapter contains information about the programming languages and other technological instruments that I used in the application implementing my data model, looks at the implementation from a broader perspective, describes how the original design was modified and for what reasons and also contains detailed implementation-level description of parts of the application such as the graphical user interface, embedded scripting language and individual classes.

3.1 Technologies

C++

I chose C++ [1] as the main programming language for my application because of its multi-platformness, the right level of abstraction from the hardware specifics (C being too low-level and python too abstract, for comparison), availability of a lot of libraries and mostly because of my previous experience with the language. The C++ STL library is also used.

MySQL

MySQL is an open-source database system. [9]

The idea of using text files as the data back-end was quickly abandoned. Although text files would have an advantage of being easily editable, working with them would either mean a significant overhead (because searching in a text file would be linear in the file's length) or I would end up designing my own database management system. Using a third-party database management system is much more convenient for me as a programmer of the application and does not impose too much restrictions on the users of the system if an open and widely adopted alternative is chosen. Moreover, database management systems provide additional features that might prove useful — for example backups or user management.

The two open database management systems that I was considering were SQLite and MySQL. I chose MySQL because I was familiar with it and also because there might be a need to use my system over a network and read the data by more clients, which the official SQLite website states as a reason to consider not using SQLite [14].

Lua

Lua is a lightweight embeddable scripting language. [11]

Lua was chosen as a scripting language because of its simplicity, flexibility and because it is easily embeddable. As opposed to the data back-end, complex additional features in the scripting language would be unnecessary because event handlers are intended to be as small and quick as possible.

Regarding the performance of Lua, the official website of Lua claims:

> “Lua has a deserved reputation for performance. To claim to be “as fast as Lua” is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages.”

Qt

Qt is a multi-platform GUI toolkit. [17]

It was chosen because it is open and multi-platform. It is also well documented and has many interesting features.

Mysql++

Mysql++ [10] is used as interface with the MySQL C API [10]. It is a C++ MySQL wrapper that uses similar principles as STL.

SCons

SCons is a software construction tool. [13]

I use SCons as the build management tool, because it is an interesting alternative to the traditional Unix make tools. Although the current build was tested only on GNU/Linux, it should not be very difficult to adapt it to other systems due to SCons’ multi-platform nature. SCons is written in the Python programming language. [12]

Operating system

The operating system that was used for development was the Ubuntu GNU/Linux distribution. However, the system is multiplatform and can be compiled on a variety of other major operating systems.

3.2 General view

The application can be divided into several distinct components, each having a well-defined purpose and consisting of several singleton classes. The main entry function of the program instantiates these classes, connects them together by giving them pointers to the other singletons, parses the arguments using the argument parser class and executes the actions that correspond to the parsed arguments and then enters a loop by calling the Qt function `QApplication::exec()`. The event-based Qt GUI library then executes my functions and thus returns control to my code.

Three main components

The Qt-based part of the application displays the main window, handles the events for the GUI elements such as button presses, drag and drops or mouse movements. The code for this part of the application is in the `src/qt` directory. Most of the code involves creating new classes for graphical representation of matches and match groups and enabling the user to interconnect them interactively. This part also includes displaying a console for debugging and similar purposes, wherein the user can access the Lua environment without having to use events.

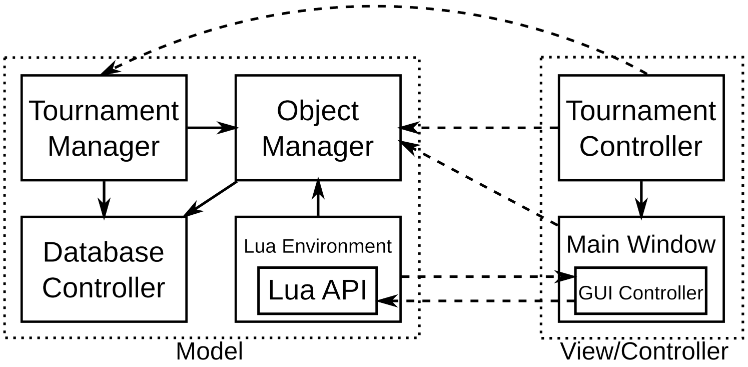


Figure 3.1: *MVC view of the system. Classes are solid-lined rectangles, dotted rectangles are parts of MVC. Solid arrows denote flow of control inside the dotted rectangles, while the dashed lines represent communication that will have to be realized over the network, if the application is to be divided.*

Another distinct part of the application is the data model that handles connection to the back-end database system and provides the object abstraction over the database. Another important task for this part of the system is to provide an observer mechanism, so that other parts of the application can be notified when something in the data model changes.

Last component of the system is the Lua environment. It provides a C++ extension to the original Lua C API, handles the initialization of the Lua environment and creates the objects and tables that are specific to the tournament manager application — the so-called Lua scripting API. Another important task for the Lua environment is to load events from text files and allow them to be executed.

Deviations from the design

Due to the fact that implementing the original design in its completeness would be way out of the scope of this work, I did not subdivide the system into several applications that communicate over a network and made the whole system as one application instead. However, the application was still programmed with modularity in mind, so as not to decline from the original design too much and to allow future division of the application into the originally proposed components.

To see how the division of the application is possible, let's observe how the three main parts of the application communicate and how that resembles the Model-View-Controller design pattern. To divide the application, it would suffice to separate the data model part, that corresponds to the "Model" in the MVC pattern, together with the Lua scripting API from the rest of the system. It would be necessary to add a network communicator and serializer to the Model and create a library that would need to be linked to the other applications communicating with the Model.

In the figure 3.1, you can see that the data model and Lua scripting API form the "Model" part of the MVC design pattern, whereas the Qt part of the application together with the `TournamentController` class form the "View-Controller" part. Each new application that will want to communicate with the data model will have to implement the View and Controller patterns.

3.3 Use of the Observer pattern

Observer is one of the behavioral design patterns that were described in *Design Patterns, Elements of Reusable Object-Oriented Software* [4], whose main purpose is to "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically". This is something that is needed in my application — when some data in the data model change, all the depending instances need to be notified to be able to reflect these changes. For example, if a match is added via the scripting API, the graphical front-end has to display a new graphical element that will correspond to this new match object. There are many such places in the code.

The original Observer design pattern, as described in *Design Patterns*, consists of a `Subject` class that provides an interface to attach and detach `Observers` and that has a method `notify()`, which notifies the attached `Observers` of a state change in the `Subject`. The second class that participates in the design pattern is the eponymous `Observer` class. This class has a pure virtual method `update()` that is called inside the `Subject`'s `notify()` method. The book then adds another two classes to this design pattern — `ConcreteSubject` and `ConcreteObserver`. `ConcreteSubject` calls the notify() method whenever that is necessary and ConcreteObserver implements the update() method from the interface of the abstract class Observer.

Modifying the pattern

My approach is a little bit different. Instead of a Subject class that has to be derived from, I used a class called Observable that is added as a member to the class that needs to be observed. The disadvantage of this approach is that I have to create special methods for IObserver (that is how I named my Observer class) attachment in each observed class. An advantage on the other side is that I can have multiple Observables inside one class, which is needed because the only class that needs to be observed in my application is the ObjectManager. The Observable's notifyObservers(int) method then corresponds not to the change of its state, but to a certain signal that may interest the IObservers. The integer parameter of this method identifies the type of the signal.

Because I am able to identify the signal type in the IObservers, I can attach them to multiple Observables and according to the signal type decide what to do in the notify(int) method. Thus the dependency in my adaptation of the IObserver pattern is not one-to-many, but many-to-many.

The references to depending IObservers in Observable are stored in a linked list, because the list needs to be iterated over. I also store pointers to the Observables that the given IObserver depends on, because when the IObserver is deleted, it needs to tell the Observable to delete it from its list of pointers. However, this in turn creates the need to disconnect the IObservers from an Observable if it is deleted, otherwise the IObservers would have a pointer to an invalid Observable object.

This design worked well for a long time, until I decided to disconnect IObservers from Observables as a reaction to certain signals. What happened was that inside the Observable::notifyObservers(int), the Observable was iterating over its IObservers using a list iterator. It called the incriminated IObserver, which decided that it needed to be removed from this Observable and removed itself. However, this caused the iterator to be invalidated and further iteration over the Observable's IObservers was prevented. I solved this problem by setting the pointer to the IObserver to 0 instead of removing the list item. That way, iteration can continue and next time, when a 0 is encountered instead of a valid pointer, the Observable removes that list item safely.

Observation patterns

As the authors of [4] correctly point out in chapter one of their book, design patterns have a disadvantage to them: they often rely on dynamic object composition and the patterns of inter-referencing objects cannot be easily derived from the source code. The dynamic composition patterns of the objects that take part in the Observer design pattern in my application can be seen in the figure 3.2.

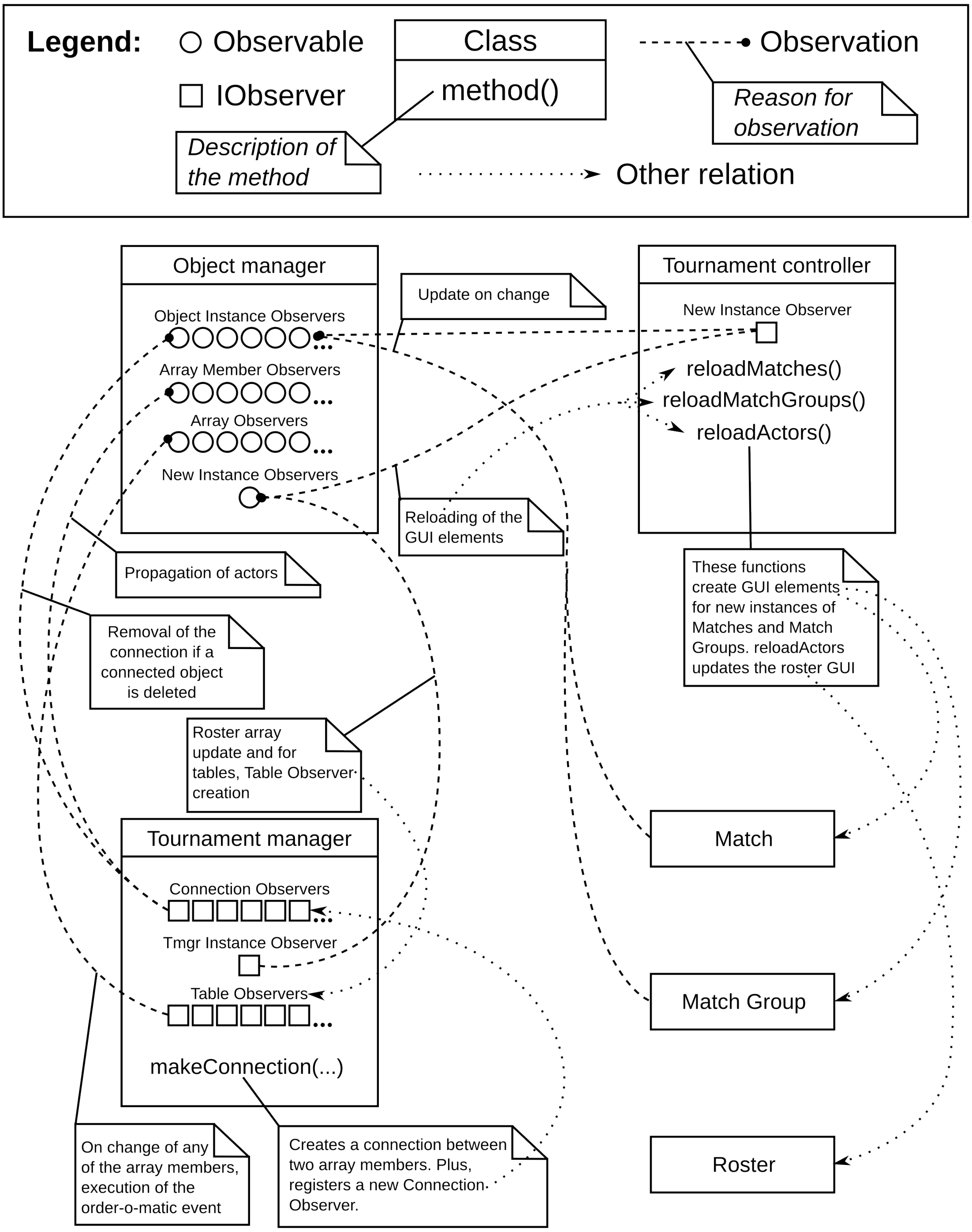

Legend:
○ Observable
□ IObserver
Class
method()
Description of the method
Observation
Reason for observation
Other relation
Object manager
Object Instance Observers
Array Member Observers
Array Observers
New Instance Observers
Update on change
Tournament controller
New Instance Observer
reloadMatches()
reloadMatchGroups()
reloadActors()
Reloading of the GUI elements
Propagation of actors
Removal of the connection if a connected object is deleted
These functions create GUI elements for new instances of Matches and Match Groups. reloadActors updates the roster GUI
Roster array update and for tables, Table Observer creation
Tournament manager
Connection Observers
Tmgr Instance Observer
Table Observers
makeConnection(...)
Match
Match Group
Roster
On change of any of the array members, execution of the order-o-matic event
Creates a connection between two array members. Plus, registers a new Connection Observer.

Figure 3.2: *Diagram of observation patterns throughout the application*

Most of the descriptions in that figure are self-explanatory, but I will describe one of the observation relations in detail, because it is important. It is the one that is described as "Propagation of actors" and leads from the `ConnectionObserver` inside the `TournamentManager` to the `ObjectManager`, where the `ConnectionObserver` is registered as an array member observer.

At the start of the application, the `TournamentManager` looks at all the connections that are stored in the database and for each such connection it creates and stores within itself a `ConnectionObserver` instance. This observer is then registered in the `ObjectManager` to observe changes of a single array member — the source of this connection. When that array member is changed, the `ConnectionObserver` sets its destination array member to the same value. This is how propagation of actors through connections is done.

Note that there are no cycle detectors yet, so you could create a cycle with the connections, which would throw the application into an endless loop, because a new connection always propagates its source as if it has just been changed. As soon as you finish the cycle, the last connection starts propagating its source value along the circle, which never stops.

3.4 Match interconnection GUI

The graphical user interface contains a part of the data model for the purpose of drawing the tournament structure and letting the user change it in an interactive way. This is all done inside a Qt drawing platform. The data is kept in `QGraphicsItem` objects that are part of a `QGraphicsScene`, which is displayed in a `QGraphicsView` widget in the main application window. The `Match`, `MatchGroup` (this corresponds to the table base type in the data model) and `Roster` classes, that were described in the figure 3.2, are an example of such `QGraphicsItems`.

The basic class of the interconnection GUI is `Connectable`. It is a `QGraphicsItem` that contains a numerical identifier and a name. A `Connectable` can have number of outputs and inputs. It is painted onto the graphics scene as a rectangle of the given base color with inputs on the left and outputs on the right side of the rectangle. All the entities that can be connected via connections are of a subclass of `Connectable`. Each such subclass has a unique identifier that is returned by the virtual `connectableType()` method and can be used to determine the class of the `Connectable`.

Connections

Connections (the relevant class is named `PlayerConnection`) are another type of `QGraphicsItem` in the interconnection framework. They have two `Connectable` pointers and contain information about the port type — input or output — and number of the two ports — source and destination — that they connect. In order to be able to draw itself, the connection has to know the positions of its source and destination ports. When the redraw is needed, the connection asks its source and destination `Connectables` about the positions. This is done by calling the `portPosition(...)` method of `Connectable`. This is a virtual method, so the individual `Connectable` sub-classes can re-define the position of their ports this way if needed.

Connections are created using a drag-and-drop mechanism. Clicking on a port of a `Connectable` and dragging the mouse pointer away creates a `MatchMimeData` object that contains information about the `Connectable` and the port being dragged. Upon dropping the mime data object onto the destination port, the destination `Connectable` performs a few checks, orders the source and destination port so that the output port is first and tells the `TournamentController` object to create a new connection.

Scenes

There are more than one `QGraphicsScene`s in the main window and they are stored in a hash-table, indexed by numerical identifiers. The main tournament structure is always in the scene with the index 0. For each table in the data model there is a scene, indexed with that table's numerical identifier. All matches and tables that are grouped in the given table are then drawn in that scene instead of the scene with index 0. The main window contains widgets that can change between the displayed scenes.

Connectables

Apart from the obvious `Connectable`s like `Match` and `MatchGroup`, there are several other types that are mostly used inside `MatchGroup`s.

First such `Connectable` is `GroupInputs` class. This class represents the input array of a match group. It is the same array that is represented by the `MatchGroup`'s inputs. However, `GroupInputs` belong to the scene that has the numerical identifier of the given match group and they can be used to propagate the input values of the match group further inside the group. Class `GroupOutputs` serves similar purpose, only for match group's outputs.

Another `Connectable` is `OrderOMatic`. This is a `Connectable` that represents the ordering mechanism inside match groups. The inputs of the `OrderOMatic` correspond to the input ordering array of the match group and outputs to the output ordering array.

The last `Connectable` is the `Roster`, which is a representation of the global variable that holds all the actors in the data model.

3.5 Incorporating Lua

All the information about the state of the Lua interpreter is saved in a structure called Lua state. This state is created and initialized in the `LuaEnvironment` class. `LuaEnvironment` also loads the standard Lua libraries into the state and registers the objectManager and eventManager with the use of a C++ wrapper that I created. A wrapper is needed because the Lua C API is able to register a C function (that follows a pre-defined protocol) into the Lua state, but not a method of a C++ class.

C++ wrapper

The main class of the C++ wrapper is `LuaScriptApi`. This class provides methods for registration of C++ objects and their methods into the Lua state as well as methods for inspection of the registered classes and methods.

To be able to understand the registration mechanism of the wrapper, one needs to first know several things about the Lua C API. One needs to understand how the Lua C API communicates with the Lua interpret, how the C functions are registered into Lua in the plain C API and what the protocol that they must follow looks like.

Communication with Lua is done by the use of a Lua value stack. Values are pushed onto the stack in the C code and the Lua C API functions work with these values. To register a C function into Lua, pointer to the function has to be pushed onto the stack and can then be saved into a variable inside Lua using the C API. The function has to have a single parameter of type `lua_State*` and its real parameters can be found on the stack at the time of the execution. Return values are pushed onto the stack as well and the return value of the function in C is an integer value that should be set to the number of Lua return values pushed.

To register a method of a C++ object into Lua using the wrapper, a string identifier has to be set for the object first. Then, a registration method of the `LuaScriptApi` has to be called with the name and signature of the method that is to be registered. The information is saved inside the `LuaScriptApi` object for argument type-checking purposes. After that the `proxyMethod` of the `LuaScriptApi` is registered into Lua with two upvalues (that is values that can be registered with the function and the function can later access them): identifier of the object and name of the method. The function is saved inside Lua under the method name in a table named after the C++ object.

When called, the proxy method looks at the two upvalues and finds the corresponding method information in `LuaScriptApi`. It checks the argument number and types according to the method's signature, creates one list out of the arguments and another one for the results and calls a method `callMethod` of the registered C++ object (pointer at the object was also part of the saved method information). The `callMethod` method then looks at the name of the method to be called and calls it, giving it arguments from the list of arguments. Then, it modifies the result list (both lists were passed to the C++ object from the `proxyMethod` by reference) and returns.

The `callMethod` method is a pure virtual method of `ILuaScriptable` abstract class. The method has to be defined in the classes derived from `ILuaScriptable`. For the purposes of method name comparison, `ILuaScriptable` contains a trie (see Appendix B) that can be utilized to be able to use a switch command in the `callMethod`.

Events

Events are handled inside the the `EventManager` class. They are loaded from a directory and a new Lua table is created that contains them. The table is then saved as `tmgr.events`.

Each event is a Lua function and arguments of the event are accessible as local variables of the function. This is achieved by saving the all of the event's arguments in a local table `args` by adding the following code as the first line of the event's body:

```
local args = {...}
```

Each argument is then converted to a line that is added to the beginning of the function's body in the following manner:

```
local home_goals = args[1]
```

In the previous example, `home_goals` has to be substituted by the name of the argument and the `args` table is indexed with the argument's order in the event definition.

After the argument definitions, the body of the event is appended and the whole code is loaded into Lua as a function and saved into the event table under the name of the event.

3.6 Database organization

This section will describe all the database tables that the application uses to save its data.

Global variables

Each global variable's name is saved in the **var** table together with its type. For each tournament manager type (see section Types in Appendix A), there is a table named **var_typename**, where *typename* is the name of the given type. This table contains values of global variables of that type indexed with the variable's name.

Arrays

Instantiation of a new array consists only of assigning a unique reference number to that array. The highest used reference numbers for arrays of non-array types are saved in a table named **array_instance_numbers**. When a new array is created, the corresponding record in this table is incremented by one and the incremented number is assigned to the new array. Variables of array type are represented only by their array reference number (also called array id) in the database.

Values of all arrays are saved in tables named **array_typename**, where *typename* is a name of a non-array type. The values are indexed by the reference id of their array and by their ordering number — their index inside the array.

Object model variables

Each new sub-type in the object model is saved into a table named **types** together with a number that indicates, which of the four base types is the base for the new sub-type.

Two further tables exist for each sub-type. One of them is **meta_typename**, where *typename* is the name of the new sub-type. This table contains two columns: **colid** and **coltype**, which contain name and type (represented by a number) for each member variable in the sub-type. The second table is named **table_typename**, where *typename* is again the name of the sub-type. This table contains records for each instance of the sub-type and its columns correspond to the member variables of the sub-type.

A record for each instance is also saved into one of the four tables that are named **base_basetypename**, where *basetypename* is name of the base type of the instance — actor, match, table or object. These tables contain information about the sub-type of the instance, so that the system knows, in which of the **table_typename** tables it should look for the instance's values. They also contain mapping of the instance names to numerical reference identifiers that are used as keys in the **table_typename** tables. These identifiers are also saved in variables that are of one of the four base types (so, when assigning an actor instance to a variable, for example, only its identifier is saved inside the variable), similar to the arrays.

Connections

Connections are saved in a separate table **connections**. Columns **from_arr** and **to_arr** contain array identifiers of the source and destination arrays, whose members this connection links. The member indices for the source and destination array members are saved in columns **from_nr** and **to_nr**, respectively. Columns **from** and **to** are identifiers of the source and destination objects. The type of these objects can be derived from the **type** column. All the possible connection types are listed in the file `src/connectionType.h` of the source code.

3.7 Important classes

This is just a quick overview of what some of the classes in my application do. It is not meant as a complete source code description. If more thorough understanding of the code is required, reading the source code documentation or the source code itself is recommended. Some of the classes that were already mentioned in the preceding description are not listed here in order to avoid repetition. The classes that are listed here are alphabetically sorted. After the name of each class, the path to the file where the class header can be found is provided.

Arguments `src/arguments.h`

Parses the command-line arguments and stores their values for further querying. The allowed parameters and flags have to be set in order for the class to recognize them. Else, the parsing method returns `false` to denote failure.

DatabaseConnector `src/databaseConnector.h`

Handles the connection to the database and provides an intermediate layer between the database and the application. In theory, this is the only class that would have to change when changing the underlying data back-end, although in practice use of constructs specific to SQL wasn't avoided in a few places in higher layers.

Event `src/lua/event.h`

A container class for event information. It also handles putting the event body together with all the implicit code that needs to be prepended before the main code.

EventDialog `src/qt/eventDialog.h`

GUI-related class that creates a dialog box with input widgets for each event so that they match the event's arguments.

EventManager `src/lua/eventManager.h`

Event manager loads events into the Lua environment and allows them to be executed from the outside of Lua.

GuiController `src/qt/guiController.h`

A helper class that offloads some GUI modification tasks from the main window.

Logger `src/logger.h`

A class that takes care of the output of debugging information, error messages and standard messages. Unlike the other classes, it is globally referable instead of being referenced in all classes that use it, simply because almost any part of the application uses it.

LuaHighlighter `src/qt/luaHighlighter.h`
Gives Lua highlighting capabilities to the user interface to make writing Lua code easier.

MainWindow `src/qt/mainWindow.h`
Creates the main window of the application, handles click events of its buttons, manages `QGraphicsScene`s that represent the individual match groups and also indexes all the `Connectable`s, so that tournament controller can check, whether a `Connectable` needs to be created or just reloaded.

ObjectManager `src/objectManager.h`
Object manager provides the object model abstraction to the database, as per the design. It allows for creation of new sub-types and their instances, can read and modify their data and contains methods for registration of various types of observers.

ObjectMetadata `src/objectMetadata.h`
Contains information about an object sub-type such as its name, base type and member data (or columns). It has to be provided to the `ObjectManager` to create new types. The object meta-data should not be created manually, but using the `TournamentManager` that adds all the implicit columns that are needed in the tournament manager extension of the object model.

StringMatcher `src/lua/StringMatcher.h`
A trie-based string-to-integer mapping class.

TmgrType `src/tmgrType.h`
An enumeration type of all the possible variable types in the object model. Several utility functions for parsing and various conversions are also included in the source file.

TournamentController `src/tournamentController.h`
The tournament controller controls loading of data into the main window and also takes care of refreshing the data if it is told by the Observers that refresh is needed. It also contains a few methods related to connection creation.

TournamentManager `src/tournamentManager.h`
Provides the tournament manager extension to the object model. It takes care of the connection creation, running of the Order-o-matic and input or output number modification.

Chapter 4

Application description

When talking about the use of the system, it is important to distinguish between several different perspectives. The information that a user of the application needs to know varies depending on the role that the user takes. My description of the system takes this into account and each section of this chapter looks at the application from a different standpoint. First, the responsibilities of the administrator and programmer are listed and then the use of the application is described from an ordinary user's point of view.

4.1 Administrator

Duties of the administrator of the tournament management system include compilation and installation of the system and preparation of the database. Let's look at these duties closer.

Compilation

Before compiling the application, the administrator should install development versions of the needed libraries. The operating system documentation should describe how to do this. The required libraries are:

libc Standard C library

libc++ Standard C++ library (STL)

dl Dynamic linking library

qt4 [17] Qt 4 application and UI framework. Before compilation, the QTDIR environment variable should be set to the installation directory of Qt. Some systems do not set this variable.

mysql [9] MySQL C API library

mysql++ [10] MySQL++ C++ wrapper

lua [11] Lua language library

The default build system is SCons [13], so it should also be installed along with a Python interpreter. To compile the application, it should suffice to simply run ‘`scons`’ in the directory where the source code was unpacked (that is one level up from the `src` directory). To also generate debug information and force the program to output debug messages to a text file, the `debug` compilation variable should be set to 1, so the compilation command would be ‘`scons debug=1`’.

Installation

Installation is trivial. It only consists of copying the executable into a desired directory and setting the PATH environment variable, so that the system can be run from the command line.

Requirements on the hardware

The system can be run on a single computer, provided that a database server can be run on it too. If that is not the case, the communication with the database can be realized over a network. In that case, networking hardware needs to be included in the requirements. If a presentation is required, the network will be probably necessary anyway to connect the presenting computer to the database. The presenting computer also has to have a data projector or at least a large enough display so that the presentation is clearly visible to everyone at the event. Of course, custom solutions like displaying the score on a huge digital display are possible as well due to the open nature of the database.

Preparation of a new tournament

Every tournament needs to load its definition files from two directories. One is named `events` and contains event definition files and the other one is named `types` and contains type definition files. When the application is run, these directories can be either specified with a parameter or the application searches for these two subdirectories in the current working directory. The administrator should decide about the location of these directories.

The administrator should also create a new MySQL database for the application and grant rights to a user that will be used to run the application.

4.2 Programmer

A programmer in the context of my application is a person with an analytic mind and at least some programming experience. He or she should think about all the statistics and information that will need to be kept in the system, create an object model that will contain all the needed data and analyze the events that will be needed to modify the data.

The programmer should not forget to bear in mind the structure of the tournament when creating the analysis, because a specific structure of a tournament can create need for additional types and events (especially match group types with their ordering events). Apart from the obvious score setting and similar events, data entry events will also be needed, such as naming events or events for registration of contact information of the individual players.

Defining new types

After the analysis is finished, the programmer has to define new types according to the analysis. Each type is defined in a separate file in the `types` subdirectory. The file can have any name, but a good convention is to give it the name of the type that it defines. Each type definition file has the following structure:

New sub-type name First thing in the file should be the name of the new type. It must contain only alphanumeric characters or underscores and it should begin with an alphabetic character. It also must not be longer than 54 characters.

Base type name Next is the name of the base type for this new sub-type. This is one of the four: `actor`, `match`, `table`, `object`.

Member variables Then, for each member variable of the new type, write its tournament manager type name (see Section Types in Appendix A) followed by the variable name.

When all types have been defined, the programmer should load them into the database. This is done by running the application (see Running the application in Section User) with one additional parameter: `--types`. When set, this parameter tells the application to load type definitions from the given directory. If the database is not empty, it can be cleared with the `--delete-everything` parameter. These two parameters can be combined to reload the type definitions. But be careful — this also deletes all the data that could have been in the database.

Defining events

Similar to the definition of new types, each event has its own file in the `events` subdirectory. Again, the file can have any name, but giving it the name of the event is recommended. The structure of event definition files is as follows:

Event name First comes the event name. The name should contain only alphanumeric characters or underscores and should not begin with a digit.

Number of arguments The event parser must know how many arguments will follow, so a number is required here.

Arguments For each argument, write the type of the argument (tournament manager type, that is) followed by the argument's name.

Event body Everything that follows after the argument definition is considered Lua code and will constitute the event body.

Defining ordering events

Definition of ordering events is similar to normal events with a few distinctions. The event name has to be `order_matchgroup`, where *matchgroup* is the name of the match group sub-type that we are defining an ordering event for. Also, all ordering events must have only one argument of type `table`. At the execution time, this argument will contain reference to the match group that has to be ordered.

Creating scorekeeper's guide

In most of the cases, the tournament programmer will not be using the system himself or herself, so an important part of the programmer's work should be creating a short guide where the basic idea behind the system is described. Even more important is to include descriptions of all the events in the system with their arguments and their purpose.

The guide should also contain description of the typical work-flow when using the tournament (Something like "At the beginning, you do *XYZ* for each team. Then, you start the tournament by clicking *ABC* and handle the individual matches with *FOO* and *BAR*..." and so forth), so that even an uninformed user can quickly start managing the tournament.

4.3 User

A user is anyone who will work with the running application, so user categories like tournament planner, referee, statistician or scorekeeper all belong here. I will describe how the whole application can be used and leave up to the reader to decide which part of the description is relevant to which user category.

Running the application

The application has several parameters and flags that can be set. Flag `--help` or `-h` lists them all. This is a list of the other parameters:

--db=NAME This mandatory parameter has to be set to the name of the database scheme that should be used.

--user=USER This parameter is also mandatory and sets the user-name used in the database connection.

--pass=PASS Also a mandatory parameter that sets the password used when connecting to the database.

--server=HOST This parameter defines the ip address or DNS name of a computer where the database server is running. Default value is `localhost`.

--events=DIR When set, this parameter loads events from the given directory. If not set, the application looks for the `events` subdirectory of the current working directory.

--types=DIR When set, this parameter loads type definition files from the given directory. This needs to be set only when initializing the tournament.

--delete-everything Deletes everything in the given database, so that it can be re-initialized. This parameter should be used with care.

Creating a shortcut for running the application is recommended. That way, a less proficient user can run the application without knowing anything about command line and arguments.

Main window description

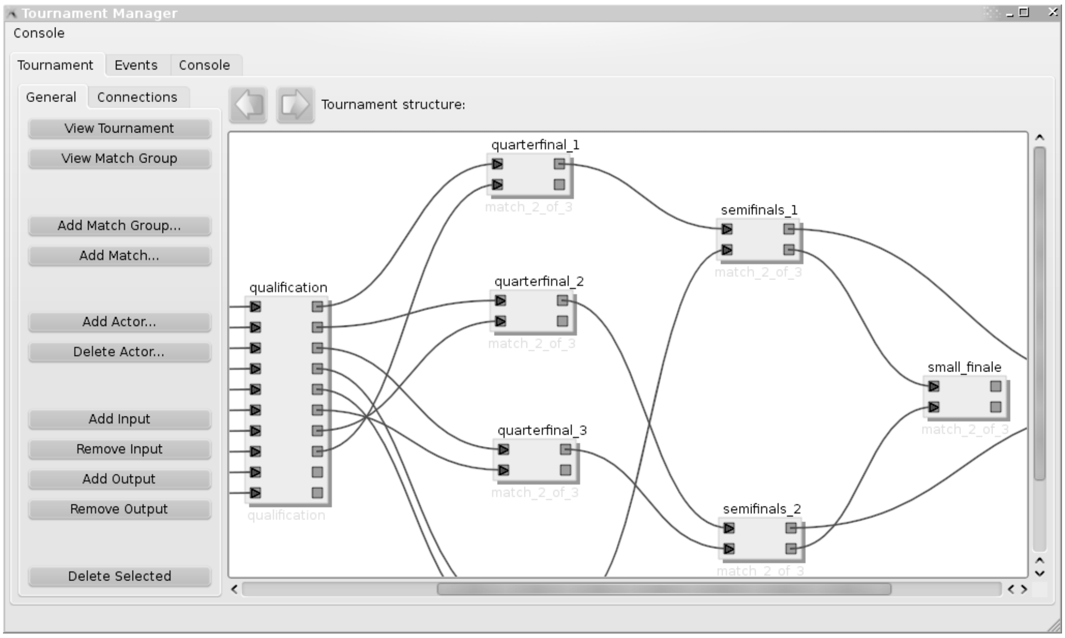


Figure 4.1: *The main window of my application*

The main window of the application can be seen in the figure 4.1. The user interface is divided into three tabs: **Tournament**, **Events** and **Console**. The **Tournament** tab can be used to create new actors, matches or match groups and allows them to be connected. The **Events** tab serves for execution of events and the **Console** tab can be used to debug the application by editing and executing Lua code.

Tournament tab

The right side of the **Tournament** tab contains the tournament structure viewer. When the application is started, it shows the uppermost level in the match group hierarchy — that is the whole tournament. If there is a match group in the tournament, its contents can be displayed by selecting it in the viewer and clicking the **View Match Group** button. The viewer will display structure of the match group. To get back, either the **View Tournament** button can be used or the navigation arrows at the top of the viewer. The arrows can switch back and forth between recently displayed scenes.

The viewer also serves for manipulation with the connectables and most importantly for connection creation. New connections can be easily created using the drag-and-drop method by dragging from one of the two ports to be connected and dropping over the second one.

Buttons in the **General** tab will now be described. **Add Match Group**, **Add Match** and **Add Actor** all work in the same manner. They show a dialog box that lets the user select the desired type of the new object from a list that shows all the defined sub-types of the given base type. The dialog box also contains a text edit line, where a name for the new object should be entered. When no name is given, the system will generate one that will consist of an underscore followed by the numerical id of the new object. **Delete Actor** is pretty self-explanatory. It lets the user select one from all of the actor instances and then deletes the selected instance.

Another group of related buttons serves for modification of the input or output number of matches and match groups. To change the input or output number, the user selects one or more matches and match groups and clicks one of the buttons **Add input**, **Remove input**, **Add output** or **Remove output** and the selected connectables' number of inputs or outputs is changed.

The **Delete selected** button deletes all selected matches and match groups.

The **Connections** tab has several features related to connection manipulation. **Multi-connection helper** lets the user connect all of the possible inputs or outputs between two `Connectable`s. To do this, one of the two `Connectable`s has to be selected and then one of the two buttons **Set from** or **Set to** clicked and the `Connectable` is saved as one end of the multi-connection. Then the second `Connectable` has to be selected and the second button clicked and the connections are automatically created. The blank space under **Connection remover** can be used as a drag-and-drop target to remove all connections from the port from which the drag-and-drop originated.

Events tab

If the `--events` parameter was set when running the application, each loaded event shows here as a button that displays an event dialog box. This dialog box contains input elements for all of the event's arguments. Each event button is movable by clicking the small blank square button on the left, so the buttons can be re-arranged at will.

Some buttons also have automatically created shortcuts, so that the user does not need to move hands from the keyboard. The shortcuts can be activated by pressing the `[ALT]`+`[key]` combination, where *key* corresponds to the underlined letter on the button label. If there is no such letter, it could not be chosen from the letters of the button label without colliding with other shortcuts.

Console tab

This tab provides a console that can be used to run simple Lua commands as well as complex scripts. The scripts are written into the left part of the tab, while the right part serves as an output window. The **Execute** button executes the script that is on the left and any eventual error messages are displayed in the status line at the bottom of the tab. The menu group **Console** contains commands for loading and saving the scripts and for clearing the script and output windows.

Chapter 5

Case study

This chapter describes all the steps that are needed to create a tournament on a specific example — the Eurobot 2010 competition [6]. The setup was successfully tested on the real data from the competition. In addition to using the application itself, an example is given at the end of this chapter that shows the possible solution to presenting the data and generating the needed paperwork like match sheets. All the files needed to recreate the example can be found on the accompanying CD.

The scoring system of Eurobot competitions is not trivial and by using it as a case study, I show the power of my system that stems from the use of a scripting language to handle events. The implementation of qualification and playoff phases is also noteworthy, as they resemble those used in other tournaments and competitions.

5.1 Administrator's work

Situation

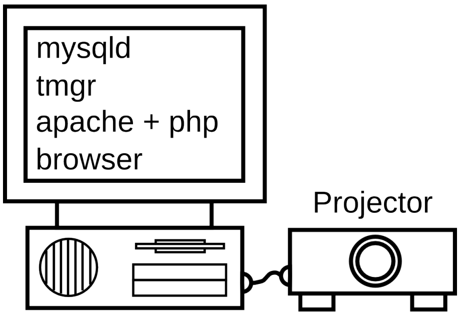


Figure 5.1: *Situation in our case study*

I will skip the software installation as there is not much to describe. As for the hardware, the Eurobot committee have a projector at their disposal and there is usually only one person filling in the tournament data, so I decided to have everything on one machine — the database server, the tournament manager and presentation of the data as well. Presentation will be realized using html pages generated by php, so if the situation in the venue would require it, it would be possible to connect a dedicated presentation computer to the main computer. The whole situation is clear from the figure 5.1.

MySQL setup

Let's presume that the MySQL server is installed and running. We now need to create a MySQL table and a user that will have all the rights to this table. This can be done for example in the MySQL Administrator application. In this study, I named the table "tmgr" and the user "tmgr_host" with the password set to "tmgr_host". Note that this is not a strong password at all, but in our case, we do not need to worry about anyone abusing or changing our data. In case of a bigger system or if running on a shared network, a stronger password would be necessary.

Other setup

All the event definitions and type definitions are located in a directory named `eurobot` (the exact location is not important). In this directory, two shortcuts will be created that run the application with different parameters. The first shortcut is intended to do the initial setup, as it erases all the data and reloads the types. It could also be called without the `--delete-everything` parameter, but then it could not be used for re-initialization. The second one is then intended for normal use. They execute the following commands:

Initial setup:

```
tmgr --db=tmgr --user=tmgr_host --pass=tmgr_host --delete-everything\
   --types=./types
```

Normal run:

```
tmgr --db=tmgr --user=tmgr_host --pass=tmgr_host
```

Also, to enable the presentation using php, a web-server has to be installed with a php module and be able to access the database.

5.2 Programming the logic

After everything has been set up correctly, we can begin with definition of our types and events. However, before we delve into the programming itself, we need to make a short analysis of all the requirements on our scoring system and think about how we will implement them. I took the rules that would be relevant to the scoring system from the official Eurobot 2010 rules [7].

Relevant rules

The basic premise of the competition is that each competing team builds a robot to face other teams' robots in a series of friendly matches (that means no destructive behavior). All matches follow the same rules. Each match takes 90 seconds and the robots must collect as much fruit, vegetables and corn as they can during this time without bumping into the other robot or otherwise preventing the other team to score. The rules are relatively complex, but the subset that interests the programmer of the score-keeping system is small:

Team colors In each match, two teams compete and their sides are colored yellow and blue.

Points The robots can collect three types of objects on the play-field, each having a different scoring value. **Tomatoes** score 150 points each, ears of **corn** 250 points each and **oranges** 300 points each. These points constitute the team's base score in a match.

Penalizations They are subtracted from the team's base points in a match, each one subtracting 20% of the base score or 150 points, whichever is larger. The points can go below zero this way.

Bonus points At the end of a match, the resulting points are compared and the winner is declared. The winning team gets a 200 point bonus. Loser gets 50 points. If the teams tie, they both get 100 points. If a team has zero or less points, it loses automatically even if it has more than the opponent. In case of a so-called scratch (disqualification from the match), the incriminated team gets 0 bonus points and the opposing team wins (unless it was also scratched or had a score of zero or less)

Disqualifications They will be either from a single match (scratch) or from the whole competition. In the latter case, the team will be automatically scratched from each subsequent match and will not be able to play in the final rounds.

Warnings They will be issued to a team for various forbidden actions and will be recorded in the score-keeping system for future reference. Repeated misconducts of the same kind will result in in-match penalties.

Qualification rounds The qualification will consist of several rounds so that each team gets to play at least five matches. At the end, the top eight teams with the highest score will continue to the playoff rounds. In case of a tie, the bonus points will be used to determine the better team. If there is still a tie, an extra tie-breaking match can be issued to count towards extra points that will decide which team is better.

Playoff rounds The playoff matches will be played in the best-of-three format, so the team will have to win two times in order to stay in the competition. If a match ends in a tie, double defeat or double scratch, it is repeated. If the repeated match also ends in a tie, double defeat or double scratch, the team with better qualifying position is the winner.

The basic approach

Now that we have all the requirements written down, we can start deliberating about the object types and events that will be needed and how everything will be put together. Most of the score-keeping will be pretty straightforward — there will be a current match that will be displayed in the presentation (if simultaneous matches would be needed, an other method would have to be used), a scoring event that will set the score of the current match and an event that will finish the match. The match-finishing event will add the score to both teams and assign the teams to the outputs of the match.

Finishing playoff matches would have to be different, because their score does not add to the team scores anymore. We could add another match type, but it will be better to have a single flag that will say whether the match is a qualification match or a playoff match. One more match flag will be needed to denote an extra match. This is the tie-breaking match in case of equal points after qualification rounds. When the match finishing event is called, it will have a few flags to denote the match type and will finish the match according to these flags.

The playoff best-of-three matches could also be implemented as another match type, but since the individual matches in the playoff rounds do not differ from the qualification matches, we would be creating unnecessary duplicate types. Instead, we will have a match group that will contain the individual matches and will determine who is the ultimate winner depending on their outcomes.

I will add one extra feature to the system — a possibility to erase matches. It sometimes happens that the scorekeeper inputs the data incorrectly. While this can be solved by backing the database up and resetting it to a previous state, a more consistent and elegant solution would be to provide events that would take care of this. There will be an event that will be able to erase the data from a given match, according to its type. If the match is finished, the event will correctly subtract the match's points from the participating team points, whereas if the match is still running, this event will only reset the data.

Object types

Eurobot only has one type of teams and matches, so there will be a single match type named `eurobot_match` and a single actor type named `eurobot_team`.

The `eurobot_team` type will contain a name for the description of the team, variables to hold the points, bonuses and extra points from the qualification phase, number of qualification matches played (this is a good statistic to show in the qualification ranking) and the qualification rank. It will also need to carry disqualification information and information about the issued warnings.

The eurobot_match type will contain more information. It will have to remember the number of items collected by each team, each team's base points, bonus points and final points, the number of penalties issued against each team and also all the warnings given to each team. It also has to contain scratch information for both teams and because we will handle different types of matches differently, also a few flags to denote match type.

The rest of the types are match group types. The match_2_of_3 type will contain information about the number of wins of two teams, so its definition is short. Another match group type is team_permutation. The ordering event of this match group will create a pseudo-random permutation seeded with the given number, so that the team seeding is random. Other match group types serve only for grouping purposes and have no member data. These are qualification and qualification_round.

All contents of the types subdirectory are listed below (and can be found on the accompanying CD):

eurobot_team.txt:

```
eurobot_team
actor
string name
int points
int bonuses
int extra
int qualification_order
int qualification_matches
bool disqualified
bool violent
bool shutdown
bool damage
bool unfair
```

eurobot_match.txt:

```
eurobot_match
match
int t1_oranges
int t1_tomatoes
int t1_corns
int t1_points
int t1_bonus
int t1_final
int t1_penalties
bool t1_scratch
int t2_oranges
int t2_tomatoes
int t2_corns
int t2_points
int t2_bonus
int t2_final
int t2_penalties
bool t2_scratch
bool t1_violent
bool t1_shutdown
bool t1_damage
bool t1_unfair
bool t2_violent
bool t2_shutdown
bool t2_damage
bool t2_unfair
bool finished
bool final_match
bool extra_match
```

team_permutation.txt:

```
team_permutation
table

int seed
```

match_2_of_3.txt:

```
match_2_of_3
table
actor actor1
int wins1
actor actor2
int wins2
```

qualification.txt:

```
qualification
table
```

qualification_round.txt:

```
qualification_round
table
```

5.3 Events

Event definitions will be slightly more complicated, but not much. Note that although the files are named so, they are strictly speaking not Lua code (because the first few lines are not in Lua).

Due to their length, not all event definitions are shown. The complete code of all event definitions used in the case study can be found on the accompanying CD.

Initialization

Let's start with the initialization. A global variable `current_match` will be needed and the initialization event will create it. That is all it will do.

init.lua:

```
init
0
--Initialization event that creates globals
omgr.newGlobal("match", "current_match")
```

After initializing the tournament and creating the tournament structure, we will want to set team names and the `name_team` event will do exactly that.

name_team.lua:

```
name_team
2
actor team
string name
team.name = name
```

Managing matches

The `start_match` event only sets the `current_match` global variable. There is also a fail-safe to avoid starting a finished match multiple times and an `if` statement that ensures that the tournament has been initialized. In order to restart a match, it would have to be erased first (see further).

start_match.lua:

```
start_match
1
match match
if match.finished then return end

if tmgr.globals.current_match == 0 then
  tmgr.events.init()
end

tmgr.globals.current_match = match
```

Follows the core of the system — the scoring event `score`. First, this event does a few checks and then the scoring begins. Disqualified teams are automatically scratched from the match, base points and bonus points are calculated for each team and the winning team is determined.

Some of the longer sections are skipped in the code of this event. Such sections are denoted by `[...]` in the code.

The event has to be run even if no team has scored any point, because it has to set the bonus points. Note that the winner is set only in case of no tie.

score.lua:

```
score
10
int tomatoes_blue
int oranges_blue
int corns_blue
int penals_blue
bool scratch_blue
int tomatoes_yellow
int oranges_yellow
int corns_yellow
int penals_yellow
bool scratch_yellow
if tmgr.globals.current_match == 0 then
  tmgr.events.init()
end

local match = tmgr.globals.current_match
if match == 0 then return end

local blue = match._in[1]
local yellow = match._in[2]
local points, penal, p1, p2

--Automatically scratch disqualified teams
if blue.disqualified then scratch_blue = true end
if yellow.disqualified then scratch_yellow = true end

--Calculate blue points
[...]
match.t1_points = p1

--Calculate yellow points
[...]
match.t2_points = p2

--Calculate blue bonus
[...]

--Calculate yellow bonus
[...]

--Calculate final outcome and determine the winning team
match.t1_final = p1 + match.t1_bonus
match.t2_final = p2 + match.t2_bonus

--Only set the winner if there is no tie
if match.t1_final > match.t2_final then
  match._out[1] = match._in[1]
  match._out[2] = match._in[2]
elseif match.t2_final > match.t1_final then
  match._out[1] = match._in[2]
  match._out[2] = match._in[1]
end
```

The previous event can be run multiple times, because it only sets the match data. When the match has ended and the correct score has been set, the event `finish_match` has to be run in order to reflect the match's outcome in the rest of the tournament. The event has two parameters that determine the match type. After doing the necessary checks, the event adds the match points to the teams that earned them. If the `extra_match` flag is set, the points are not added to the teams' qualification points, but to their extra points instead. Whereas, if the `final_match` flag is set, no points are added at all, because only the winner of a final match is important.

After adding the points, the event looks at the parent table and if it is of type match_2_of_3, the win of this match's winner (if there is one) is added to the win count in the table data. An ordering event of the parent table is then called so that it can correctly set its outputs if this win is the second win of the team.

Another event is erase_match. This event checks, whether the match has been finished already. If it was, the erase_match event subtracts the match points from the data of the participating teams. Regardless of whether the match has been finished, the event also sets all its data members to default values — that is either to 0 or false.

The ordering events

We can now manage matches and keep the scores. We should define the events that will order the match groups according to these scores.

The ordering event order_qualification sorts the teams in the qualification table into the correct order and sets their data member qualification_order accordingly. First, a list is created that contains all the input teams. This list is then sorted using the given sorting function that follows the ordering rules that we defined in our analysis. The sorted list is then traversed one more time to determine the ranks of all the teams and the output array is filled with the input teams in the new order.

Ordering event order_match_2_of_3 only checks its member variables that were set by the child matches. If it finds that one of the teams has won enough times, it sets the outputs correctly.

The last ordering event is order_team_permutation. Together with the seed event, these two events ensure the pseudo-random seeding of teams into the qualification matches. A seed is set to an arbitrary number using the seed event and the order_team_permutation event is called automatically to permutate the teams.

order_team_permutation.lua:

```
order_team_permutation
1
table perm
local out_num, list, current_output, it, _in, _out

_in = perm._order_in
_out = perm._order_out

out_num = perm._out_num
list = {}

--Initialize the list with input teams
it = _in.iterator
while it.current ~= 0 do
```

```
  --table is a Lua library object that contains table
  --manipulation functions
  table.insert(list, it:get())
  it:next()
end

--Now, randomly remove teams from the input list and
--write them into the outputs
math.randomseed(perm.seed)
current_output = 1
while table.maxn(list) ~= 0 and current_output <= out_num do
  _out[current_output] =
    table.remove(list, math.random(1, table.maxn(list)))
  current_output = current_output + 1
end
```

Disqualifications and warnings

The last rule that needs to be implemented are disqualifications and warnings. Disqualification is easy, because is consists only of setting or un-setting the disqualification flag in a team. Issuing warnings is also straightforward — we just set the corresponding flags in the team data and if a match is running, also in the match data. A problem comes when we want to erase the warnings. If we remove the warnings from a player, they will still remain in the match (we have no way of knowing in which of the matches the warning was issued). If we erase them in the match, we cannot remove them from the player because the player could have had the same warning from an earlier match as well and it should not be erased.

However, we only need to save the warnings for future reference, so we can afford to sacrifice a little of our system's robustness for the sake of simplicity here. Thus the warning data will be erased in the `erase_match` event and erasing warnings from the player will be handled by a special event called `erase_warning`.

5.4 Creating the tournament structure

Finally, it is time to start the application with the initialization parameters and plan the tournament. There were ten teams in the 2010 competition, so we will add ten actors of type `eurobot_team` and give them either some meaningful names, or simply “team1”, “team2” and so on. Note that these are only the instance names. To set the real team names, we would have to run the `name_team` event, but we will run events after we have prepared the whole tournament structure so as to stay consistent.

The overall picture

The tournament will have a `team_permutation` table that will permutate our teams. The permutation will then be fed into a `qualification` table that will itself contain five interconnected `qualification_rounds`. The top eight teams from the qualification table will then continue into the elimination phase that will consist of four quarterfinal matches, two semifinals, small finale and finale. All these matches will be realized as `match_2_of_3` match groups. A better understanding of the interconnection of the tournament elements can be obtained from figure 5.2.

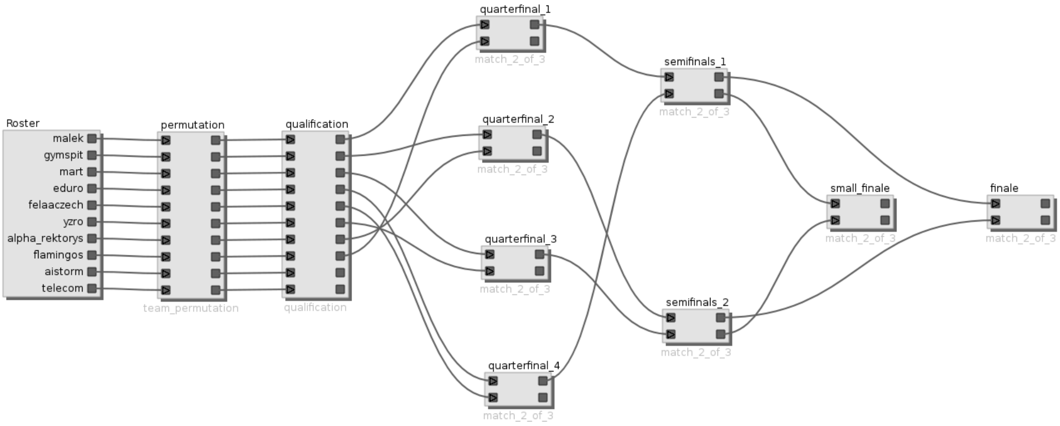


Figure 5.2: *The top-level structure of the tournament*

Inside the tables

The permutation table will be very simple. All its inputs will be fed into the Order-o-matic and from there they will continue to the outputs. The ordering event will handle the permutation for us. The interconnection is pictured in the figure 5.3.

The inputs of the qualification table have to be connected to the Order-o-matic and then to the output. We also need the same teams in each of the five qualification rounds. We will use a little trick here so that the connections look more tidy and do not cross each other as would be the case if we connected the group inputs to inputs of each qualification round. We connect the inputs to the first round, then we connect the outputs of the first round to the inputs of the second round and so on, until we have connected all rounds into a chain that leads to the Order-o-matic. Each round will connect its inputs to the outputs inside itself, so it will be equivalent to connecting everything to the inputs of the whole qualification table. The situation in the qualification table can be seen in the figure 5.5.

Each qualification round will contain four matches that will be connected to the table's inputs in such a manner that they form matches of a round-robin tournament. The outputs will not be connected anywhere because we will not need to propagate the winners of the individual qualification matches anywhere. The Order-o-matic

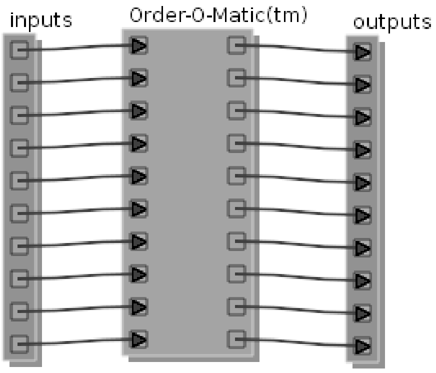


Figure 5.3: *Inside the team_permutation table*

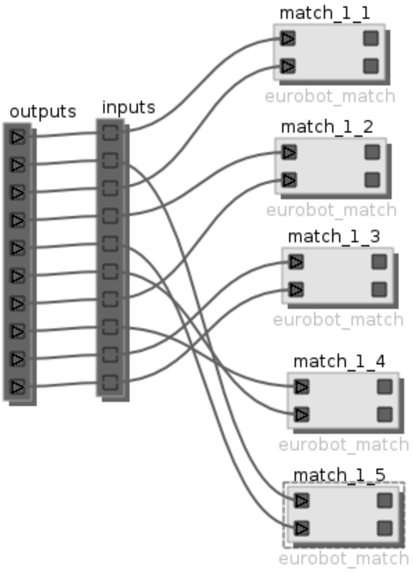


Figure 5.4: *Situation inside a qualification round*

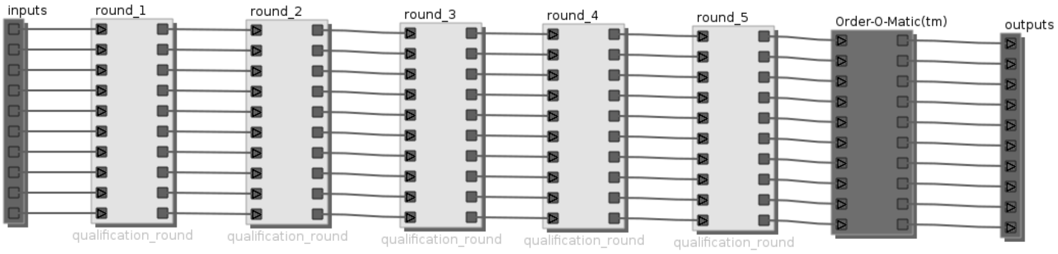


Figure 5.5: *Connections inside the qualification table*

will also be left unused here. Look at the figure 5.4 for reference on the situation inside a single qualification round. Note that the connections to the matches will look different in each round or else same teams would play against each other in each round.

On-the-fly modifications

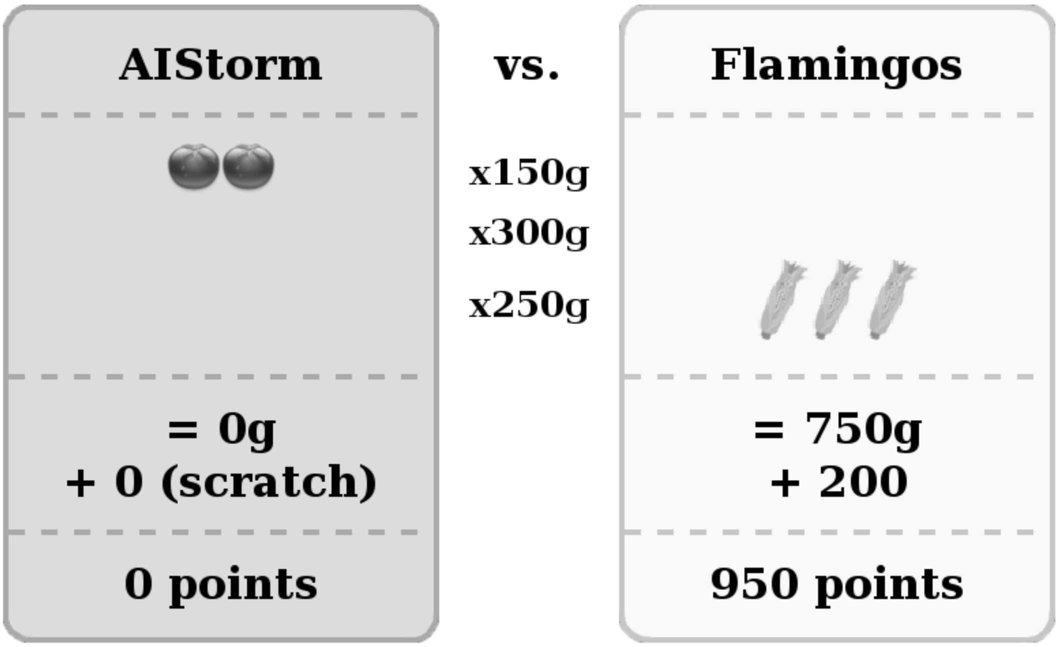
The system allows the user to modify the tournament on-the-fly easily. Reducing or increasing the player number is very quick using the graphical interface. Only the matches need to be modified inside each round to match the new number of contestants. Adding extra match is one click of a GUI button and connecting its two inputs.

5.5 Presentation

Presentation of the data is not handled by the application, but with the knowledge of the database structure, it is easy to implement by reading the data from the database. The separation of the data back-end layer from the rest of the application has the advantage of being able to use a system of choice to read and process the data. For this case study, I chose php-generated html pages. All the php files can be found on the enclosed CD.

Because reading the data from the database manually would be cumbersome, I created a simple php class called `TmgrViewer` that simplifies this task. In its constructor, it connects to the MySQL database and maintains the connection during its lifetime. It can then be used to get globals, instance data and array values.

I created four php pages to show various data from the tournament. The first one displays the progress of the ongoing match in a nice graphical manner suitable for a projector (figure 5.6). Another page shows the team rankings in the qualification phase (figure 5.7). The other two pages are intended to be used by the referees and the scorekeeper. One shows the list of all matches and their status for the reference of the organizers (figure 5.8). The match list page contains links to the last page that generates so-called match sheets for each match (figure 5.9). The match sheets are used by the referees to write down the match report (results, notes, etc...) of each match and are used as a backup in case of failure of the electronical system and also as an official means to archive the outcomes. The match sheet page can be printed out for each of the matches at the beginning of the tournament to hand out to the referees.



AIStorm vs. Flamingos

x150g

x300g

x250g

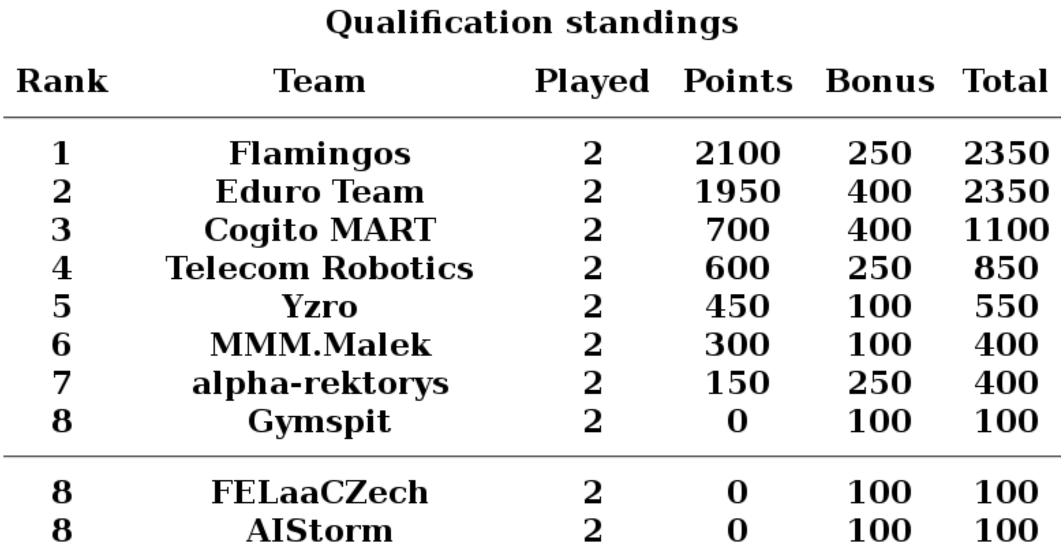
= 0g
+ 0 (scratch)

0 points

= 750g
+ 200

950 points

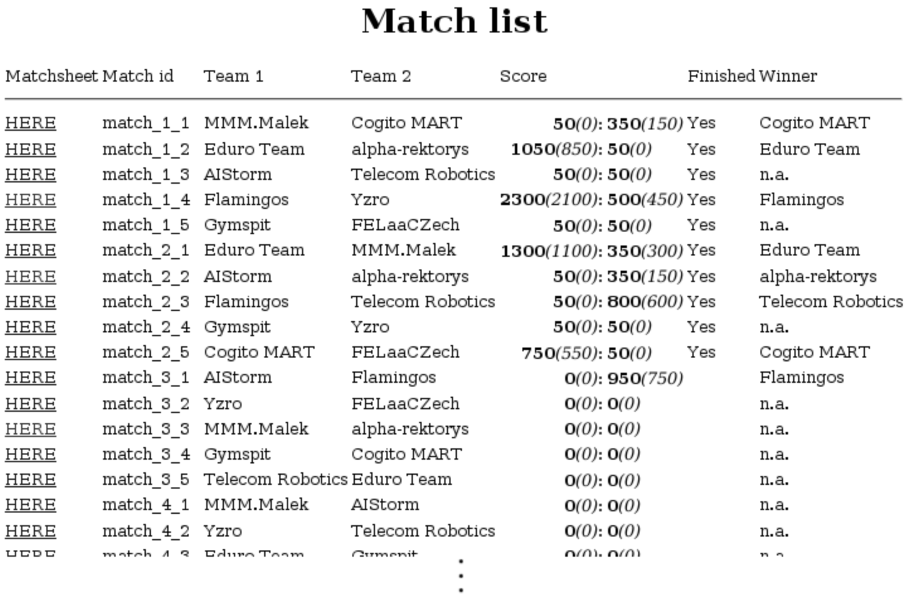
Figure 5.6: *Score of the current match*



Qualification standings

Rank	Team	Played	Points	Bonus	Total
1	Flamingos	2	2100	250	2350
2	Eduro Team	2	1950	400	2350
3	Cogito MART	2	700	400	1100
4	Telecom Robotics	2	600	250	850
5	Yzro	2	450	100	550
6	MMM.Malek	2	300	100	400
7	alpha-rektorys	2	150	250	400
8	Gymspit	2	0	100	100
8	FELaaCZech	2	0	100	100
8	AIStorm	2	0	100	100

Figure 5.7: *Presentation of qualification standings*

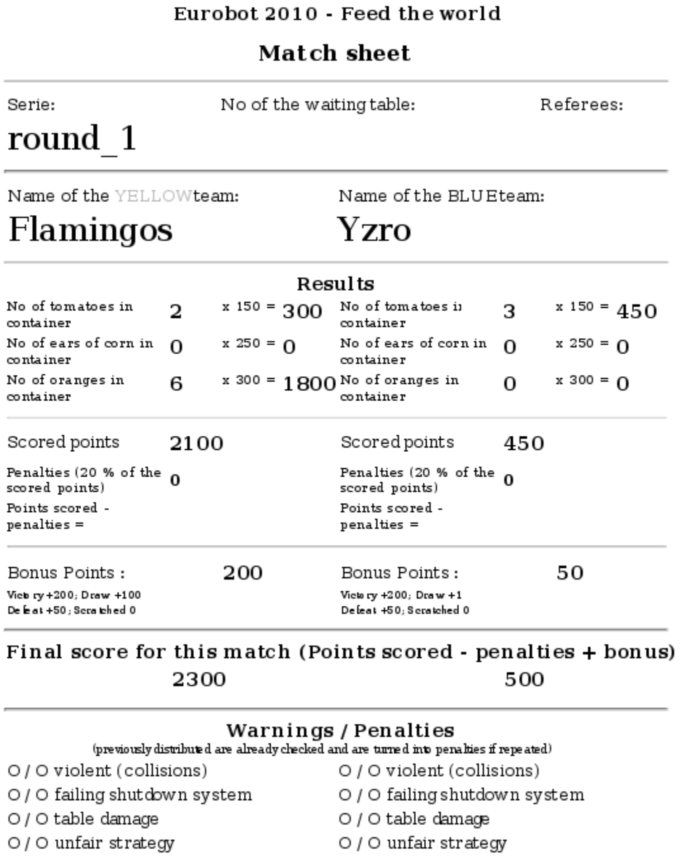


Match list

Matchsheet	Match id	Team 1	Team 2	Score	Finished	Winner
HERE	match_1_1	MMM.Malek	Cogito MART	**50**(*0*): **350**(*150*)	Yes	Cogito MART
HERE	match_1_2	Eduro Team	alpha-rektorys	**1050**(*850*): **50**(*0*)	Yes	Eduro Team
HERE	match_1_3	AIStorm	Telecom Robotics	**50**(*0*): **50**(*0*)	Yes	n.a.
HERE	match_1_4	Flamingos	Yzro	**2300**(*2100*): **500**(*450*)	Yes	Flamingos
HERE	match_1_5	Gymspit	FELaaCZech	**50**(*0*): **50**(*0*)	Yes	n.a.
HERE	match_2_1	Eduro Team	MMM.Malek	**1300**(*1100*): **350**(*300*)	Yes	Eduro Team
HERE	match_2_2	AIStorm	alpha-rektorys	**50**(*0*): **350**(*150*)	Yes	alpha-rektorys
HERE	match_2_3	Flamingos	Telecom Robotics	**50**(*0*): **800**(*600*)	Yes	Telecom Robotics
HERE	match_2_4	Gymspit	Yzro	**50**(*0*): **50**(*0*)	Yes	n.a.
HERE	match_2_5	Cogito MART	FELaaCZech	**750**(*550*): **50**(*0*)	Yes	Cogito MART
HERE	match_3_1	AIStorm	Flamingos	**0**(*0*): **950**(*750*)		Flamingos
HERE	match_3_2	Yzro	FELaaCZech	**0**(*0*): **0**(*0*)		n.a.
HERE	match_3_3	MMM.Malek	alpha-rektorys	**0**(*0*): **0**(*0*)		n.a.
HERE	match_3_4	Gymspit	Cogito MART	**0**(*0*): **0**(*0*)		n.a.
HERE	match_3_5	Telecom Robotics	Eduro Team	**0**(*0*): **0**(*0*)		n.a.
HERE	match_4_1	MMM.Malek	AIStorm	**0**(*0*): **0**(*0*)		n.a.
HERE	match_4_2	Yzro	Telecom Robotics	**0**(*0*): **0**(*0*)		n.a.
HERE	match_4_3	Eduro Team	Gymspit	**0**(*0*): **0**(*0*)		n.a.

⋮

Figure 5.8: *List of all the tournament matches*



Eurobot 2010 - Feed the world

Match sheet

Serie: No of the waiting table: Referees:

round_1

Name of the YELLOWteam: Flamingos	Name of the BLUEteam: Yzro

Results

No of tomatoes in container	2	x 150 = 300	No of tomatoes i container	3	x 150 = 450
No of ears of corn in container	0	x 250 = 0	No of ears of corn in container	0	x 250 = 0
No of oranges in container	6	x 300 = 1800	No of oranges in container	0	x 300 = 0
Scored points	2100		Scored points	450	
Penalties (20 % of the scored points)	0		Penalties (20 % of the scored points)	0	
Points scored - penalties =			Points scored - penalties =		
Bonus Points : Victory +200; Draw +100 Defeat +50; Scratched 0		200	Bonus Points : Victory +200; Draw +1 Defeat +50; Scratched 0		50

Final score for this match (Points scored - penalties + bonus)

2300	500

Warnings / Penalties

(previously distributed are already checked and are turned into penalties if repeated)

○ / ○ violent (collisions)	○ / ○ violent (collisions)
○ / ○ failing shutdown system	○ / ○ failing shutdown system
○ / ○ table damage	○ / ○ table damage
○ / ○ unfair strategy	○ / ○ unfair strategy

Figure 5.9: *Match sheet with automatically entered data*

Chapter 6

Conclusion

In this work, I have created a data model for modeling tournament systems and an event model that serves for manipulation and management of the data. I have also programmed an application implementing the designed model. The functions and features of the application were then shown on a case study that uses the application to manage the robotic competition Eurobot 2010. Both the application and the case study can be found on the included CD.

The data and event model that I created proved very universal and was applicable to the Eurobot 2010 competition without problems. The modular design of my application was also a good choice and I harnessed it when creating special presentation and match sheet generation php pages in the case study. Multiplatform technologies allow to use the application on a variety of operating systems.

Using the database back-end ensures that the data is accessible manually without the application, which enhances recoverability. Also, it allows to use external tools such as simple custom scripts that can be utilized to provide backup capabilities.

After the administration and programming has been done, use of the application to manage the tournament is simple. The user interface is visual and intuitive and is therefore usable even by less experienced users.

A slight disadvantage might be the learning curve that is associated with creation of new tournament definition files, because adapting the system to new conditions requires knowledge of the data model and the Lua scripting API. I shortly considered several ways to make adaptation of the system to another type of competition easier, but concluded that universality comes at a price of greater complexity and decided not to sacrifice the abilities of the application for slightly more user-friendly behavior.

6.1 Future work

The current system meets all of the initial requirements, but there is much room for improvement and the application can be expanded in several ways.

One of the major improvements would be to split the application into modules as it would make it possible for more people to use the system simultaneously. To do this, it would be needed to create a communication library that would ensure serialization of the requests from all the clients, as I originally proposed. This improvement would make the system more modular and even more universal.

Another area of possible improvements lies in the graphical part of the application. Although the graphical representations of the tournament matches and match groups serve their purpose well and the connection system is intuitive, a lot can be done to make the menus better, create tool-tip texts to provide in-application help and give the user more alternatives to accomplish a certain task, for instance by using keyboard shortcuts.

It would be also nice to provide a way to backup the data and allow the user to create restore points that can be used to revert the tournament to saved previous states. Together with serialization and logging all the data manipulation through the central module of the system, it would also be possible to revert the state of the tournament to any point in the past — not only the saved ones.

Automatization is also a possible direction to take when improving the application. Some connection patterns will be common to more tournament types and it would be very useful to be able to create these patterns automatically and avoid the so far necessary use of the mouse during the creation of the tournament structure. A feature that would allow duplication of match groups would also be very useful.

Lastly, it would also be possible to create a default presentation module for the application. Its abilities could include generating web-pages and various paperwork or displaying data over a live video stream.

Appendix A

Lua script API

This chapter contains all the information that is needed to write event handler routines in Lua. Basic knowledge of Lua lexical conventions is assumed. The complete language reference of Lua can be found in [8].

In the description of the system's scripting API, following conventions are used:

- Object identifiers are printed in **bold**.
- Source code snippets and examples are written with a `monospace` font
- Methods and member functions are described by their signature written with a `monospace` font (although they are actually not Lua code) followed by the description of what they do in *italics*. Note the difference between a method and a member function in Lua — method has an implicit and hidden first parameter "self". Methods are accessed by a colon instead of a dot. In this text, methods will be distinguished from member functions by preceding their name with a colon.
- Function signatures are written in the form `rettype name(params)`, where "rettype" is a return type (or a list of return types) of the function or method and "params" is a list of parameters separated by commas. For each of these parameters, first their type and then their name are given. The names are given only to be able to reference the parameters in the text description of the functions. An example:

  ```
  crop_table(table t, number size)
  ```

A.1 Object manager

The object manager can be accessed in Lua by the global table **omgr**. It contains three functions that create new globals, create object instances and delete object instances. They are:

`boolean newGlobal(string type_name, string name)`

Creates a new global of type type_name *named* name. *Returns true in case of successful execution and false otherwise (mostly because the given name is not valid or already occupied). For info on what is a type name, see section Types.*

`number newInstance(string object_type_name, string name)`

Creates a new instance of sub-type object_type_name *with an identifier* name. name *can be an empty string, in which case the application creates one, beginning with an underscore (which is normally an invalid name). Return value is the reference id of the newly created instance or zero in case of failure.*

`boolean deleteInstance(userdata to_delete)`

Deletes the given instance. The to_delete *argument has to be a Lua userdata object, created by the tournament manager Lua table.*

A.2 Tournament manager

The tournament manager table contains instance databases for the four base types for access to all the instances in the data model. It also contains the global variable manipulator for access to global variables and the event manager for event execution (that is, if one wants to nest an event execution inside an event handling routine). Nesting of the events should be used with care though — there is no infinite recursion detection and the event execution could end in an infinite loop of calls. The tournament manager table is saved as a global Lua variable (this should not be confused with global variables in my data model) under the identifier **tmgr**.

Instance databases and instance manipulators

The instance database objects are saved inside the **tmgr** table under the identifiers **actors**, **matches**, **tables** and **objects**. So to access an actor instance database, you type `tmgr.actors`. It is worth noting here that the dot syntax in Lua is syntactic sugar, so the previous code is equivalent to `tmgr["actors"]`.

Each instance database is a userdata object that can be indexed with the identifier of an existing instance of the given base type in order to obtain an instance manipulator userdata object. If no instance with the given identifier exists, the instance database returns `0`. The instance database cannot be indexed with a nonexistent identifier nor can it be used to save an instance under another identifier. It can only construct instance manipulator userdata objects.

Instance manipulator objects are Lua userdata objects that correspond to a single instance in the data model. They can be used to read and write member variables of their instance. To read a member variable value, index the instance manipulator with the variable's identifier. Similarly, to assign a member variable, assign the indexed instance manipulator. The manipulator objects can be freely copied in the event code. Also, wherever a reference to an instance is needed, either the manipulator or a reference id can be supplied. As an example, here is a Lua code that increases the value of a member variable:

```
--We save the manipulator in a local variable
local match = tmgr.matches.match1
--Then we increment the score for the home team
match.home_score = match.home_score + 1
```

By saving the value using the instance manipulator, we saved the value into the underlying data model.

Note that we could also have written:

```
tmgr.matches.match1.home_score = tmgr.matches.match1.home_score + 1
```

Apart from the user defined member variables, there are three useful system read-only variables that can also be read by the instance manipulator. They are `_name`, which is the name of the instance, `_id`, which is the instance's numerical identifier and `_type_name`, which contains a string with the name of this instance's sub-type. The last one can be used as a run-time type information for example to check for instance types before attempting to read a variable that is defined only in some sub-types.

Implicit variables

The tournament manager extension over the object model creates implicit variables in match and match group types to be used by the system. Some of them are useful to the event programmer because they provide valuable information. These implicit variables are listed here.

Match and match group variables

`array_actor _in`

An array with the actors entering the match or match group. The actors are also called inputs *of the match or match group.*

`array_actor _out`

An array with the outgoing actors. The actors are also called outputs *of the match or match group.*

`table _table`

The parent match group of this match or match group. If the match or match group has no parent, this variable is zero.

`int _in_num`
Number of inputs of this match or match group.

`int _out_num`
Number of outputs of this match or match group.

Match group-only variables

`array_actor _order_in`
The array with inputs of the Order-o-matic.

`array_actor _order_out`
The array with outputs of the Order-o-matic.

Global variable manipulator

The global variable manipulator object acts as a table that is indexed with the identifier names of the global variables. The object itself can be accessed at the index **globals** under the **tmgr** table. Reading values of the global variables and writing them works in the same manner as in instance manipulators.

A.3 Arrays

Arrays are created automatically if an array global variable is created or if a new instance is created with an array member variable. Arrays can be assigned, but this does not involve copying, only references are assigned. In many cases, assigning arrays can be avoided and is strongly discouraged. If you need to reference common data from multiple places, do it via references to an instance of base type object.

If you index the global variable manipulator or an instance manipulator with the name of an array variable, what you get is an array manipulator userdata object. This object can be indexed in order to access the values inside the array in the same manner as one accesses member variables:

```
--Incrementing the first array value by five
array[1] = array[1] + 5
```

The array values can be read or written into. Note that array indices have to be natural numbers, that means positive non-zero integers. A zero index is reserved to be used as an invalid index for the purposes of array iterators.

Values inside arrays cannot be erased, with the exception of data model object arrays (for example array of actors). Values in such arrays can be erased by assigning zero value to an array member. Another difference between object arrays and the other array types is that object arrays can be safely indexed with indices of non-existing array members, in which case the array returns 0 (invalid reference) as opposed to the other array types that end the script execution on invalid index. Note that this does not apply to setting new values — in that case, assigning to an invalid index is perfectly legal and creates a new array value.

An array manipulator also has a method for automatically appending a value to the end of the array. Its signature is as follows:

`:append(`*`type`*` new_value)`
Append a new value at the end of the array

The same could be accomplished by finding the `current` index of the `back_iterator`, incrementing it by one and assigning to it (and in fact, this is exactly what the `:append(...)` method does).

Array iterators

Apart from direct access to the values inside an array, one can get forward or backward iterators for iterating over the values in ascending or descending order, respectively. To get the iterator, index an array manipulator either with **iterator** or **back_iterator**:

```
--Accessing an array iterator
local it = array.iterator
```

Array iterator is a simple Lua table, not a userdata object. But it contains several member values, member functions and methods. Follows a description of these member values and functions.

`int id`
Member variable containing the numerical identifier of the array that this iterator is tied to.

`int current`
Index of the value that the iterator is currently pointing at.

`int type`
Type of the array values casted to an integer. See the source code file tmgrType.h *to see which number corresponds to which type.*

`int :next()`
This method increments the current *index to the index of the next value in the array (which does not necessarily have to be the next integer number, because the array can have gaps) and returns the new value. If there is no valid next index, zero*

is returned.

`int :prev()`

This method decrements the current *index to the index of the previous value in the array (which does not necessarily have to be the preceding integer number, because the array can have gaps) and returns the new value. If there is no valid preceding index, zero is returned.*

`type :get()`

Returns a value that the iterator currently points at. The type *of the returned value is the same as the type of all values in the array (naturally).*

`:set(type new_value)`

Sets the value of the array member that the iterator is currently pointed at to the value provided. The method has no return value.

To iterate over an array, you then use the while loop construct as follows:

```
--Use of the forward iterator
local it = tmgr.globals._roster.iterator
while it.current ~= 0 do
  print (it:get().name) --Do something with the iterator
  it:next()             --Increment the iterator
end
```

To iterate the array backwards in the same example, you would simply change `iterator` to `back_iterator` and `it:next()` to `it:prev()`.

A.4 Events

All the events that are loaded at the start of the application are saved into the table **events** in the **tmgr** table. The values saved are actually functions, whose parameters match those in the corresponding events. Thus, in order to execute an event from Lua, call the function that resides in the **events** table under the index that is the same as the event's name. Here's an example that uses the scoring event that was used in the design description to erase the score of the current match:

```
tmgr.events.score(tmgr.globals.current_match, 0, 0)
```

A.5 Types

In some places of the scripting API, one has to provide a type name. This can be either a name of a sub-type derived from the four base types (actor, match, table or object) or a name of a tournament manager type, depending on the context in which the type name is required. If a sub-type name is required, you have to use the name of a sub-type that was declared before launching the application.

If a tournament manager type name is needed, use one of the following or precede one of the following type names with a prefix *array_*, in which case the type will be an array of variables of the given type. Arrays of arrays are forbidden, so you can append the prefix to the type name just once. The tournament manager types are:

int An integer number

float A floating-point decimal number

string A string

bool A boolean value (true or false)

object Reference to an instance of an object sub-type

actor Reference to an instance of an actor sub-type

match Reference to an instance of a match sub-type

table Reference to an instance of a table sub-type

Appendix B

Terminology

Throughout this work, I use several terms that might not be obvious to everyone. Also, some widely known terms can be used with a different meaning or in a different context. This glossary lists those terms and clarifies their meaning.

Actor: *See: Base type*

Base type: One of the four general types (actor, match, table, object) in my object abstraction over the back-end database. Actor represents a player, team or other kind of contestant in a tournament. Match is pretty self-explanatory. Table is a grouping of matches or tables in a tournament. It can represent a round, qualification table, phase or any other similar group of matches. Object represents other objects that do not fall into the previous three categories. Specific sub-types must be derived from these base types in order to use them.

Best-of-three: A playoff match format that consists in playing three subsequent matches and declaring winner the team that wins at least two of them. Often, if a team has won twice in the first two matches, the third match is dropped.

Bracketing: Creating tournament brackets — that is selecting the players that will play together in one match, planning the structure of the matches and drawing it on the paper. In context of tournament management applications, bracketing denotes the ability to plan tournament brackets in the application and often also the ability to print the brackets out. *See also: Drawing*

Double-elimination: *See: Elimination*

Drawing or draw making: This notion can be a bit misleading, because in sports a draw means an indefinite outcome of the match (noone has won). However, in context of tournament management, draw making is the act of selecting the players that will play against each other in a match. It is actually similar to and by my definition a subset of bracketing. *See also: Bracketing*

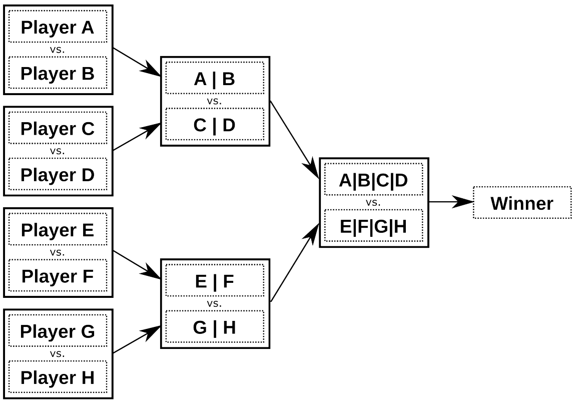


Figure B.1: *Single-elimination tournament for eight players*

Elimination: A tournament scheme that eliminates (hence the name) players out of the tournament after one or two losses, depending on whether the scheme used is single-elimination or double-elimination respectively. See figures B.1 and B.2 for reference.

In single elimination, each round consists of half the matches and players than in the previous round, resulting in a binary tree of matches and a logarithmic number of rounds in the tournament. Only winners are kept in the tournament each round and they continue to the next round. In double elimination, losers are given one more chance to win the tournament and play a parallel single-elimination, whose winner ultimately faces the winner of the main elimination tree to determine the winner of the whole tournament.

If the number n of contestants in the first round is not a power of 2, we can add a $0th$ round that will consist of $m = n - 2^{\lfloor log_2(n) \rfloor}$ matches, so that we have $2^{\lfloor log_2(n) \rfloor}$ contestants in the $1st$ round, which is a power of two. Those players that won't play a match in the $0th$ round will automatically advance into the $1th$ round (this is called a bye).

Match: *See: Base type*

Member function (Lua): *See: Method (Lua)*

Method (Lua): In Lua, tables can hold member variables as well as member functions. These functions are not different from global functions in that they are not tied to their table in any way. Methods are a mere syntactic sugar — a different way to call member function. To call a function as a method, use colon instead of dot to separate the name of the table and the function. When a member function is called this way, the containing table or object is automatically provided to the function as the first parameter. It can be accessed inside the function as a variable named `self`.

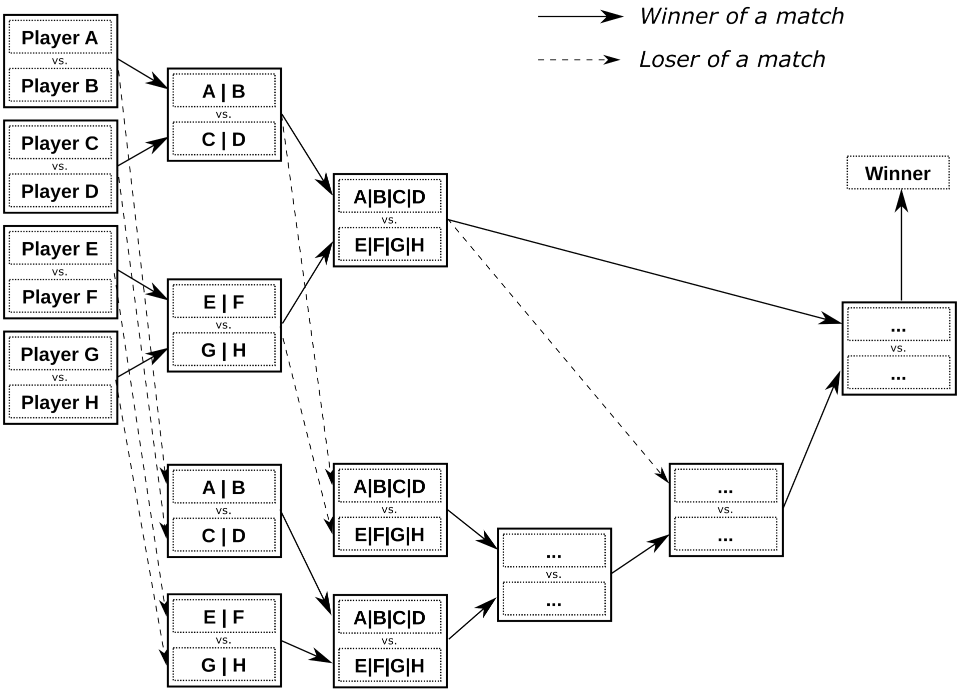


Figure B.2: *Double-elimination tournament for eight players*

```
--Calling the member function as a function
tab.func()
--Calling the member function as a method
tab:func()
```

Object: *See: Base type*

Round-robin: In general, this term means a cyclic list of items that has no end or beginning. In tournament management, round robin denotes a tournament scheme, where all the players but one are put into a cyclic list. The list is then rotated each round to pair players together to form matches in that round. See figure B.3 for a better understanding of how this works.

After playing a predetermined number of rounds (possibly $N - 1$ where N is the number of contestants), the tournament is finished and the winner is determined by a number of wins, score or by similar means. If there is an odd number of contestants, a dummy contestant can be placed in each round that will be played by one of the contestants, but the dummy's score won't count toward the final score of the contestant that plays the dummy.

Single-elimination: *See: Elimination*

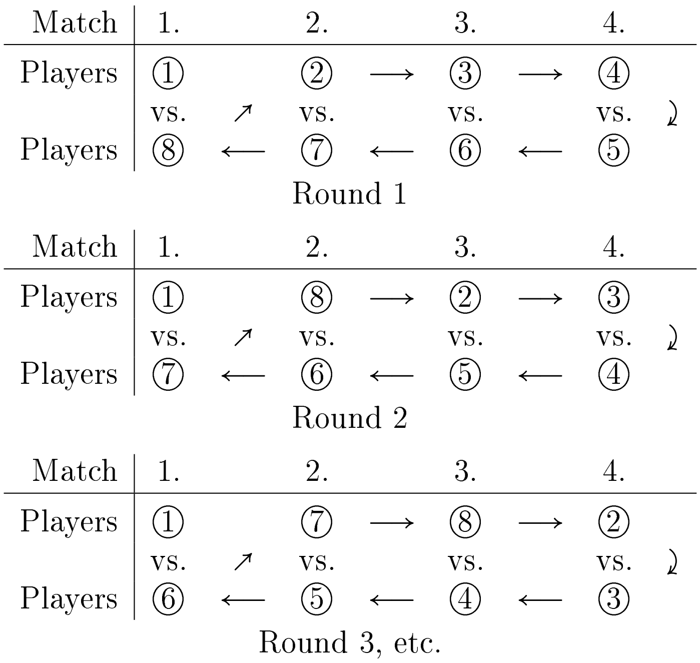


Figure B.3: *First three rounds of an eight-player round-robin tournament*

Sub-type: *See: Base type*

Swiss pairing: A tournament scheme in which the individual players are assigned a rating that is calculated by a given formula and changes according to outcomes of the tournament matches. The tournament in this system is divided into rounds and an adequate opponent is selected for each player in each round, based on their ratings before the start of the round. The idea behind this system is to pair players, whose performance during the tournament is similar. The winner of the whole tournament is usually determined by the performance score, which means there is no elimination round at the end of the tournament. Note that as described, the scheme could theoretically pair the same players in more than one round, so there is often a condition in the pairing algorithm that prevents this and pairs different players instead.

Table: *See: Base type*

Trie: A tree-like data structure in which branches of each node are indexed by characters. Traversal of the structure is done by iterating over characters of a string and going down the corresponding branch for each of the characters. The structure is used to map strings to other values and find the mapping very quickly.

Appendix C

Contents of the CD

The compact disc that is enclosed in this work contains the text of the work in a digital form, source code and programmer documentation of the application and also the whole case study. This chapter describes the directory structure of the CD. The directory structure and contents of the disc are also described in a file `README.TXT` in the root directory on the disc.

app This directory contains all the data that is related to the application. The `SConstruct` build configuration file is located here. The directory also contains the `LICENSE.TXT` with license information for the source code. The license information is also prepended to each source file separately.

- **src** The `src` directory contains the whole source code of the application and its graphical interface.
  - **lua** This directory contains code for the parts of the application that use the Lua scripting language.
  - **qt** This directory contains the graphical user interface source code.
  - **res** The icon images for the two arrows used in the graphical interface are here as well as their license.
- **doc** This directory contains the source code documentation for the application and the programmer's documentation (which is most of the Chapter 3).

work The LaTeXsource file for this work is contained in this directory as well as all the images that are used in it.

- **illustrations** Svg sources and rendered raster images of my illustrations used in the work can be found here.

case study This directory contains the whole Eurobot 2010 case study.

- **events** This directory contains the event definition files for the case study.
- **types** This directory contains the type definition files for the case study.

presentation The presentation files for the case study can be found in this directory.

References

[1] Stroustrup B.: *The C++ Programming Language*
http://www2.research.att.com/~bs/C++.html.

[2] Richards J., Hill D.: *Complete Handbook of Sports Scoring and Record Keeping*, Parker Publishing Company, Inc., West Nyack, N.Y., 1974.

[3] *Tournament Manager - Safrad's Pages*,
http://safrad.own.cz/Software/TournamentManager/index.html.

[4] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, pages 293-303, Addison-Wesley, 1994.

[5] *Eurobot official website.* http://www.eurobot.org.

[6] *Eurobot 2010, Czech cup official website.* http://www.eurobot.cz.

[7] *Eurobot 2010 rules.* http://www.eurobot.org/eng/rules.php.

[8] Ierusalimschy R., de Figueiredo L. H., Celes W.: *Lua 5.1 Reference Manual*, Lua.org, 2006. http://www.lua.org/manual/5.1.

[9] *MySQL :: The world's most popular open source database.*
http://www.mysql.com.

[10] *MySQL++.* http://tangentsoft.net/mysql++.

[11] *The Programming Language Lua.* http://www.lua.org.

[12] *Python Programming Language – Official Website.* http://www.python.org.

[13] *SCons: A software construction tool.* http://www.scons.org.

[14] *SQLite Home Page* - Appropriate Uses For SQLite.
http://www.sqlite.org/whentouse.html.

[15] *Strongvon website.* http://strongvon.com.

[16] *TMS - Tournament Management System.*
http://www.tennisinformation.com/products.

[17] *Products – Qt - A cross-platform application and UI framework.* http://qt.nokia.com/products.