University of Hradec Králové
Faculty of Informatics and Management
Department of Informatics and Quantitative Methods

Desktop Java Application for Data Analysis in E-sport

Bachelor thesis

Author: František Bláha

Study Programme: Information Management

Supervisor of the bachelor thesis: Vojtěch Vorel

Hradec Králové April 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

This thesis is dedicated to my supervisor, whose guidance, support and expertise have been essential in the development and completion of this thesis. I would also like to thank my girlfriend for her understanding and support throughout this process. Her patience and motivation have been crucial in helping me reach this milestone.

Title: Desktop Java Application for Data Analysis in E-sport

Author: František Bláha

Department of Informatics and Quantitative Methods

Supervisor: Vojtěch Vorel

Abstract: This thesis introduces a Java-based application, developed in IntelliJ IDEA, designed to assist League of Legends coaches and analysts in match preparation. The application streamlines the process by automating data acquisition, storage, and processing from multiple external sources, including in-game statistics, player performance metrics, and historical match data. By analyzing this information, the app generates valuable insights that can be utilized for strategizing and optimizing team performance in upcoming matches. In addition to its core functionalities, the application offers a user-friendly interface, allowing for efficient navigation and interpretation of the generated insights. This research highlights the potential of technology in esports analytics, specifically within the context of League of Legends, and contributes to the growing field of data-driven strategies in competitive gaming. The development of this application sets the stage for future advancements in integrating analytics and technology in esports coaching and analysis.

Abstrakt: Tato práce představuje aplikaci v jazyce Java, vyvinutou v prostředí IntelliJ IDEA, která má pomáhat trenérům a analytikům hry League of Legends při přípravě na zápasy. Aplikace zefektivňuje proces automatizovaným získáváním, ukládáním a zpracováním dat z několika externích zdrojů, včetně herních statistik, metrik výkonnosti hráčů a historických dat ze zápasů. Analýzou těchto informací aplikace generuje cenné poznatky, které lze využít pro tvorbu strategie a optimalizaci týmového výkonu v nadcházejících zápasech. Kromě svých základních funkcí nabízí aplikace uživatelsky přívětivé rozhraní, které umožňuje efektivní navigaci a interpretaci vygenerovaných poznatků. Tento výzkum poukazuje na potenciál technologií v oblasti analýzy e-sportu, konkrétně v kontextu League of Legends, a přispívá k rostoucí oblasti strategií založených na datech v profesionálním hraní. Vývoj této aplikace připravuje půdu pro budoucí pokrok v oblasti integrace analytiky a technologií do koučování a analýzy e-sportů.

Keywords: League of Legends; E-sports analytics; Match preparation; Java-based application; Data-driven strategies

Contents

Introduction

The competitive gaming landscape has evolved significantly over the past few years, with a growing emphasis on the use of data analytics and technology to optimise team performance. As a result, coaches and analysts in *e-sports*, particularly in *League of Legends* require efficient tools to assist them in match preparation. This thesis aims to address this need by developing a *Java-based* application in *IntelliJ IDEA* that streamlines the process of extracting, persisting and processing data from multiple external sources.

One of the key features of the application is its ability to be customised based on the specific requirements of each customer. This adaptability ensures that the output generated by the application remains relevant and valuable to individual users, allowing them to tailor their strategies according to their unique needs and preferences.

The choice of *Java* as the programming language for this application was made due to its versatility, widespread adoption, and strong community support, which allows for easy integration with various external data sources and databases. In addition, Java's platform independence ensures that the application can be easily deployed and maintained across different operating systems, further enhancing its usability.

IntelliJ IDEA was chosen as the development environment for its advanced features, such as code completion, debugging, and refactoring tools, which greatly improve the overall development process. The *Integrated Development Environment* (IDE) also offers seamless integration with various *Java* libraries and frameworks, making it an ideal choice for the development of this application.

In summary, this thesis presents the development of a customisable Java-based application in *IntelliJ IDEA*, aimed at enhancing match preparation for *League of Legends* coaches and analysts. By leveraging the strengths of *Java* and *IntelliJ IDEA*, this application offers a comprehensive, adaptable, and user-friendly solution for data-driven esports analysis and strategy development.

1. Theoretical part

1.1 League of Legends

League of Legends (LoL) is the most popular computer video game and belongs to the subgenre of *Real-Time Strategy* (RTS) games called *Multiplayer Online Battle Arena* (MOBA) [1, 2, 3]. It was released in 2009 by a company called *Riot Games* [1, 3, 4]. MOBAs offer an unprecedented research opportunity thanks to two important features: the huge player base and easy access to recorded in-game data via *Application Programming Interface* (API) [3].

It is a team-based strategy game that pits two teams of five players against each other with the main objective of destroying the opposing team's base, called the *Nexus* [2, 5, 6]. The game is played on a map called *Summoner's Rift*, which has two bases in opposite corners, with three lanes connecting them, and the area between the lanes full of side objectives (neutral enemies to kill) which, once captured, gives an advantage in achieving the main objective [2, 5, 6]. Each lane has 3 lines of defence in the form of laser weapon turrets called outer, inner and inhibitor turrets. The Nexus has two more turrets as its own defence. This area between the lanes is then divided into the neutral area between the centres of the lanes, called the *River*, and the rest, called the *Jungle* [5, 6]. The map of the Summoner's Rift is shown in Figure 1.1.

1.1.1 Important game mechanics

Snowball effect

Snowball effect refer to scenario where momentum builds progressively, similar to a snowball rolling downhill, collecting more snow and gaining speed as it moves [7]. In *League of Legends*, for example, when a team wins a teamfight, they gain gold from kills, spend that gold on items, thereby gaining an advantage and having a higher probability of winning the next teamfight.

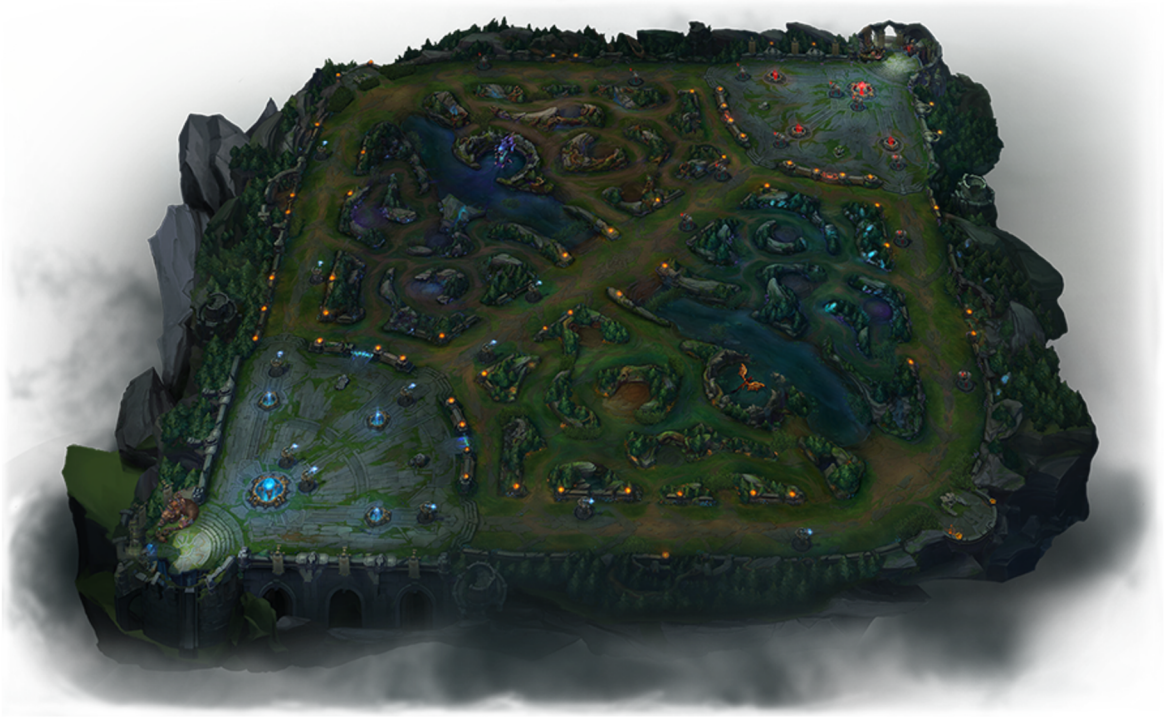

Figure 1.1: Summoner's Rift map with three lanes connecting opposing bases and the river in the middle. Taken from [5].

Scaling

Scaling in *League of Legends* refers to the concept of a champion or team composition becoming more powerful as the game progresses into the later stages [8]. Understanding scaling can be complex, as it involves recognising how game dynamics change over time, how the distribution of gold and experience affects champion stats, and how certain abilities and move sets allow champions to excel in the late game [8].

Counterpicking

Counterpicking is the act of gaining a significant advantage over the opponent by selecting champions whose abilities, items, and scaling give them an advantage over the opponent's champions that have already been selected.

Skirmish vs Teamfight

Both terms refer to fighting between teams, but a *teamfight* involves all or nearly all of the players in the game, while a *skirmish* involves anywhere from three to seven

players. Teamfights tend to take place in the mid and late game, and skirmishes in the early and mid game.

Wave management

Every half minute, three groups of *Non-Player Characters* (NPCs) called *minions* spawn at both Nexuses and head towards the enemy Nexus, one group for each lane. Due to their recurring arrival, this group of minions is known as the *Wave*. These minions are much weaker than players. Players kill minions to defend their tower, but more importantly, to get gold for being the last one to hit the minion.

The main concepts involved in *wave management* are slow pushing, fast pushing and freezing. *Slow pushing* involves killing only a portion of the minions, allowing the wave to slowly advance towards the enemy's tower while being replenished by other waves coming from the nexus [9]. This strategy works well when timed with the spawning of key objectives or killing enemies under their turrets [9].

Fast pushing involves eliminating entire wave quickly, forcing the enemy to react or risk losing their last hits. *Freezing* involves hitting the last minions at the last possible moment, while matching the enemy's damage to keep an even number of minions in the lane, effectively denying them gold and experience if the enemy is not present in the lane [9].

Warding

Warding is the act of a player placing a ward on a point on the map to maximise a team's vision. A Ward is a tool that reveals a certain area of the map around it [10]. Wards can be destructed by opposing players.

Counter-jungling

Counter-jungling refers to the act of invading the enemy's jungle to steal their resources, deny them gold and experience, or disrupt their gameplay [11].

Ganking

Ganking is the act of a jungler moving into the lane to engage in a small skirmish and fight for a kill, thereby gaining a significant advantage [11]. Even laners can sometimes move from their lane to another lane to gank, but this is more commonly referred to as *roaming*.

1.1.2 Positions in League of Legends

In *League of Legends*, *positions* define both a player's function within a team and how they contribute to the team's overall performance, as well as the area on the map where they spend most of their time in the game [5, 12]. The positions marked in their significant areas are shown in Figure 1.2. These positions are crucial in differentiating each lane and providing structure to the game.

Figure 1.2: Summoner's Rift map with positions marked in their significant areas. Original image taken from [5].

Understanding these positions helps players identify their own and their opponents' team compositions and strategies in both the early and late stages of the game [12]. Understanding the positions in LoL is essential for improvement and success in the

game, as it helps play a character according to its design and maximise its potential and effectiveness, as characters in the game are generally better suited to certain positions based on their stats, abilities, and playstyle [5, 13, 12].

According to [5, 13], making a balanced and competitive team requires all five of the positions listed here:

- Top Lane
- Jungle
- Middle Lane
- Attack Damage Carry
- Support

Top lane

The *top lane* is notoriously known as the "island" because it is the most isolated position of all. It is also the most volatile for two reasons. Firstly, the player with the counterpick usually has a significant advantage [10]. Secondly, it is the most snowballing position in the game. Players in this position tend to choose strong duel characters that can take a lot of damage and engage in fights with the opposing team [14]. These characters must be able to fend for themselves as they are often isolated from other players in the game [11]. As the game progresses, top lane players can move to other parts of the map to gank opponents and gain an advantage [14].

Jungle

The *jungler* is the most complex role in the game, as it is responsible for controlling the flow of the game, making decisions based on information gathered from warding, counter-jungling, and ganking, while keeping an eye on the opposing jungler [10, 11].

They spend most of their time in the jungle area between the lanes, collecting gold and *eXperience Points* (XP) to build up their economy. They need to provide good visibility around key points on the map with wards and support their teammates,

especially the midline champion, by covering their laners or ganking enemy laners [10, 14].

There are different classes of junglers, such as *farming* junglers, who will constantly farm for gold and only gank when they're sure of success [11]. *Carrying junglers*, on the other hand, focus on getting gold quickly through kills and dominating the game [11].

Mid lane

The *mid laner* in LoL has two main responsibilities: farming and roaming [10]. Wave management is important, although it differs from other lanes due to the shorter length of the mid lane [10].

Choosing a character that complements jungler's play style is crucial, as they need to work together to succeed [10]. The symbiosis between the mid laner and the jungler works in such a way that the jungler covers the mid laner so that he can push, and then the mid laner has time to roam and create map pressure, which helps the jungler with ganking or counter-jungling [10].

Mid laners often deal a lot of damage and are usually ranged [14]. They need good map awareness and vision control as they can be attacked from a variety of positions [14].

Attack Damage Carry

A *bot laner* has a choice of playing either an *Attack Power Carry* (ADC) or an *Ability Power Carry* (APC), which differ in their main damage type. Traditionally, it is an ADC, which is why the bot laner is often referred to as an ADC. Their main focus is to farm and generate gold from last hit minions, with the aim of becoming the main damage dealer for their team [10].

Bot laners tend to be ranged champions with relatively low base health and defence stats, making them vulnerable while still dealing significant damage [14]. That's why wave manipulation and positioning are critical skills for bot laners [10]. Their damage

is critical to neutral targets such as Dragons, Barons, and Turrets [10].

Support

Support players assist their teammates, especially the ADC, by warding to get vision and engaging in teamfights at the right time [10, 11, 13, 14]. In teamfights, Supports shift from dealing damage to providing utility with their abilities [10, 11].

There are two types of support: *Engage* and *Enchanter* [11]. Engage supports are more durable and initiate fights, while Enchanter supports focus on healing and shielding allies [11]. By adapting their play style and understanding their role, support players have a significant impact on the outcome of the game [10].

1.1.3 Phases of the game

The game is divided into phases, which are characterised by typical player behaviour at certain points in the game. All phases are shown in Figure 1.3 and Table 1.1 summarises the characteristics of each phase.

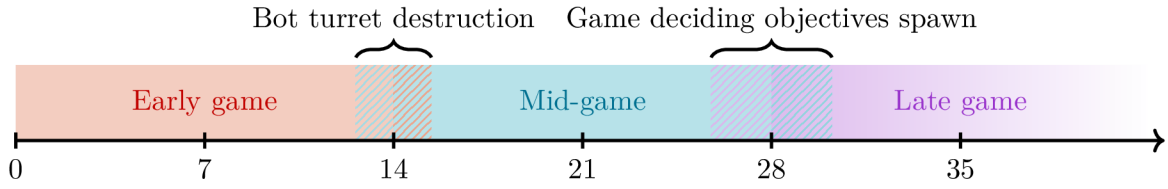

Figure 1.3: Timeline of League of Legends game phases.

Early game

In the *early game*, often referred to as the *laning phase*, all laners (top, mid, and bot laners) play cautiously, staying in their lanes and focusing on farming gold and experience to strengthen their champions. The jungler, sometimes with support, will try to spot and punish any mistakes made by the opposing laner all over the map (but mostly around mid lane), giving them a significant advantage as they move into the next phase of the game. Sometimes both junglers will try to make an action in the same area, which will usually result in a skirmish.

Phase	Lane cover*	Individual behavior	Fighting	Impact positions	Objectives
Early game	1:2:2	Laning Farming	Ganks Skirmishes for neutrals	Jungler Support	Herald Dragons
Mid game	1:3:1	Holding mid Capturing sides	Teamfights for neutrals	Top Mid	Baron Dragons
Late game	5/4:1	Grouping	Teamfights for neutrals	ADC	Baron Dragon soul Elder dragon

Table 1.1: Summarise of the characteristics of each phase of the game.
*For example 1:3:1 means one player playing around top lane, three around mid lane and one around bot lane.

The early game begins at the start of the game and lasts until the first turret in the bot lane is destroyed, usually just after the 14th minute, which is when the plates disappear from the turrets (most lol experts now agree on this). Plates are turret fortifications that give extra money when destroyed. The main neutral objectives in this phase are the first *herald* and the *dragons*. The time when teams try to capture these objectives is the only time in this phase when players are grouped and teamfighting 4v4, sometimes even 5v5.

Mid-game

The *mid-game* begins immediately after the early game, when players begin to group together to get objectives, push lanes, and fight opponents. At the start of this phase, the ADC will swap positions with the midlaner. The jungler and support will mostly play around the midline to cover the ADC and help him hold the most important tower in the game, the outer midline tower. Mid and top laners will continue to push waves of minions down the side lanes, visiting the mid lane whenever possible to make an impact.

Farming is still important, but as this phase progresses, objectives become more important. The main neutral targets in this phase are outer turrets and dragons, but the most important is the *Baron*, which once killed gives enough advantage to destroy

inner and inhibitor turrets, usually leading to victory before the next phase even starts. This phase is characterised by a lot of teamfights for objectives and skirmishes forced by teams who feel they have an advantage at the time and place. It usually starts around the 14th minute and ends around the 28th minute.

Late game

The *late game* begins when players reach high levels and are almost fully equipped with items, so they no longer need to farm for gold and experience. Most of the time, players are grouped as 5, or in certain scenarios, when the top laner or mid laner is playing a character suitable for dueling and pushing, he is separated to push one of the sidelanes. The grouped teams engage in large team fights with the opposing team for game-changing objectives such as *Dragon Soul*, Baron, or *Elder Dragon*. Typically, the most important position in this phase is the ADC, who scales the best and therefore has the most impact in team fights by dealing the most damage. Nowadays, only a fraction of games make it to late game, many games snowball very quickly and end in mid-game.

1.2 E-sport

E-sport is video game played in a highly structured competitive environment across many different genres [1]. *E-sport* is an industry with a growing global market revenue of almost $1.4 billion in 2022, which is shown in Chart 1.4. *E-sport* is a complex global phenomenon with a rapidly growing audience of 532 million viewers in 2022, which is shown in Chart 1.5 [15].

1.2.1 The History of E-sport

The first signs of the fast-growing e-sport phenomenon were seen in *South Korea* in the early 2000s, with the establishment of the *Korean e-SPorts Association* (KeSPA) and a television channel dedicated to e-sports [1].

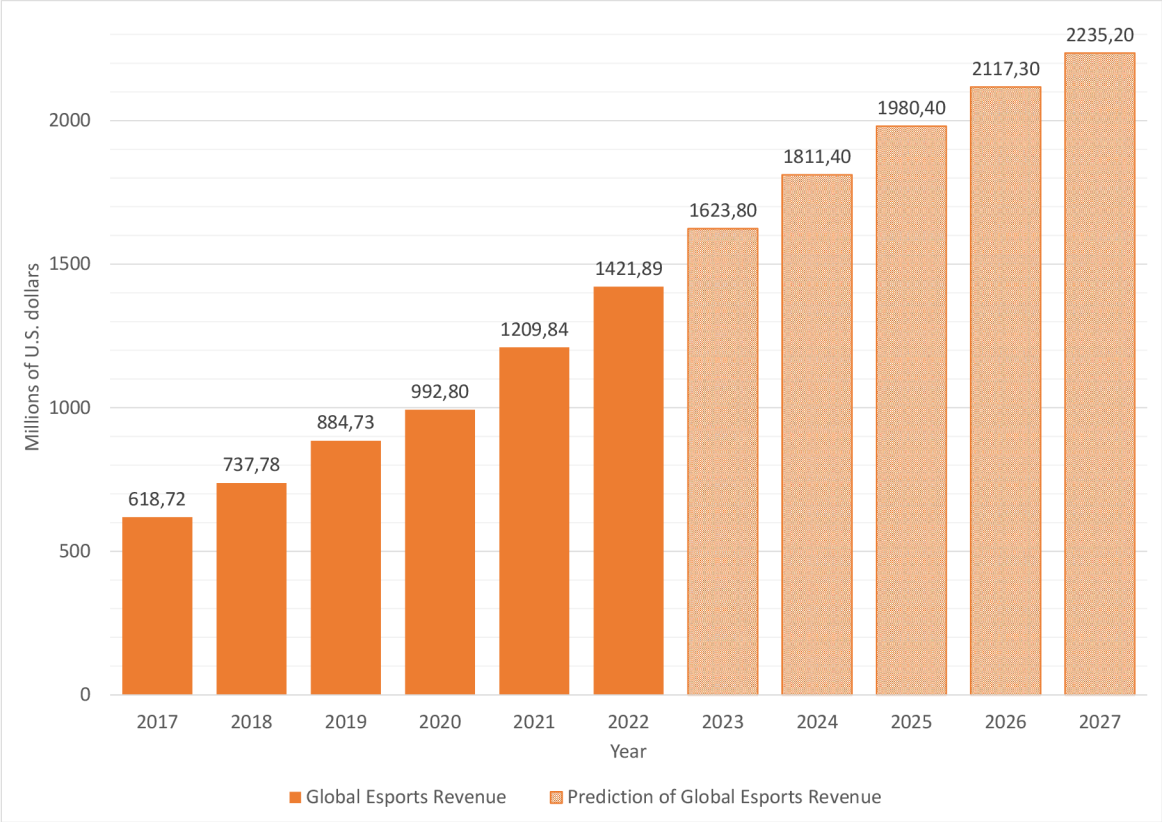

Figure 1.4: Global E-sports Market Revenue with actual data from 2017 to 2022 and forecasts for 2023 to 2027. Data taken from [16].

A great example of this industry's growth is *Defense of the Ancients 2* (DOTA2), a game that started in the early days of e-sports with a network of small tournaments with prize pools of hundreds of dollars, but has grown massively over time [1, 17]. The winner of *The International 2021* (TI21) — the biggest tournament of the year in DOTA2 — was awarded $18 million out of a total prize pool of $40 million, the largest prize pool in e-sports history and larger than many prize pools in mainstream traditional sports [1, 18].

Today, e-sports organisations have players with contracts, coaching and support staff, and headquarters or training facilities where they practice together, much like traditional sports [1].

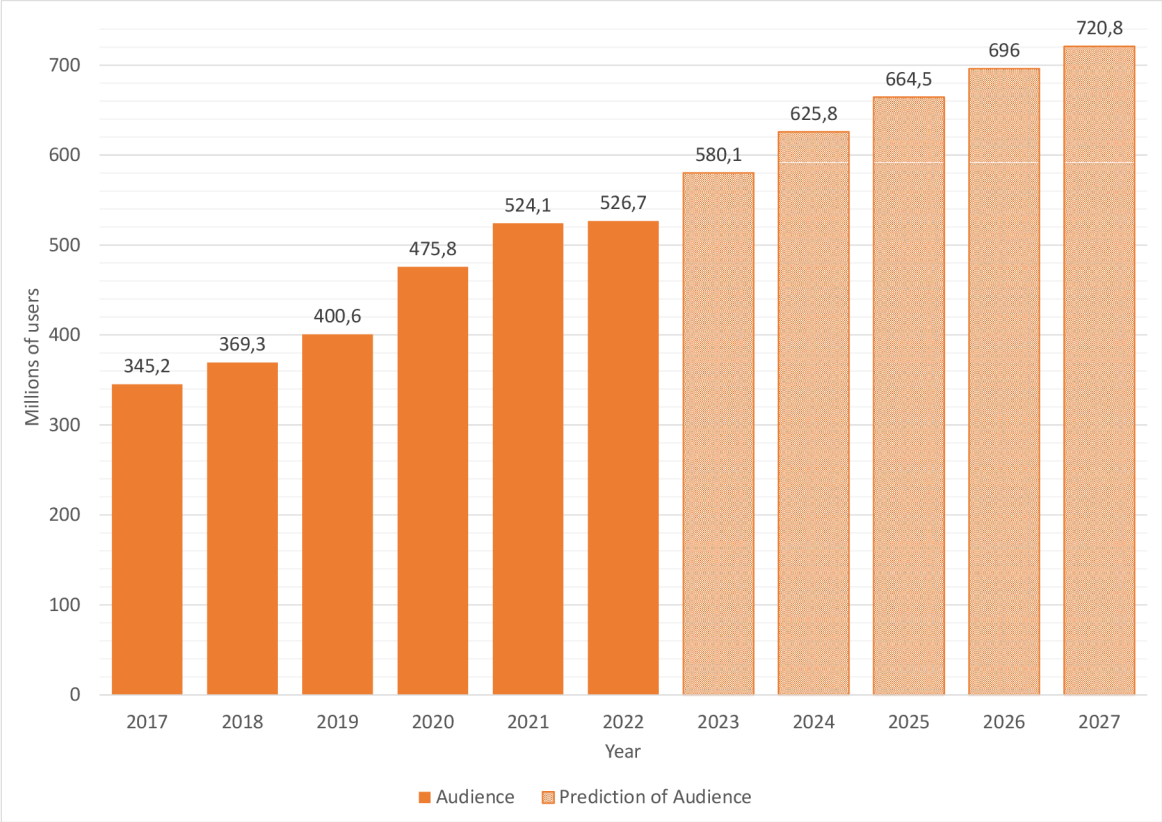

Figure 1.5: Global E-sports Audience with actual data from 2017 to 2022 and forecasts for 2023 to 2027. Data taken from [16].

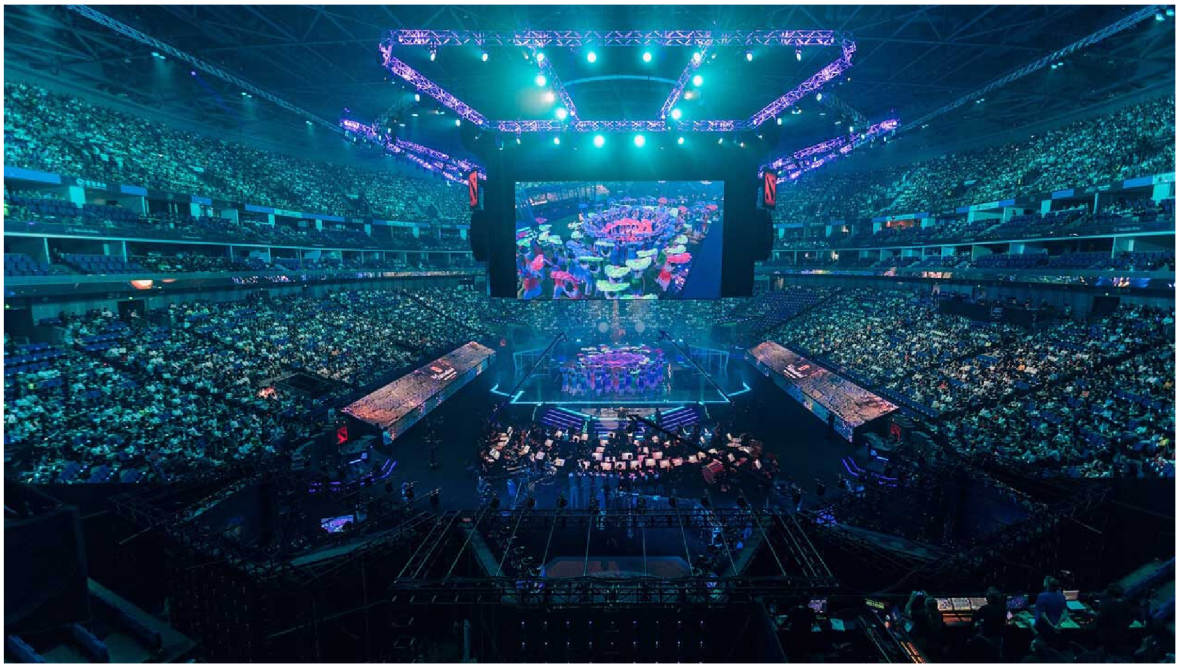

Figure 1.6: Venue of The International 2022. Taken from [19].

1.2.2 E-sports in League of Legends

The history of LoL e-sports follows a typical e-sports example. Although LoL was released later (in 2009), it has also grown from individual, privately organised tournaments to the first *World Championship* in 2011 with eight teams, a peak audience of over 210,000 concurrent viewers and a prize pool of $99,500, to the *2016 World Championship* with more viewers than the *National Basketball Association* (NBA), to the *2021 World Championship* with twenty-two teams, a peak audience of 73,860,000 concurrent viewers and a prize pool of $2,225,000, plus a percentage of revenue from special in-game *microtransactions* [1, 20, 21]. *Microtransactions* are the purchase of small in-game items for small amounts of money [22]. The growth in *viewership* is shown in Chart 1.7. The growth in *prize money* is shown in Chart 1.8.

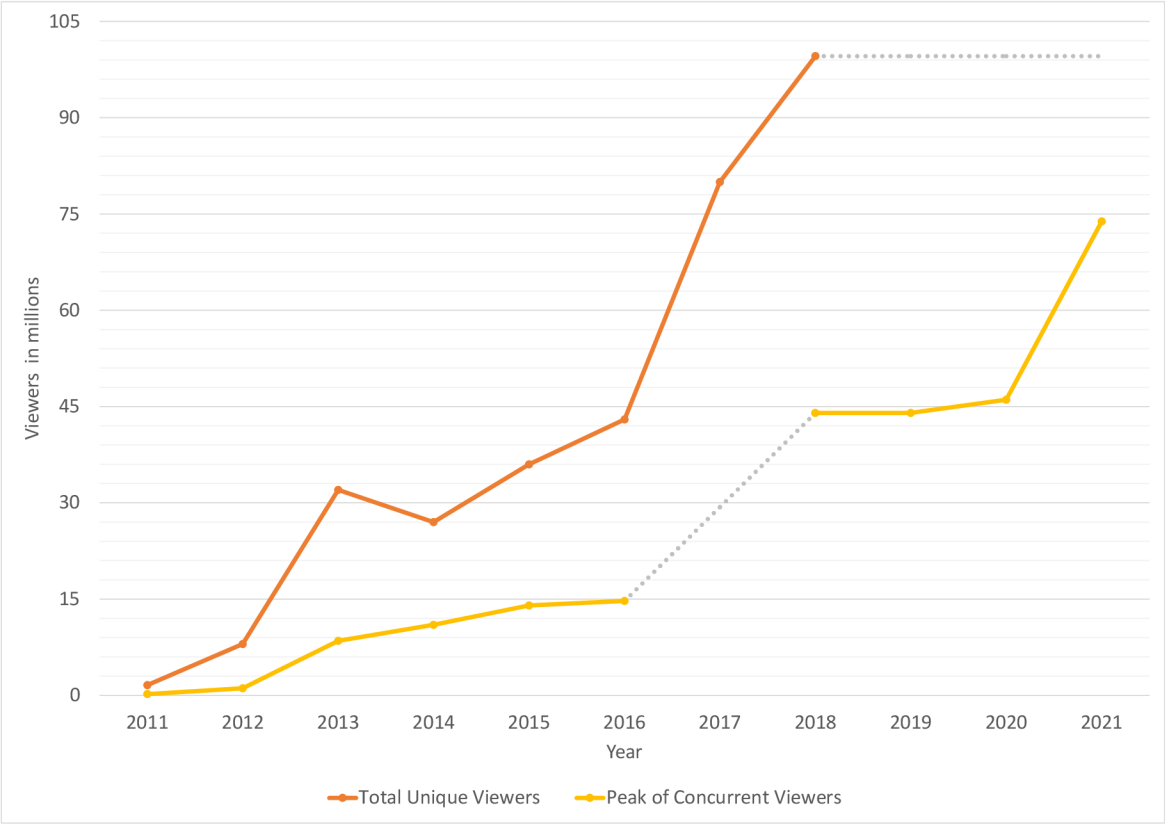

Figure 1.7: Growth in total unique viewers and peak concurrent viewers for the World Championships from 2011-2021. Data taken from [21, 23, 24, 25].

According to [4], the LoL e-sports ecosystem is the largest and most popular in the e-sports industry, and according to [26], it is the fastest growing sport in the world.

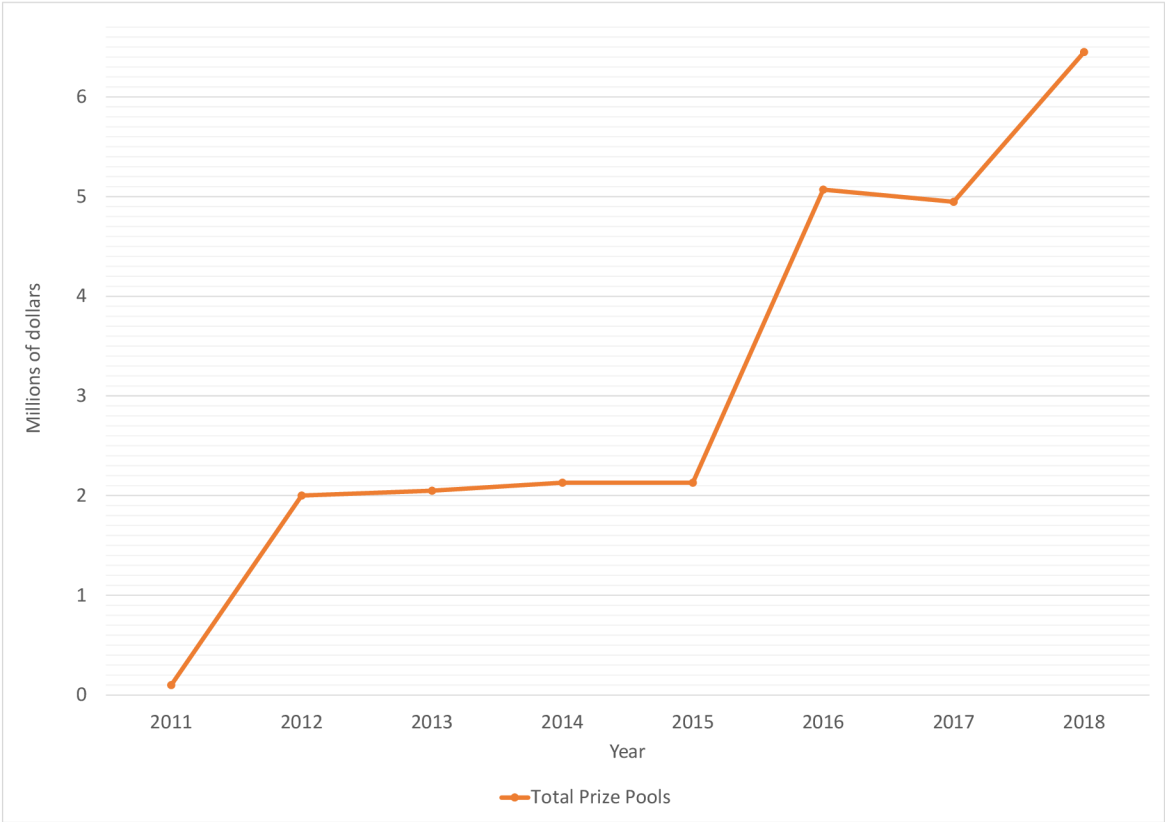

Figure 1.8: Growth in prize pools of World Championships from 2011-2018. Later data has not been published. Data taken from https://lol.fandom.com/.

Riot Games, the owner and developer of LoL, is an example of a "hands on" developer, meaning that they oversee all of the professional competition and therefore have total control over LoL e-sports [1, 26].

The LoL ecosystem is divided into 9 regions which host *regional leagues*, with a variety of lower leagues supporting and resourcing a region's top league with new talented players [26]. All the regions and their top leagues with shortcuts are shown in Table 1.2. All Regional Leagues except VCS are franchised, meaning that an organisation must buy into the league in order to compete, similar to traditional sports leagues such as the *National Hockey League* (NHL) or the *National Football League* (NFL) [1, 26].

The regular Regional League season is divided into two parts, known as *splits*. After the first split, the top teams from each region compete in an international tournament called the *Mid-Season Invitational* (MSI), and after the second split, the top teams from each region compete in the *World Championship* [26].

Region	League	League Shortcut
North America	*League of Legends Championship Series*	LCS
EMEA*	*League of Legends European Championship*	LEC
China	*LoL Pro League*	LPL
South Korea	*LoL Champions Korea*	LCK
Vietnam	*Vietnam Championship Series*	VCS
APAC**	*Pacific Championship Series*	PCS
Brazil	*Campeonato Brasileiro de LoL*	CBLOL
Latin America	*Liga Latinoamérica*	LLA
Japan	*LoL Japan League*	LJL

Table 1.2: League of Legends Regional Leagues. Data taken from [26].
**Europe, the Middle East and Africa* (EMEA) ***Asia-PACific* (APAC)

1.3 Application Programming Interface

An *Application Programming Interface* (API) is a way for two different applications, or modules of an application, to communicate with each other in a specific way, defined by a set of rules and protocols, usually written in API documentation [27]. The side that sends requests for data is called the *client*, and the other side that receives requests and returns the requested data is called the *server* [28, 29].

1.3.1 Representational State Transfer API

There are many different types of API, but the most commonly used API today is the *Representational State Transfer* (REST) API [27]. REST is a set of *architectural principles* that an API must implement in order to be called a REST API or RESTful API [28]. The six principles of the REST API are a uniform interface, client-server decoupling, statelessness, cacheability, layered system architecture and code on demand (optional) [29]. A *uniform interface* ensures consistent API requests for the same resource, while *client-server decoupling* maintains the independence of client and server applications [29]. *Statelessness*, meaning that servers do not store client data between requests, is a key feature of the REST API [27]. *Cacheability* improves performance and scalability by allowing resources to be cached on the client or server side [29]. *Layered system architecture* accommodates potential intermediaries in the communication loop between client and server applications [29]. *Code on demand*, although optional, allows

executable code to be sent in certain cases to run only when needed [29]. The sum of these principles is shown in Table 1.3.

Data between server and client travels via *Hyper Text Transfer Protocol* (HTTP) and is received in many different formats such as *JavaScript Object Notation* (JSON), *eXtensible Markup Language* (XML), or plain text, etc [28].

Principle	**Explanation**
Uniform Interface	Consistent API requests for identical resources
Client-Server Decoupling	Independence between client and server applications
Statelessness	No server-side sessions or client request data storage
Cacheability	Resource caching on client or server side to improve performance and scalability
Layered System Architecture	Design accommodating intermediaries in client-server communication
Code on Demand (Optional)	Provision for sending executable code when required

Table 1.3: *Representational State Transfer Application Programming Interface* principles and their short explanations.

1.3.2 Used APIs

Riot Games API

The *Riot Games API* is a REST API with documentation available at [30] and [31]. The Riot Games API always returns data in JSON format [30]. They use their internal identifiers, such as the *Player Universally Unique IDentifier* (PUUID) for player accounts, which are then used in the application.

The Riot Games API can be accessed through different types of keys, development, personal and production [30]. I have always accessed the API using a *personal API Key*. Personal API keys are intended for developer products or a small private community and have a low rate limit and expire after 24 hours [30].

Leaguepedia API

The *Leaguepedia API* is a REST API with documentation available at [32]. The Leaguepedia API returns data in JSON format [32].

Basic access to the Leaguepedia API does not require authentication and has a limit of 500 results per query [32].

1.4 Object-Oriented Programming

Object-Oriented Programming (OOP) is a fundamental programming paradigm that is essential to understand before writing programs in any OOP language. Programs consist of two elements: code and data, and can be organised around either [33]. The process-oriented model focuses on code acting on data, while OOP organises a program around data and its interfaces [33].

OOP uses abstraction to manage complexity. Hierarchical classifications help to break down complex systems into manageable layers [33]. For instance, a smartphone can be perceived as a singular device composed of various subsystems, including the camera, battery, and operating system. This methodology can also be employed in software development, by breaking down information into constituent objects.

OOP is at the heart of many modern programming languages, and understanding how these concepts are translated into programs is crucial. It provides a powerful paradigm for creating programs that can adapt to changes throughout the lifecycle of a software project [33]. With well-defined objects and clean interfaces, it becomes easier to maintain or replace parts of an older system.

1.4.1 OOP Principles

Encapsulation

Encapsulation binds code and data together, protecting them from outside interference and misuse [33]. It can be thought of as a protective shell that only allows access to the code and data inside through a well-defined interface [33]. For example, an automatic gearbox in a car encapsulates various data and the driver can only interact with it through the gear lever, a unique interface.

In programming, encapsulation is based on classes, which define the structure and

behaviour shared by a set of objects [33]. *Classes* consist of member *variables* (data) and member *methods* (code) that operate on the data. By marking methods or variables as *private* or *public*, a class can hide the complexity of its implementation [33]. Public methods represent the *interface*, so what external users need to know and use, while private methods and data can only be accessed by members of the class. The public interface should be carefully designed to avoid exposing too much of the inner workings of a class. An encapsulation scheme is shown in Figure 1.9.

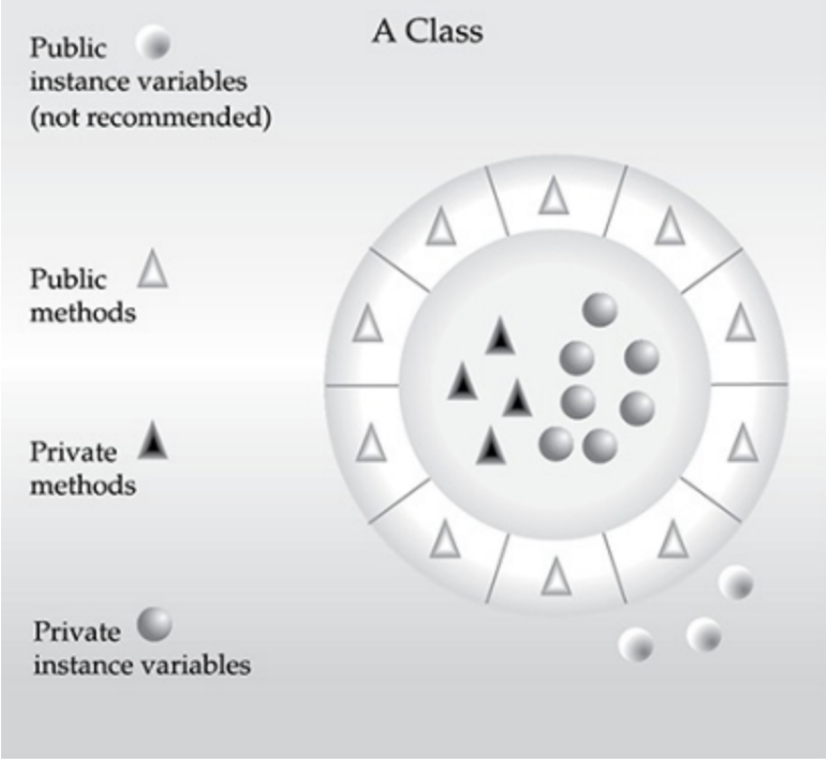

Figure 1.9: Encapsulation of private instance variables using public methods. Taken from [33].

Inheritance

Inheritance allows one object to take on the properties of another, supporting hierarchical classification [33, 34]. Without hierarchies, each object would have to explicitly define all of its properties [33, 34]. However, through inheritance, an object only needs to define its unique properties within its class, inheriting general properties from its parent [33, 34]. This mechanism allows an object to be a specific instance of a more general case [33, 34].

People typically perceive the world as consisting of hierarchically related objects, such as animals, with attributes such as size, intelligence and type of skeletal system, and behaviours such as eating, breathing and sleeping, which would be a class definition of animals. More specific classes of animals, such as mammals, have additional attributes or methods, such as gestation period or breastfeed, which make them a subclass of animals and animals their superclass.

Mammals, as more specific animals, inherit all animal attributes. A deeply inherited subclass inherits all the attributes of every ancestor in the class hierarchy [33]. Inheritance interacts with encapsulation; if a class encapsulates certain attributes, any subclass will have the same attributes, plus any it adds for its specialisation [33]. Scheme of inheritance at work is shown in Figure 1.10. This concept allows object-oriented programs to increase in complexity linearly rather than geometrically, because a new subclass inherits all the attributes of its ancestors without causing unpredictable interactions with most of the system's code [33].

Polymorphism

Polymorphism, a concept originating from the Greek words for "multiple shapes", is a characteristic that enables an interface to be utilized for a broad range of actions, with the specific action being determined by the particular context [33, 34]. Consider a program that necessitates three kinds of mathematical operations (addition, subtraction, and multiplication) for various data types such as integers, decimals, and complex numbers. The algorithm for implementing each operation stays consistent, regardless of the data types involved. In a non-object-oriented language, three separate sets of operation routines with distinct names would be necessary. However, polymorphism permits a universal set of operation routines to share the same names.

Often encapsulated by the phrase "one interface, multiple methods", *polymorphism* implies that a single, versatile interface can be created for a collection of related tasks, thereby reducing complexity [33, 34]. The *compiler* is responsible for selecting the specific action or method for each situation, eliminating the need for manual selection

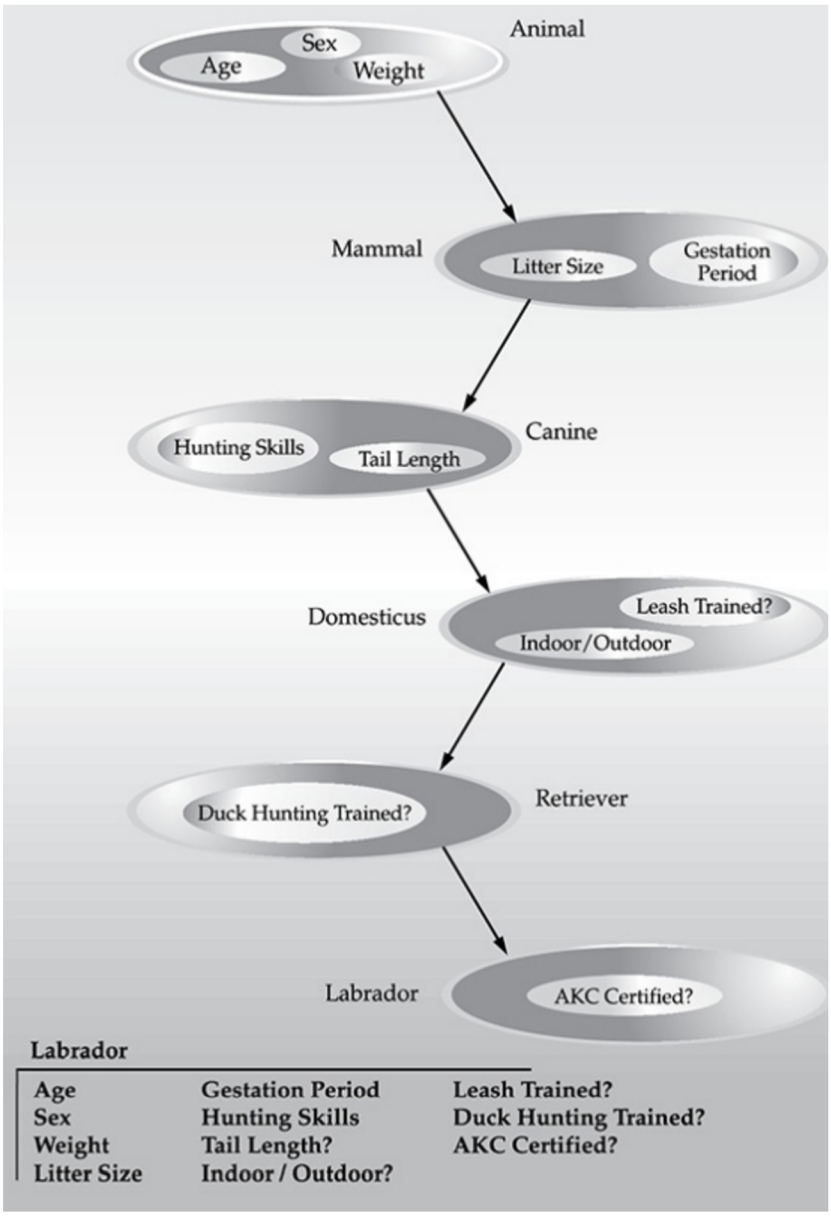

Figure 1.10: Inheritance: Example of Lanrador inheriting properties from its ancestors. Taken from [33].

by the programmer, who only needs to remember and use the generic interface [33].

Using the dog analogy, a dog's sense of smell is an example of polymorphism. When a dog smells a cat, it barks and chases it; when it smells food, it salivates and goes to its bowl. In both cases, the same sense of smell is at work, the difference being the type of data the dog's nose is responding to. This concept can be applied in a similar way to methods within a programme.

1.5 Java Programming Language

The *Java programming language* is a high-level language that according to [33, 34, 35] can be described by all of the following keywords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java is designed to be easy for professional programmers to learn and use effectively [33]. Those with programming experience, particularly in *C++*, will find it simple to master [33, 35]. Java is automatically removing unrefenreced objects and discarded complex and barely used features [35].

Secure

Dynamic and networked applications, while desirable, can raise significant security and portability concerns [33]. To ensure that downloaded applications don't cause harm, Java confines them to their execution environment, preventing unauthorised access to system resources [33]. This protection provides a level of confidence that programs can be downloaded and run without causing harm, and is arguably one of Java's most innovative features [33].

Portable

The diversity of computers and operating systems on the Internet requires portability [33]. Java allows applications to run on different systems using the same security mechanism [33]. The goal is to allow the same application code to run on different computers without the need for multiple versions, ultimately simplifying the process of creating portable executable code [33].

Object-Oriented

Java is fulfils all the concepts of OOP. Java was created as an independent language, unbound by source code compatibility with its predecessors [33]. This resulted in a clean, pragmatic approach to objects that balances the "everything is an object" paradigm with a more practical model [33]. Java's object model is easily extensible, while *primitive types* remain non-object for optimal performance.

Robust

It employs *robust memory management* while eliminating *pointers* to prevent security issues [35]. Java restricts certain areas to help developers catch errors early, while eliminating common programming errors through its strictly typed nature [33]. Memory management issues and exceptional conditions, common causes of program failure, are addressed by Java's automatic memory allocation and deallocation and object-oriented *exception handling* [33].

Multithreaded

Java supports *multi-threaded* programming to meet the demands of creating interactive, networked programs [33, 35]. Its sophisticated multi-process synchronisation enables the creation of smooth-running interactive systems, allowing developers to focus on the behaviour of the program rather than the multitasking subsystem [33].

Architecture-Neutral

Java addresses code longevity and portability by designing the Java language and the *Java Virtual Machine* (JVM) to be architecture-neutral [33]. The goal of "write once; run anywhere, anytime, forever" has been largely achieved through Java's design choices [33]. For example, unlike C, Java's int data type uses 4 bytes of memory on both 32-bit and 64-bit systems in Java [35].

Interpreted and High Performance

Java's cross-platform compatibility is achieved through Java *bytecode*, an intermediate representation that doesn't sacrifice performance [33]. Java bytecode can be easily translated into native machine code, and a just-in-time compiler ensures high performance [33, 35].

Distributed

Java accommodates the *distributed* environment of the Internet by handling *Transmission Control Protocol/Internet Protocol* (TCP/IP) protocols and supporting *Remote Method Invocation* (RMI), allowing networked method invocation [33, 35]. Accessing resources using a *Uniform Resource Locator* (URL) is very similar to accessing a file.

Dynamic

Java programs contain extensive *runtime* type information for object access verification and resolution [33]. This enables safe and efficient dynamic linking of code, which is essential in the Java environment where bytecode fragments can be dynamically updated on a running system [33].

1.6 Java libraries

1.6.1 Hibernate

Hibernate is an external Java library that allows an application to persist data in a *relational database* [36, 37]. It is an *Object-Relational Mapper* (ORM), which means that it represents relational data as simple Java objects that can be accessed through a *session manager*, making it much easier to write applications that interact with relational databases [36, 37]. The documentation of Hibernate is availible at [38].

Hibernate addresses several challenges and eases the pain of managing resources and database connections. By working with a `Session` object, it simplifies resource management and exception handling [36]. Hibernate also manages object-to-database table mapping, database schema construction, and relationships between objects, such as storing a list of addresses for an object [36].

In addition, Hibernate can map new types to the database and provides customisable serialisation options [36]. While Hibernate's startup time may be longer than direct *Java DataBase Connectivity* (JDBC) code, this is negligible in the context of the overall runtime of an application [36]. Its maintenance and object management benefits outweigh any initial configuration time.

Hibernate is an ideal solution for persisting Java objects in databases, replacing ad hoc approaches, or serving as a persistence engine in applications without database persistence [36]. Using Hibernate maintains flexibility in an application design decisions, including database selection.

Hibernate can be accessed directly from any Java application or through other frameworks such as *Swing*, servlets, portlets or *Jakarta Server Pages* (JSP) pages [36]. It is typically used to create or replace a data access layer in applications. Hibernate supports Java standards such as *Jakarta Server Pages* (JSP), *Java EE Connector Architecture* (JCA) and *Java Naming and Directory Interface* (JNDI), allowing for run-time configuration and integration with various frameworks [36]. It uses standard JDBC drivers to access relational databases and sits on top of the JDBC layer [36, 37].

Many Java web and application frameworks, such as *Spring*, integrate with Hibernate because of its simple and clean API [36]. In any environment, configuration details have to be defined and then used to create a `SessionFactory` object. `Session` objects are instantiated from the `SessionFactory` and provide access to Hibernate's database representation.

After incorporating Hibernate into an application, there's no need to modify existing Java object model with persistence markers or other hints [36]. Hibernate works with standard Java objects created using the 'new' operator or by other objects. These objects can be classified into two groups: those with Hibernate entity mappings and those not directly acknowledged by Hibernate. Correctly mapped entity objects possess fields and properties that may be either mapped entities, collections of entities, or value types.

According to [36] mapped objects in Hibernate can be in one of these four states:

- Transient
- Persistent
- Detached
- Removed

Transient object

Transient objects are independent of Hibernate and don't have a database representation [36]. Figure 1.11 shows the scheme of the transient object. To persist changes to a transient object, a session has to be asked to save it to the database. Once Hibernate has assigned an identifier to the object, it becomes persistent [36]. There also exists `@Transient` annotation for property of object which makes it not managed or affected by Hibernate.

Figure 1.11: Transient objects are independent of Hibernate. Taken from [36].

Persistent object

Persistent objects have a database representation, and Hibernate manages their persistence [36]. Figure 1.12 shows the scheme of the persistent object. When changes are made to a persistent object, Hibernate updates the database representation when the application commits the changes [36].

Figure 1.12: Persistent objects are maintained by Hibernate. Taken from [36].

Detached object

Detached objects have a database representation, but aren't connected to it [36]. Figure 1.13 shows the scheme of the detached object. Changes to detached objects don't affect the database and vice versa. Detached objects can be created by closing the associated session or by evicting them using the session's `evict()` method [36]. To persist changes to a detached object, it has to be reattached to a valid Hibernate session using methods such as `load()`, `refresh()`, `merge()`, `update()` or `save()` [36].

Figure 1.13: Detached objects exist in the database but are not maintained by Hibernate. Taken from [36].

Removed object

Removed objects are managed by Hibernate, but have been passed to the session's `remove()` method [36]. When changes are committed to the session, the corresponding database entries for removed objects are deleted.

Entities

Entities are Java objects with mappings that allow them to be stored in a database, specifying how fields and properties are stored in database tables [36]. It's possible to represent a Java class in the database in different ways, such as having a single class for users but two different tables. Hibernate can handle these scenarios, which are common in legacy systems [36].

Objects that represent entities are standard Java classes with entity names that are typically the same as the class type. However, it can be changed using mappings or annotations to distinguish between objects of the same type mapped to different tables. Some Session API methods require an entity name to determine the correct mapping. If omitted, the method assumes that the entity name is the same as the class name, or that no distinction is required.

Identifiers

Identifiers, or identity columns, correspond to *primary keys* in relational databases [36]. They are of two types: natural and artificial [36]. Natural identifiers have application meaning, such as user identifiers or social security numbers, while artificial identifiers have arbitrary values, such as database-generated identity columns.

Artificial identifiers may be preferred because they can be smaller in memory than natural identifiers, remain unchanged during the natural lifecycle of the data, and are easy to use [36]. In Hibernate, an object attribute is marked as an identifier with the `@Id` annotation.

The `@GeneratedValue` annotation indicates that this is artificial identifier and that Hibernate manages its generation. There are five different generation types: identity, sequence, table, auto, and none [36].

Associations

Associations are references between entities, either directly as an embedded property or indirectly through collections [36]. These associations are represented by *foreign*

keys in the underlying tables, which rely on identifiers, making small artificial keys preferable [36].

Associations can be unidirectional, where only one entity has a reference to the other, or bidirectional, where both entities reference each other. In associations, only one participating class manages the relationship [36]. It is possible to specify the entity that manages the association using the mappedBy attribute of the association annotation [36, 37].

Table 1.4 shows how to select the relationship owner in a bidirectional association. The association owner is only concerned with managing foreign keys in the database [36, 37].

Type of Association	**Options**
one-to-one	Either end can be made the owner, but one (and only one) of them should be; if not specified, it will end up in a circular dependency.
one-to-many	The many end must be made the owner of the association.
many-to-one	This is the same as the one-to-many relationship viewed from the opposite perspective, so the same rule applies: the many end must be made the owner of the association.
many-to-many	Either end of the association can be made the owner.

Table 1.4: Marking the Owner of an Association. Taken from [36].

1.6.2 Gson

Gson is a external Java library designed to convert Java objects to their JSON representation and JSON strings to equivalent Java objects [39]. It can handle Java objects, including pre-existing ones with no source code available [39].

Gson's goals include providing easy-to-use mechanisms such as `toString()` and *constructor* (factory method) for seamless *Java-to-JSON* and *JSON-to-Java* conversions, allowing pre-existing immutable objects to be converted in both directions, allowing custom representations for objects, supporting complex objects at any level, and producing compact, readable JSON output [39].

In terms of performance and scalability, Gson has demonstrated impressive metrics

on a desktop running numerous other processes alongside the tests [39]. Using the `PerformanceTest` class, it was found to deserialise strings over 25 MB, serialise a collection of 1.4 million objects, and deserialise a collection of 87,000 objects [39]. In addition, Gson 1.4 increased the deserialisation limit for byte arrays and collections from 80 kB to over 11 MB [39].

Originally developed for internal use at Google, Gson is now used by several public projects and companies [39]. To use Gson, the primary class is `Gson`, which can be instantiated by simply calling `new Gson()`. Alternatively, the `GsonBuilder` class provides the ability to create a Gson instance with settings such as version control [39]. The `Gson` instance retains no state during JSON operations, allowing the same object to be reused for multiple JSON serialisation and deserialisation tasks [39].

1.6.3 Bucket4J

Bucket4j is a external Java-based rate-limiting library based primarily on the *token-bucket algorithm*, the de facto standard for rate-limiting in the IT industry [40]. It goes beyond a simple token-bucket implementation, incorporating several useful extensions not found in traditional token-bucket interpretations, such as multiple limits per bucket and overdraft capabilities [40].

Bucket4j provides absolute precision by using integer arithmetic and avoiding floats or doubles, thus protecting end-users from potential rounding errors [40]. It also provides an efficient concurrency implementation that scales well in multi-threaded scenarios through a default lock-free implementation, while providing alternative concurrency strategies when needed [40]. The API minimises the footprint of the garbage collector by using primitive types wherever possible, and includes a pluggable listener API for monitoring and logging, a comprehensive diagnostics API for examining internal state, and dynamic configuration management [40]. It is licensed under the *Apache License 2.0* [41].

Bucket

A bucket in Bucket4j is a rate limiter based on the concepts of the token bucket algorithm [40]. It consists of the `BucketConfiguration`, which specifies an immutable set of limiting rules to be used during operation, and the `BucketState`, which stores mutable state information, such as the current number of tokens available [40]. A `Bucket` object is created using the `BucketBuilder` API [40].

BucketConfiguration

`BucketConfiguration` is an object representing collection of limits used by the bucket during its operation [40]. It is immutable, so bounds cannot be added to or removed from an existing configuration [40]. However, a new configuration instance can replace the old configuration by calling `bucket.replaceConfiguration(newConfiguration)` [40]. The `ConfigurationBuilder` API allows direct creation of `BucketConfiguration` if required [40].

Limitation/Bandwidth

Bucket limits are expressed as bandwidths characterised by capacity, refill and initial tokens [40]. Capacity refers to the number of tokens in a bucket, while refill specifies the rate at which tokens are replenished after consumption [40]. The initial number of tokens in each bandwidth can also be specified, with defaults equal to capacity [40]. In addition, identifiers can be assigned to bandwidths for on-the-fly configuration replacement, particularly when multiple bandwidths are associated with a single bucket [40].

Refill

The *refill* refers to the speed at which tokens are regenerated, with three types available: greedy, interval and intervallyAligned [40]. *Greedy* regenerates tokens as quickly as possible, *Interval* regenerates tokens at the end of a specified period, and *IntervallyAligned* regenerates tokens in an intermittent manner, but allows the first refill

time to be specified, allowing clear interval boundary configuration [40].

BucketState

The `BucketState` object stores mutable state information such as the number of tokens available and the timestamp of the last refill [40]. It is not normally interacted with directly, except in cases where low-level diagnostic API access is required [40].

BucketBuilder

The `BucketBuilder` object is a key component that utilises a fluid design to efficiently construct local buckets [40]. It enables the creation of buckets with adjustable synchronisation strategies and precision levels, while ensuring future adaptability and a modern library design pattern [40].

1.6.4 Swing

Swing is one of the *Java Foundation Classes* (JFC) libraries used to develop window-based applications [42, 43]. Based on the *Abstract Windowing Toolkit* (AWT) API, Swing is developed entirely in Java [42, 44].

Swing is promoted as a collection of customisable graphical components with a run-time controllable look-and-feel [43]. But Swing is more than that; it's a next-generation *Graphical User Interface* (GUI) toolkit for large-scale Java application development, offering a variety of powerful components that can be easily modified or extended to control appearance and behaviour [43].

Today, developers often use Swing as their preferred framework for creating GUI components, as it offers more powerful and versatile options compared to the AWT [33, 44]. Swing has been a popular choice among Java programmers for many years [33].

Swing components

Swing components are predominantly lightweight, meaning they're entirely Java-based and don't map to platform-specific counterparts [33]. This makes them more efficient, flexible and consistent across platforms [33]. Swing supports *Pluggable Look And Feel* (PLAF), which allows the appearance of a component to be separated from its logic [33]. This separation allows the appearance of a component to be changed without affecting its functionality or causing side effects [33].

PLAFs provide the ability to maintain consistency across platforms, emulate specific platform styles, design custom appearances, and dynamically change the look and feel at runtime [33]. Java provides several looks and feels, such as *metal* and *nimbus*, which are available to all Swing users [33]. The metal look and feel, also known as the Java look and feel, is platform-independent and serves as the default [33].

Swing components, with the exception of four top-level containers, are derived from the `JComponent` class, which provides common functionality such as pluggable look and feel support [33]. `JComponent` inherits from the AWT container and component classes, making Swing components compatible with AWT components [33]. Swing component classes can be found in the `javax.swing` package, where each class name begins with the letter 'J' (e.g. `JLabel`, `JButton`, and `JScrollBar`) [33]. All the GUI components of the Swing library can be seen in Figure 1.14.

1.6.5 Abstract Window Toolkit

Java's *Abstract Windowing Toolkit* (AWT) is the original GUI, containing numerous classes and methods for creating windows and basic controls [33]. Although it has been superseded by more powerful frameworks such as Swing, AWT remains essential because it underpins Swing and many AWT classes are used directly or indirectly [33]. Gaining a basic understanding of AWT is crucial to using Swing effectively, and may still be appropriate for small programs that require minimal GUI usage [33]. I only use a few classes from the AWT library and list them all below.

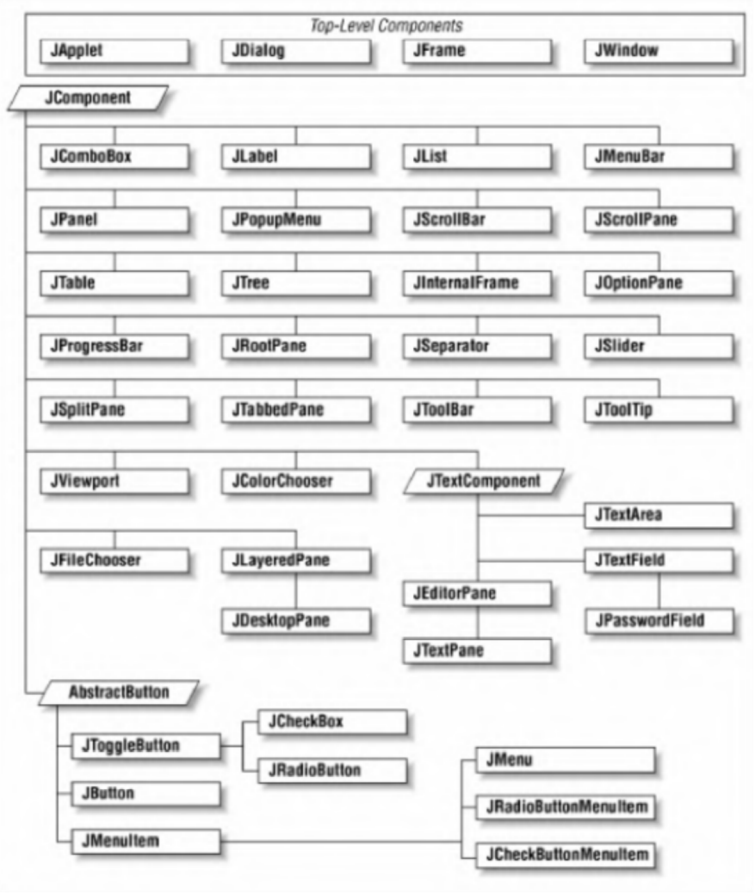

Figure 1.14: GUI Components of Java Swing library. Taken from [43].

BufferedImage

The `BufferedImage` class, a subclass of the `java.awt` package, represents an image that has an accessible buffer containing its data. It consists of a `ColorModel` and a `Raster` that holds the image data [45]. The `SampleModel` of the `Raster` must have the appropriate number and types of bands that correspond with the requirements of the `ColorModel` for representing colour and alpha components [45]. All objects created from the `BufferedImage` class possess an upper left corner coordinate set at (0, 0) [45]. Consequently, any `Raster` utilized for constructing a `BufferedImage` needs to have its `minX` and `minY` values set to 0 [45].

This particular class is heavily dependent on the methods for fetching and setting data provided by the `Raster`, as well as the colour characterisation methods provided by the `ColorModel` [45].

Graphics2D

The `Graphics2D` class, an extension of the `Graphics` class, provides sophisticated control over geometry, coordinate transformations, colour management and text layout, making it essential for rendering 2D shapes, text and images on the Java platform [33, 45]. `Graphics2D` class operates in a device-independent coordinate system called *User Space*, with an associated `AffineTransform` object defining the conversion to device-dependent coordinates in *Device Space* [45]. Rendering operations are performed in four phases: determining the content to be rendered, restricting the operation to the current clip, determining the colours to be rendered, and applying the colours to the drawing surface using the current composite attribute [45].

Rendering operations can be categorised into shape operations, text operations and image operations [45]. Shape operations involve creating a new shape object that outlines the specified shape, transforming the shape and extracting its outline [45]. Text operations involve determining the glyphs needed to render the given string, querying the current font for outlines, and filling character outlines [45]. Image operations involve defining the region of interest, transforming the bounding box from user space to device space, and determining the colours to render based on the source-destination coordinate mapping [45].

AlphaComposite

The `AlphaComposite` class in `java.awt` implements fundamental alpha compositing rules for blending source and destination colors in graphics and images. These rules are based on the twelve primary rules described by *T. Porter* and *T. Duff* in "Compositing Digital Images" [45]. This class also extends the standard equations by incorporating an additional alpha value that modifies the opacity or coverage of source pixels.

It is crucial to understand that the equations work on color components premultiplied by their corresponding alpha components [45]. As `ColorModel` and `Raster` classes permit pixel data storage in both *premultiplied* and *non-premultiplied* forms, all input data must be normalized into premultiplied form before applying the equations

[45]. `AlphaComposite` introduces an extra alpha value applied to the source alpha, as if an implicit `SRC_IN` rule were first applied to the source pixel against a pixel with the indicated alpha [45]. This is achieved by multiplying both the raw source alpha and raw source colours by the alpha in the AlphaComposite [45].

1.7 Web tools

1.7.1 Hyper Text Markup Language

HyperText Markup Language (HTML) is used to create web pages and web applications [46]. It involves hypertexts, which provide references to different web pages. As a markup language, it incorporates layout and formatting standards into textual documents, transforming text into interactive and dynamic elements such as images, tables or links using various tag elements. A tag element is specified with a name and functionality and usually has an opening tag and closing tag, for example `<p>paragraphText</p>` are paragraph tags with their content between them. Together with *Cascading Style Sheets* (CSS), individual tags and elements can be styled into infinite possibilities.

In summary, HTML is a markup language used to create visually appealing web pages with styling, displayed in an organised format on web browsers, and composed of various HTML tags containing different content [46].

1.7.2 JavaScript

JavaScript (JS) is a scripting or programming language that allows complex functionality to be built into web pages. If a web page presents more than static information and offers features such as real-time content updates, interactive maps, animated 2D/3D visuals or scrolling video players, JS is likely to be responsible [47]. As the third layer of the standard web technology stack, it complements the other two layers, HTML and CSS [47].

1.7.3 C3

C3 is a JS library and a preferred choice for creating charts in JS because it offers convenience, customisation and control [48]. By simplifying the process of creating D3-based charts, C3 eliminates the need to write complex D3 code [48]. It also assigns classes to each element, allowing users to define custom styles and extend the structure using D3 directly [48]. C3 also provides numerous APIs and callbacks to access the state of the chart, allowing updates to be made even after the chart has been rendered, making the integration of charts into an application more seamless [48].

2. Practical part

2.1 Front-end to Back-end View

The application is divided into separate components based on their purpose. This view shows whether the application component belongs to the *front-end* or the *back-end*. The view is represented by the diagram in Figure 2.1. The diagram also shows how the components of the application use each other and how they are connected to external actors. The `DefaultBrowser` has *external* stereotype to point out that it is not a part of the application.

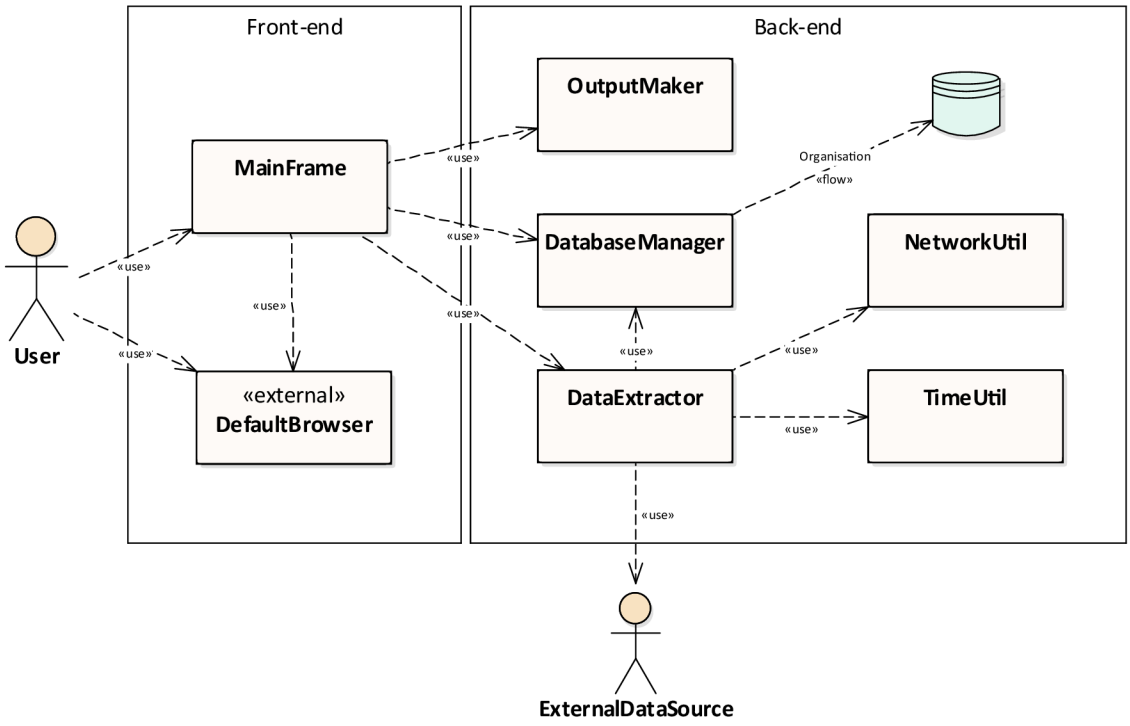

Figure 2.1: Diagram of the component's application and other external actors. With boundaries defining which domponent belongs to front-end and back-end. Created in Enterprise Architect.

2.2 Application Use Case View

The application's *use case* view represents how the user typically uses the application (each usage is referred to as a use case), by whom each use case is realised, and what other use cases are *included* or *extend* a particular use case. This view divides the application into separate *components*, each component realising one or more use cases. The complete use case view, represented by the use case diagram, is shown in Figure 2.2. *Actors* inside the application boundary represent components of the application such as *MainFrame* or *DataExtractor* and actors outside the boundary represent external actors such as *ExternalDataSource* or *Database*. The Figure 2.2 also shows that the user only interacts with the MainFrame.

Each component in the application code is represented by a Java class placed in the *components* package. The application code has been developed component by component in the order of typical user behaviour when using the application. This typical user behaviour with the corresponding actions from the application is shown in Figure 2.3.

2.2.1 MainFrame

The first component of the application, called `MainFrame`, is responsible for the use cases that include all interactions between a user and the application. It is represented by a Java class that extends `JFrame` with `GridBagLayout` to centre its GUI components vertically and horizontally. The only GUI component directly in this object is `JPanel` with `BorderLayout`, which is used as a wrapper for all other GUI components. The hierarchy of all active GUI components and their containers is shown in Figure 2.4. Active GUI components are those with which the user can interact, so `JLabel` instances are not included.

By adding an `ActionListener` instance to an active GUI component using the `addActionListener()` method, user interaction with the application through active GUI components is achieved. The parameter of this method is a *lambda expression* with a function that first creates and initialises a *modal dialogue* that forbids any

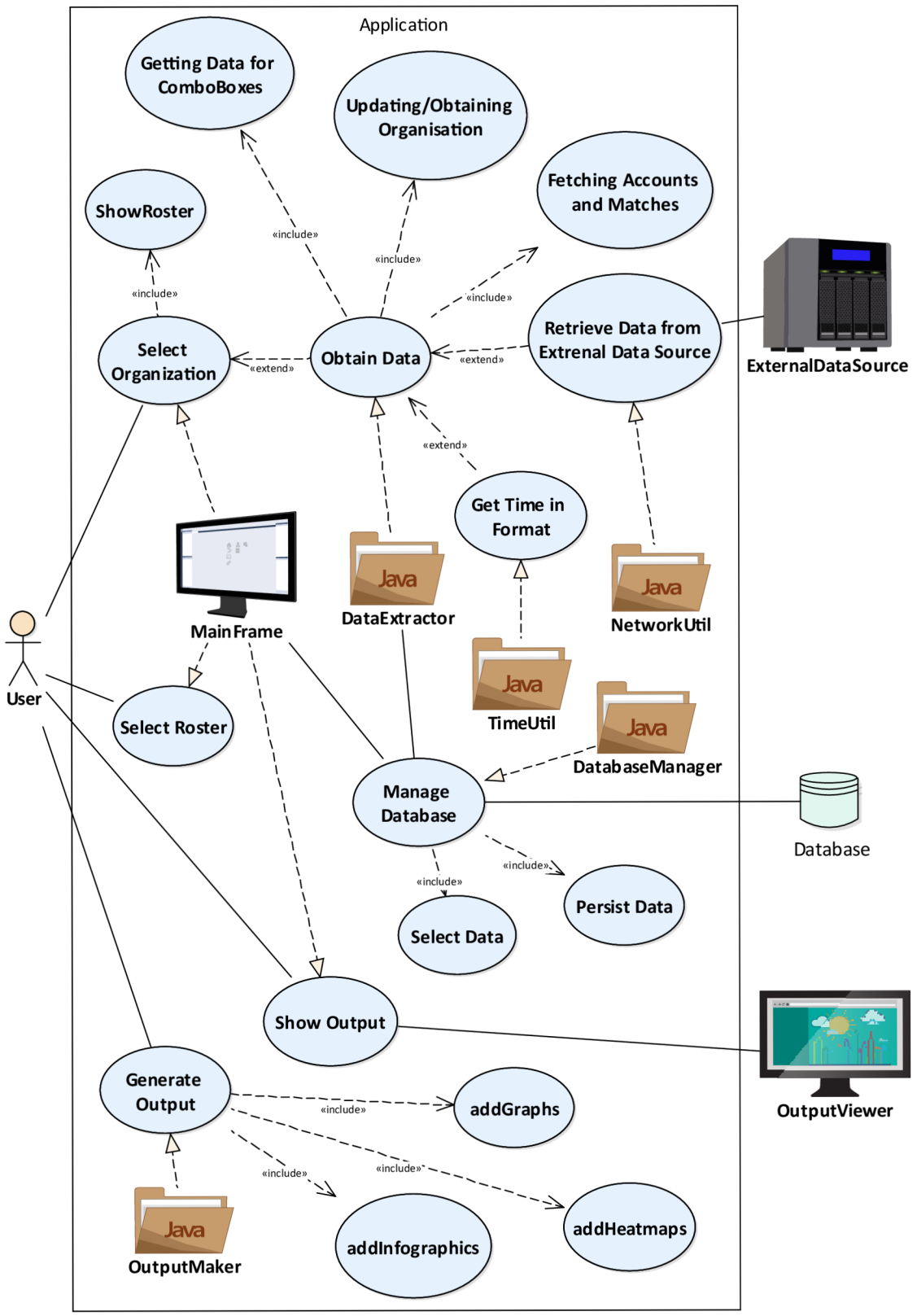

Figure 2.2: UseCase Diagram of the application and other external actors. Actors in the boundary represent modules of the application. Created in Enterprise Architect.

Figure 2.3: Process Diagram of typical user behaviour when using the application and corresponding actions from the application. Created in Enterprise Architect.

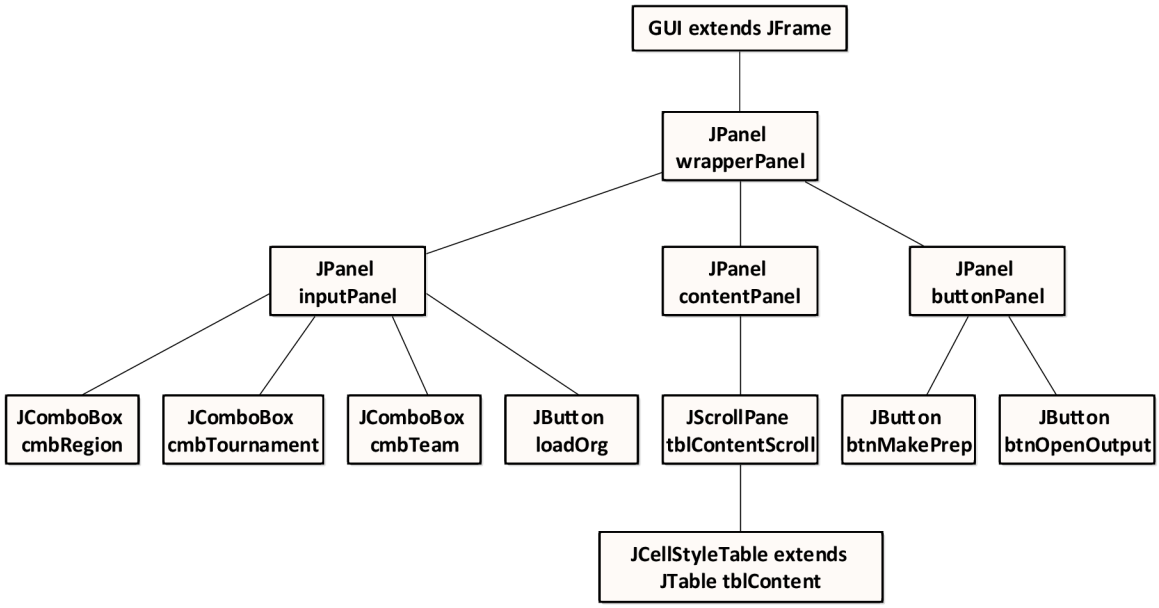

Figure 2.4: Hierarchy of all active GUI components and their containers. Created in Enterprise Architect.

interaction between the user and the `MainFrame`, while the instance of `SwingWorker` executes a block of code that implements an action desired by the user. The block of code is defined in overridden method `doInBackground()` which is executed in different thread and it is also a place in code where I catch all exceptions to deal with them. The `done()` method executes after `doInBackground()` has finished, closing the modal dialogue and displaying an error message if an exception was thrown. The code of template for this situation is shown here:

```
guiComponent.addActionListener(e -> {
   JDialog waitDialog = new JDialog();
   initWaitDialog(waitDialog);
   SwingWorker<Void, Void> worker = new SwingWorker<>() {
      @Override
      protected Void doInBackground() {
```

```
            try {
               // Block of code which is executed by SwingWorker in another thread.
            } catch (Exception e) {
               // Block of code handling exceptions
            }
         return null;
         }
         @Override
         protected void done() {
            waitDialog.dispose();
            if (errorMessage != null) {
               JOptionPane.showMessageDialog(errorMessage);
                       }
         }
      };
      worker.execute();
});
```

The `MainFrame` component realises the use cases listed below, which are described in detail in the following chapters.

- Selecting the Organisation
- Selecting the Organisation's Roster
- Launching the Output Generation
- Opening the Output File

Selecting the Organisation

This use case is referred to as the "Select Organisation" in Figure 2.2. It is implemented using three `JComboBox` instances to minimise the human error of misspelling the organisation name and also to make it easier for the user. The first `JComboBox`, called `cmbRegion`, allows the user to select the region where the desired organisation is located. At the moment, the application only allows one region to be selected, which is EMEA, because one of the external data sources only covers this region.

Once a region has been selected, the application uses the `DataExtractor` component to load all the tournament names from that region into the second `JComboBox` named `cmbTournament`. See chapter 2.2.2 for more details on how the `DataExtractor` obtains and returns this data. Similarly, once a particular tournament has been selected, the application uses the `DataExtractor` component to load all the names of the organisations playing in that tournament into the third `JComboBox` named `cmbOrganisation`. Again, see chapter 2.2.2 for more details on how the `DataExtractor` obtains and returns this data. The user then selects the desired organisation from the third `JComboBox` and confirms his choice by pressing "Show organization's roster" `JButton`.

Once the button is pressed, the selected item from `cmbOrganisation` is passed as a parameter to the `loadRoster()` method. This method then calls the `DataExtractor` component via the `getOrganization()` method for the `Organisation` instance with all the required properties. See chapter 2.2.2 for more details on how the `DataExtractor` obtains and returns this data.

Selecting the Organisation's Roster

This use case is referred to as the "Select Roster" in Figure 2.2. After the `Organisation` instance is returned, the `showRoster()` method is called. This method creates an instance of the `RosterTableModel`, which extends the Swing's `DefaultTableModel` class and is specially modified to work as a cell-based model, meaning that it is possible to specify for each cell whether it is *editable*. The model is then passed as a parameter to the constructor of `RosterTableModel`, which extends `JTable` and is also specially modified to work as a cell-based table, meaning that it is possible to specify for each cell its *renderer* and *editor*. This cell-based concept is implemented so that when there are multiple player options for a position, the user can select who they want to prepare against in `JComboBox`, which is used as the editor and renderer for that player's cell. The table is then displayed on the `MainFrame` using `contentPanel.add()`.

Once the user has decided on the roster, he confirms his decision by pressing the "Confirm roster to load match data" button. Pressing this button calls the `makePrep()`

method of `MainFrame`, and because this method can take up to several minutes to complete, the modal dialogue also contains a `JTextArea` that displays the current status. The first thing this method does is to incorporate the user's roster decision into the data model via the `setStartingRoster()` method.

Launching the Output Generation

This use case is referred to as the "Generate Output" in Figure 2.2. As the application does not yet have all the required data in the `Organisation` instance, `makePrep()` first calls the `DataExtractor` via the `fetchAccountsToPlayer()` method for each player on the roster, then the `fetchMatchesToRoster()` method for selected roster and then fetchMatchesToAccount() method for each account of each player on the roster. See chapter 2.2.2 for details on how the `DataExtractor` obtains this data. Now, before any *data processing*, the `Organisation` instance with all the required data is passed as a parameter to the `DatabaseManager` component's `insertObject()` method to be stored in the database. See chapter 2.2.3 for more details on how the `DatabaseManager` persist data. Once the data is persisted, the `OutputMaker` class is called via the `makeHTMLOutput()` method to return the path to the *output file*. See chapter 2.2.4 for more details on how the `OutputMaker` creates output file.

Opening the Output File

This use case is referred to as the "Show Output" in Figure 2.2. The output file path is then passed to `MainFrame` component's `addLinkToPrep()` method, which replaces the `btnMakePrep` for the "Go to match preparation" JButton. When the user clicks on this button, their default internet browser will open and display the output file.

2.2.2 Data Extracting

The next component of the application, called the `DataExtractor`, is responsible for this use case and all the use cases contained within it. A simplified process for implementing this use case is shown in the collaboration diagram in Figure 2.5. It fulfils all

the data needs of the other components by transforming data from various external sources into the application's *data model*. The class diagram shows the data model in Figure 2.6. The component is represented by a *static* Java class. It is not technically static, but all methods and properties are static and the construct of the class is set to *private access modifier* to prevent any misuse of the class. The `DataExtractor` component realises the use cases listed below, which are described in detail in the following chapters.

- Getting data for Combo Boxes
- Obtaining the Organisation
- Updating the Organisation
- Fetching Accounts to Players
- Fetching Matches to Roster and Accounts

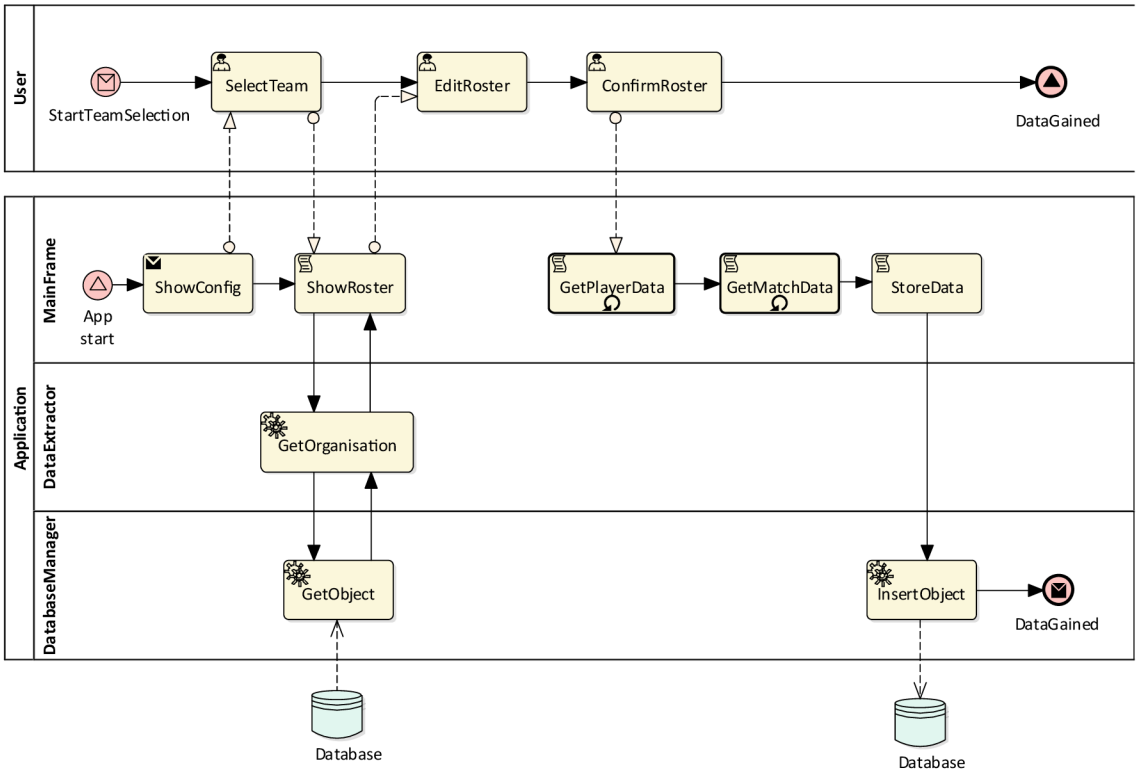

Figure 2.5: Collaboration Diagram of Simplified implemantaion of "Data Extracting" use case. Created in Enterprise Architect.

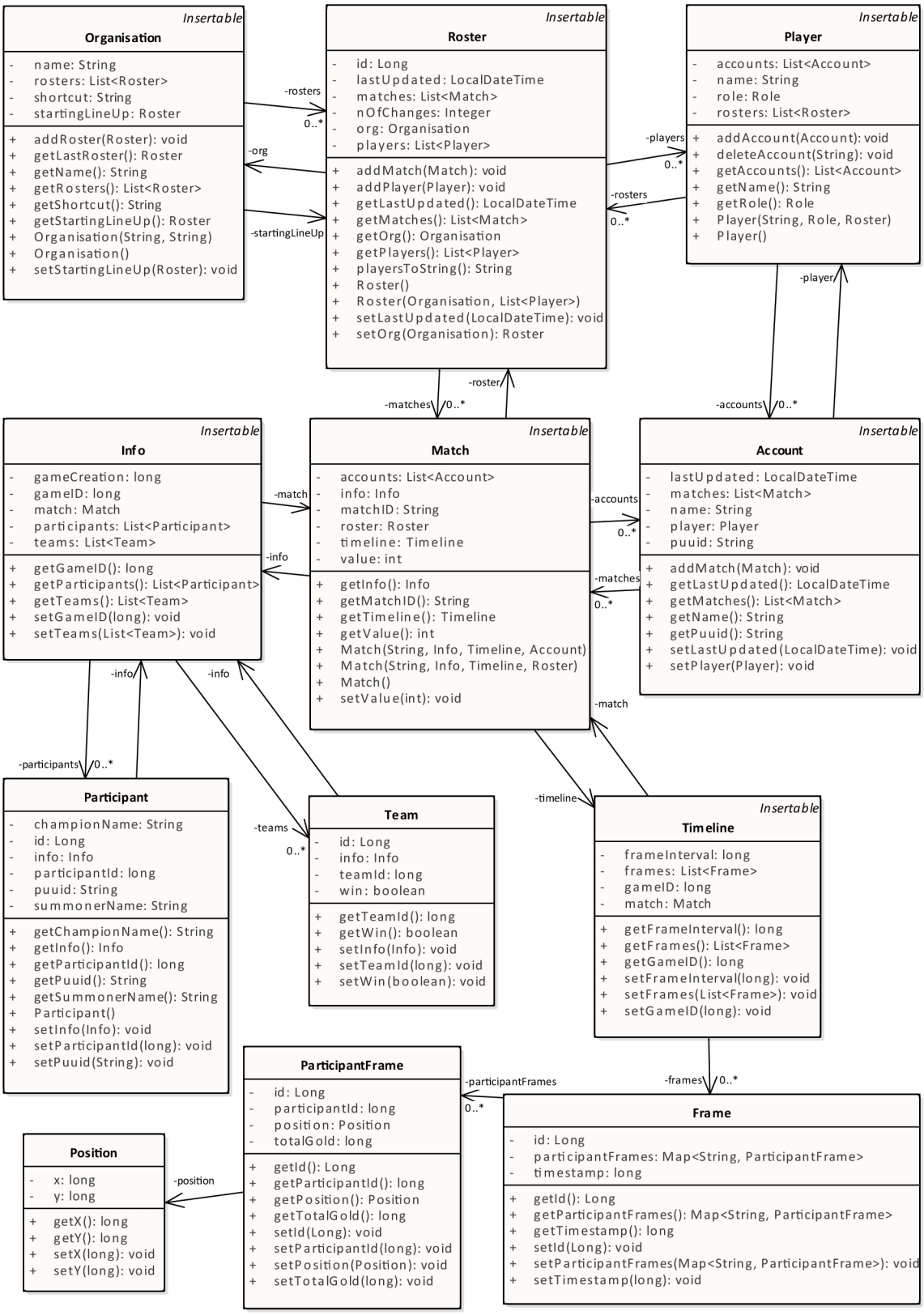

Figure 2.6: Class Diagram of Data Model. Created in IntelliJ IDEA.

Getting data for Combo-Boxes

This use case retrieves data for two `JComboBox` instances, the `cmbTournaments` and the `cmbTeams`, using the `getTournamentsByRegion()` and `getTeamsByTournament()` public methods when requested by the `MainFrame` component.

First, the URL string is built using predefined templates, parameters and methods from the `TimeUtil` component, which allows many different time formats to be built and returned. See chapter 2.2.6 for more details on how `TimeUtil` builds these time formats. The URL string contains a reference to the *Leaguepeadia API* as this is where the information is retrieved from.

Once the URL string has been built, it is passed as a parameter to the `NetworkUtil` component's method called `getJSONFromURLString()`, which will return the JSON with all the required information. See chapter 2.2.5 for details on how `NetworkUtil` gets this JSON. It then uses the Gson library to parse it into a `JsonObject`, which is then returned to the `MainFrame` component.

Obtaining the Organisation

When the `DataExtractor` component is asked for an instance of the `Organisation` class via the public method `getOrganisation()`, it then asks the `DatabaseManager` component to retrieve data from the database via the method `getObject()` with the name of the organisation as a parameter. See chapter 2.2.3 for more details on how `DatabaseManager` retrieves data from the database.

If the `Organisation` instance returned by the `getObject()` method is `null`, meaning that this organisation has never been retrieved and persisted, the constructor of the `Organisation` class is called with the name of the organisation and it's shortcut passed as parameters. The shortcut of the organisation is obtained using the `getOrganisationShortcut()` method, which, like similar methods described many times before, builds the URL string using the `getJSONFromURLString()` method, parses it using the Gson library, and extracts the shortcut from the `JSONObject`. The URL string contains a reference to the Leaguepeadia API as this is where the

information is retrieved from.

Once the instance of the `Organisation` has been constructed, the `getRoster()` method described above is called and a reference to the returned instance of the `Roster` class is added to the instance of the `Organisation` class.

Updating the Organisation

If the `Organisation` instance returned by the `getObject()` method is not `null`, meaning that this organisation was previously retrieved and persisted to the database, the `updateRoster()` method is called. This method retrieves the current roster via the `getRoster()` method.

The `getRoster()` method first builds a URL string from a predefine templates and parameters. The URL string contains a reference to the *LeaguepeadiaAPI* as this is where the information is retrieved from. Once the URL string has been built, it is passed as a parameter to the `NetworkUtil` component's `getJSONFromURLString()` method, which returns the JSON with all the required information. See chapter 2.2.5 for details on how `NetworkUtil` gets this JSON. It then uses the Gson library to parse it into a `JsonObject` from which arrays of player names and roles are extracted. The constructor of the `Roster` class is then called and the arrays are used to construct instances of the `Player` class, which are then inserted into the instance of `Roster` class. Once this is done, the instance of `Roster` class is returned to the `updateRoster()` method.

When the instance of `Roster` class is returned, the `updateRoster()` method compares the old roster and this new roster, and if there are any changes, the new roster is added to the organisation. If the rosters are the same, the new roster is discarded.

Fetching Accounts to Players

When the `DataExtractor` component is asked to obtain accounts data for each player via the public method `fetchAccountsToPlayer()`, the private method `getAccounts()` is first called with an instance of the `Player` class passed as a parameter.

This method first builds a URL string from a predefined template and parameters. The URL string contains a reference to the *Leaguepeadia API* as this is where the information is retrieved from. When the URL string is built, it is passed as a parameter to the `NetworkUtil` component's `getJSONFromURLString()` method, which returns the JSON with information for the next step. See chapter 2.2.5 for details on how `NetworkUtil` gets this JSON. The Gson library is then used to parse the JSON into a `JSONObject` and the URL of the page containing the names of all the players' accounts is retrieved from this `JSONObject`.

This URL is then passed as a parameter to another `DataExtractor` component's method called `getDocumentFromURLString()` to retrieve an instance of *jsoup* library's `Document` with all the player's account names. See chapter 2.2.5 for details on how `NetworkUtil` gets the instance of `Document`.

Sometimes the *Leaguepedia API* does not contain the required information and when this happens, `getDocumentFromURLString()` is passed template with player's name as parameter instead and if page is not found, `getDocumentFromURLString()` returns specially designed exception which is used to inform the user that player's accounts were not found.

For each account name, an instance of `Account` is created via the `getAccounts()` method by passing an account name as parameter, which is then added to the list of accounts and returned by the `getAccounts()` method as requested. The `getAccount()` method works similar to the method described above, it first builds URL string, then calls `NetworkUtil` component's `getJSONFromURLString()` method and then uses Gson to parse the JSON to `JSONObject` and retrieve the requested data from that `JSONObject`. The URL string contains a reference to the *Riot Games API* as this is where the information is retrieved from.

When the `fetchAccountsToPlayer()` method has the list of current player accounts, it compares this list with the list of player accounts already persisted in the database. If it finds an account that is not already in the database, it adds that account. And if it finds an account that is already in the database but not in the list of

current accounts, it deletes that account from the database.

Fetching Matches to the Roster

This section describes the first of two methods for retrieving match data. The matches fetched in this method are competitive matches, i.e. they are matches from tournaments, so they are played by all players on the roster, which is why they are referenced in the Roster class property.

When the `DataExtractor` component is asked to retrieve match data for the instance of the `Roster` class via the public method `fetchMatchesToRoster()`, first all matches that are already in the `Roster` instance, because they were retrieved from the database earlier, are added to the list of matches which is a field of the `DataExtractor` class. These matches are added there to avoid adding duplicate matches to the roster.

The `fetchMatchesToRoster()` method builds a URL string from a predefined template, parameters and methods of the `TimeUtil` component, which allows many different time formats to be built and returned. See chapter 2.2.6 for more details on how `TimeUtil` builds these time formats. The URL string contains a reference to the *Leaguepedia API* as this is where the information is retrieved from. One of the parameters used in the URL string is a property of the `Roster` class, which stores the time of the last time matches were retrieved for this roster. This parameter is used to reduce the runtime of the application by reducing the number of URL requests and to avoid adding duplicate matches to the roster.

When the URL string is built, it is passed as a parameter to the `NetworkUtil` component's `getJSONFromURLString()` method, which returns the JSON with all the required information. See chapter 2.2.5 for details on how `NetworkUtil` gets this JSON. The JSON is then parsed into a `JSONObject` and an array of match ids is retrieved from the `JSONObject` using the Gson library.

For each match id, the `returnMatch()` method is first called with the match id passed as a parameter, which returns an instance of the `Match` class if it is found in already persisted matches or matches that have already been retrieved in this session.

If null is returned, the `getMatchFromLeaguepedia()` method is called with the match id passed as a parameter to create and return an instance of the `Match` class.

The `getMatchFromLeaguepedia()` method obtains match data in two parts from two different pages. The first part retrieved is an instance of the `Info` class and the second part is an instance of the `Timeline` class. These parts use different predefined template and same parameters to build URL string which is then passed as parameter to `getDocumentFromURLString()` method. This method returns a *jsoup* library class called `Document` containing an important instance of jsoup library's `Element` class containing all the required information. This information is extracted from the element and parsed into the `Info` and `Timeline` instances of the data model using *Gson* library. The instances of these classes are then used as parameters in the constructor of the `Match` class to create and return the requested `Match` instance.

The final action of the `fetchMatchesToRoster()` method is to update the property of the `Roster` instance with the last time this method was called.

Fetching Matches to the Accounts

This section describes the second of two methods for retrieving match data. The matches fetched in this method are *solo queue* matches, i.e. they are matches from the public ladder, i.e. they are normally only played by one player from the roster on his particular account, which is why they are referenced in the `Account` class property.

When the `DataExtractor` component is asked to retrieve match data for the instance of the `Account` class using the public method `fetchMatchesToAccount()`, it first adds any matches that are already in the `Account` instance, because they were retrieved from the database earlier, to the list of matches that is a property of the `DataExtractor` class. These matches are added there to avoid adding duplicate matches to the account.

The `fetchMatchesToAccount()` method builds a URL string from a predefined template, parameters and methods of the `TimeUtil` component, which allows many different time formats to be built and returned. See chapter 2.2.6 for more details on how `TimeUtil` builds these time formats. The URL string contains a reference to

the *Riot Games API* as this is where the information is retrieved from. One of the parameters used in the URL string is a property of the `Account` class that stores the time of the last time matches were retrieved for that account. This parameter is used to reduce the runtime of the application by reducing the number of URL requests and to avoid adding duplicate matches to the account.

When the URL string is constructed, it is passed as a parameter to the `NetworkUtil` component's `getJSONFromURLString()` method, which returns the JSON with all the required information. See chapter 2.2.5 for details on how `NetworkUtil` gets this JSON. The JSON is then parsed into a `JSONObject` and an array of match ids is retrieved from the `JSONObject` using the Gson library.

The *Riot Games API* has a limit of returning a maximum of 100 results per query. So if the number of match ids returned is 100, the `fetchMatchesToAccount()` method is called again recursively at the end of this method with the second parameter increased by 100. This second parameter defines from which point in the result set the 100 results are taken and returned.

For each match id, the `returnMatch()` method is first called with the match id passed as a parameter, which returns an instance of the `Match` class if it is found in already persisted matches or matches that have already been retrieved in this session. If null is returned, the `getMatchFromRiot()` method is called with the match id passed as a parameter, which creates and returns an instance of the match class.

The `getMatchFromRiot()` method retrieves match data in two parts from two different pages. The first part retrieved is an instance of the `Info` class and the second part is an instance of the `Timeline` class. These parts use different predefined template and same parameters to build URL string which is then passed as parameter to `getJSONFromURLString()` method which returns JSON containing required information. Gson library is then used to parse the JSON into the `Info` and `Timeline` instances of the data model. The instances of these classes are then used as parameters in the constructor of the `Match` class to create and return the requested `Match` instance.

The final action of the `fetchMatchesToAccount()` method is to update the property of the `Account` instance with the last time this method was called.

2.2.3 Database Management

The `DatabaseManager`, another component of the application, is responsible for all interactions with the database, such as persisting or selecting data using the *Hibernate* library. This is the first of the modules described here that is *modular*, so it can be used in any other Java developed application if its methods are needed. The component is represented by a static Java class. It is not technically static, but all methods and properties are static and the construct of the class is set to private access modifier to prevent any misuse of the class. This is the component that implements the "Managing Database" use case mentioned in the use case diagram. This use case includes two other use cases listed below, which are described in detail in the following chapters.

- Persisting Data to the Database
- Selecting Data from the Database

Persisting Data to the Database

This section describes the use case that occurs when the `DatabaseManager` component is asked to persist data to the database via the `insertObject()` public method. The only parameter passed to this method is an instance of a class that implements the `Insertable` interface, which implements all classes in the data model.

Firstly, the `insertObject()` method uses the existing session where all the data from the database was previously loaded to create an instance of the `Transaction` class, then the parameter of this method is passed as a parameter to the `merge()` method which is called on the instance of the `Session` class. The `merge()` method adds the object from the parameter to the session or just updates it if it was already in the session. Finally, the `commit()` method is called on the instance of the `Transaction` class, which completes the process of inserting the object into the database.

Selecting Data from the Database

This section describes the use case that occurs when the `DataExtractor` component asks for the instance of the `Organisation` class in the `getOrganisation()` method via the `getObject()` method. This method has two parameters, the first is the reference to the required class and the second is the id of this class used in the database.

The `getObject()` method starts by checking if there is already an instance of the `Session` class in the `DatabaseManager`, and if not it creates a new one. Then it uses the `get()` method on the instance of the `Session` class with both parameters passed to this method as parameters.

The session and the session factory must obviously be closed at the same point, which happens when the application is closed. For this purpose there is a method called `closeSessionFactory()` in the `DatabaseManager` component that closes the existing session and the session factory. This method is called as a window listener that is added to the `MainFrame` class and executed when the window is closed.

2.2.4 Generating Output

This use case is implemented by the `OutputMaker` component. This component is represented by a static Java class. A simplified process for implementing this use case is shown in the collaboration diagram in Figure 2.2.4. It is not technically static, but all methods and properties are static and the construct of the class is set to private access modifier to prevent any misuse of the class. It has only one public method called `makeHTMLOutput()` which uses the jsoup library to create a document with all the neccessities like title, icon or links to JS scripts and CSS styles. Then it creates a `<div>` element called container, which is appended to the body of the document instance and stored in a *class variable* so that the other methods can add their partial outputs to it.

The partial outputs in the following sections are just examples of what can be done, as there is a lot of data available, so there is a large amount of possible infographics, graphs or heat maps that can be made. This component of the application is designed

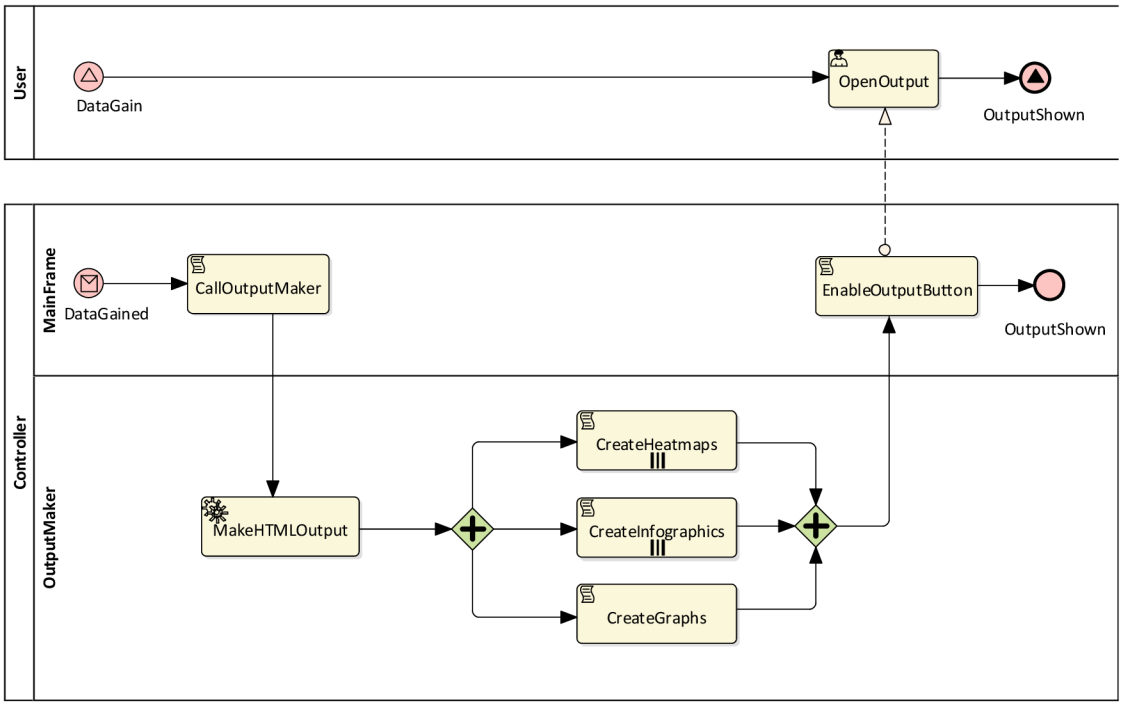

Figure 2.7: Collaboration Diagram of Simplified implemantaion of “Output Making” use case. Created in Enterprise Architect.

to be customised for each client based on their requirements and opinions about the game.

Then the `makeHTMLOutput()` method calls three private methods that implement the use cases listed below, which are included in the main use case of this section, which will be described in more detail in the following charters.

- Adding Infographics to the Document
- Adding Graphs to the Document
- Adding Heatmaps to the Document

Once all the above private methods have been called and completed, an instance of `BufferedWriter` is created to save the file to disk. It then returns the path to the file so that the `MainFrame` component can point to it.

Adding Infographics to the Document

The first involved use case described here is implemented by `addInfographics()`, which calls a few more private methods to produce these partial outputs:

- Competitive game record
- Competitive Champion Pool
- Solo Queue Champion Pool

The `addCompetitiveGameRecord()` method creates the first part of the output of this section by using the data model to find out if the organisation has won or lost any recent competitive matches, and adding a predefined representation of this record to the container element.

The next partial output is created by the `addCompetitiveChampPool()` method, which processes game by game from the instance of the roster in the method's parameter to find out the most played characters by each player in the organisation's competitive matches.

The last partial output of this section is created by `addSoloQChampPool()` method which procces game by game from the instance of player in method's parameter to find out most played characters by each player in his solo queue matches.

Adding Graphs to the Document

The second use case described here is implemented by the `addGraphs()` method, and it calls methods that produce partial outputs in the form of graphs made with the *C3 JS* library. The only partial output here is a graph of the team's average actual gold at a given point in a game, made by the `addGoldGraph()` method. More graphs will be made in future development, read more here 2.3.

The `addGoldGraph()` first creates an array in which it stores the cumulative amount of gold at a certain minute of the game. Then it processes game by game from the instance of the roster in the method's parameter to fill the array with values. The array is then used by the *C3* library to create a graph, which is added to the container element.

Adding Heatmaps to the Document

The last use case described here is implemented by the `addHeatmaps()` method, and it calls the `addHeatmap()` method twice, creating the requested partial outputs and adding them to the container element. The `addHeatmap()` method is called once with the team's *jungle* position and the second time with the team's *support* position, because movement on the map of these two positions has a big impact on the game.

The `addHeatmap()` method uses classes from the `java.awt` package to create images of the map of *Summoner's Rift* that represent where certain players spend most of their time in the game. First, it creates an instance of `BufferedImage` from the image of the empty map. Then it gets its width and height to create a two-dimensional array of intigers representing the data to create the image of the heatmap.

Once all the preparation is done, first the private method `addSoloQFrames()` is called and immediately after that the private method `addCompetitiveFrames()` is called. These two methods process one match at a time, the first processing the matches of the player, the second processing the matches of the roster. Each instance of `Match` has a reference to an instance of `Timeline`, which has a reference to a list of instances of the `Frame` class. The `Frame` class represents the state at a given time, with the same interval between each of them, that is, 60,000 miliseconds. This state at a given time contains, among other things, the position of each player.

So for each instance of `Frame` in the game, the position of a player that the heatmap is about is written to the array representing the heatmap by the `drawCircle()` method. The name of this method suggests that instead of just changing a particular pixel where the player was positioned, it affects a whole circle of pixels, making the resulting heatmap easier to read and understand.

The `drawCircle()` method takes three parameters. The first is an instance of the `ParticipantFrame` class, representing the state of a particular player at a particular time, which is referenced by an instance of the `Frame` class. The second parameter is the array representing the heatmap passed to this method as an argument. The last parameter is called value and is an `int` representing how much value a particular

`ParticipantFrame` has for a heatmap. Currently all *competitive* matches have a value of 5 and all *solo queue* matches have a value of 1. In future development every match will have the same value, read more here 2.3.

2.2.5 Retrieve data from an external data source

The `NetworkUtil` component is a modular class, so it can be reused in any other application where its methods are needed. It is responsible for all interactions with external data sources such as APIs or web pages. The component is represented by a static Java class. It is not technically static, but all methods and properties are static and the construct of the class is set to private access modifier to prevent any misuse of the class. This is the component that implements the Retrieve data from external data source use case mentioned in the use case diagram. This use case includes two other use cases that are not shown in the diagram, but are listed below and described in detail in the following chapters.

- Retrieving Document from a URL string
- Retrieving JSON from a URL string

Retrieving Document from a URL string

The first included use case is realised by `getDocumentFromURLString()` public method. It is uses when tha application needs some web page in processable format. The URL string passed to this method as parameter is firstly encoded by `encodeURLString()` method.

The `encodeURLString()` method is a private method of `NetworkUtil` class which splits the URL into base URL and query, then splits the query into individual query parameters and then on each of them `replaceAll()` method is applied with specially made regex expression to encode the URL to readable format by web servers. Finally, it reassembles all the parts of the URL and returns them.

Once the encoded URL string is returned, all of the following objects and methods are used from the *jsoup* library. Firstly the `connect()` method is used with the encoded

URL string passed as a parameter to create and return an instance of `Connection` which is immediately executed by the `execute()` method which returns an instance of `Connection.Response`. From this instance status code of the response is extracted and if it is equal to *200* response is parsed into instance of `Document` and returned. If the response status code is not *200*, an exception is thrown.

Retrieving JSON from a URL string

This second use case is handled by the `getJSONFromURLString()` public method. It is used when the application wants to retrieve data from a particular API. The URL string passed to this method as a parameter is first encoded using the `encodeURLString()` method. This method is described above.

Once the encoded URL string is returned, a combination of classes from `java.net` package and *Bucket4J* external library is used to retrieve the JSON smoothly. First `java.net` package classes and methods are used to create an instance of `HttpRequest` with the *Riot Games API* key in a header if the URL string contains a reference to the *Riot Games API*.

The `HttpRequest` instance is then sent in a do-while cycle as the *Riot Games API* has limitations. First layer to deal with the limitations is an instance of `Bucket` from *Bucket4J* library witch has some token capacity and is refilled constatly based on the limitations and every time before request is send it checks if there is any token in the bucket and then takes one and if there is none then it temporarily blocks the thread.

The `HttpRequest` instance is then sent and if a response code is *200* the thread leaves the do-while cycle and continues. If the response code is *429* or *503*, meaning the API is overloaded, then the second layer of dealing with API limitations is applied. By default, the thread is put to sleep for 10 seconds, but some APIs, such as the *Riot Games API*, will send back a response if they are overloaded, with a header indicating how long the requesting side should wait. If the API returns this instead of the default time, that time will obviously be applied. If any other response code is returned, expect those mentioned above, the exception will be thrown.

Finally, the body of the response is returned by the method.

2.2.6 Get time in a specific format

The last use case is implemented by the `TimeUtil` component, which is represented by a static Java class. It is not technically static, but all methods and properties are static and the construct of the class is set to private access modifier to prevent any misuse of the class. This component is modular, so it can be reused in any other application where methods of this class are needed. This component mainly uses the `java.time` package.

This class has three public methods. The first method creates an instance of `LocalDateTime`, which represents the current time in a particular timezone minus the offset defined by the method's parameter, and returns it in seconds from the epoch. The epoch is the default Java epoch which is 1 January 1970 00:00:00 [49].

The second method creates an instance of `LocalDateTime` which represents the current time in a particular timezone minus the offset defined in the first parameter of the method and the second parameter of the method, the instance of `DateTimeFormatter`, defines the format in which this instance of `LocalDateTime` will be returned.

The last method simply creates and returns an instance of `LocalDateTime` that represents the current time in a particular time zone.

2.3 Application's Configuration

There are two configuration files for configuring the application. The first is called `hibernate.cfg.xml` and it is, as the name suggests, a configuration file specifically for connecting the application to the database via *Hibernate*. There you can configure database properties such as URL, username and password. Also, the application is not dependent on the type of *relational database* once the *driver* and *dialect* of the database are configured here. The example of how to configure the properties is shown below. The rest of the file is the mapping of objects in the database, and changing

anything there could cause the application to stop working.

```
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost/test</property>
<property name="hibernate.connection.username">user</property>
<property name="hibernate.connection.password">password</property>
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.hbm2ddl.auto">update</property>
```

The second configuration file is called `config.properties` and you can specify API URLs, API keys, timezone or a property called `heatmap.radius` that changes the visualisation of the heatmaps.

Summary of results

In the theoretical part, it first introduces League of Legends and explains some important game mechanics and strategies so that the reader can understand the reasoning of decisions in application development. Then it introduces the reader to the growth of e-sports in a history to show the reader that it is becoming quite valuable and important part of the entertainment industry.

Then the development technologies and principles are introduced and explained. It starts with object-oriented programming, followed by the Java programming language and all the Java libraries used. The Java libraries used include Hibernate, Gson, Bucket4J, Swing and AWT. Finally, the web tools used in the output of the application are explained. These include HTML, JS and C3.

In the practical part, the application is derived from two different views. The first is the front-end to back-end view, which shows how users, external data sources, application components and databases interact with each other. The second is the Application Use Case view, which presents many use cases into which the application is divided and introduces and explains in more detail how each component of the application works and which use cases the component implements.

Conclusions and recommendations

This thesis successfully presents the development of a *Java-based* application in *IntelliJ IDEA* that assists *League of Legends* coaches and analysts in match preparation by providing customised output based on their unique requirements. The application demonstrates the potential of integrating data analysis and technology to optimise team performance in *e-sports*.

Looking ahead, there are several avenues for future development to further enhance the application's capabilities:

- Match evaluation: Future improvements could involve a more comprehensive approach to match evaluation, taking into account various attributes that provide deeper insights into match contexts. Attributes such as roster stability, match age and roster changes could be considered in the evaluation process to better inform coaching and analysis decisions. A refined evaluation system that incorporates these factors would enable users to better understand the factors influencing team performance and make more informed strategic decisions.

- Expanded output options: In addition to the existing outputs, more customised output options can be explored based on customer requests, extending the utility of the application in different coaching and analysis scenarios. Examples of potential outputs include player-specific performance metrics, champion synergy analysis and objective control statistics. By offering a broader range of outputs, the application can cater to a wider variety of user needs, making it a more versatile tool for coaches and analysts.

- AI-driven draft/win predictions: Implementing *Artificial Intelligence* (AI) to predict draft phases and win/loss outcomes represents a significant opportunity for the future development of the application. By incorporating machine learning algorithms and leveraging historical match data, the application could provide users with data-driven predictions of draft outcomes and match results. This would

add another layer of sophistication to the application, providing even more strategic value to users and enabling them to make more informed decisions during the draft process and throughout the match.

- Support for additional regions: Currently limited to the EMEA region, future development could incorporate data from other regions by exploring alternative data sources or conducting more comprehensive research on available resources for these regions. Expanding the application's regional coverage would require assessing the reliability and accuracy of alternative data sources as well as ensuring that the data is current and relevant. By expanding the application's regional support, users from different regions would benefit from the insights generated by the application, making it a more inclusive and valuable tool for the global LoL community.

- Downloadable output: Implementing a feature to download the output would enhance the usability of the application, allowing users to access and share the generated insights more easily. The downloadable output feature could include different formats such as *Portable Document Format* (PDF) to accommodate different user preferences and requirements. By providing convenient access to the generated data, coaches and analysts can collaborate more effectively and make data-driven decisions based on the insights provided by the application.

In conclusion, the development of this application marks an important step in the integration of data-driven strategies into *e-sports* coaching and analysis. The proposed future developments will further expand the application's capabilities, enabling it to better serve the evolving needs of LoL coaches and analysts worldwide. By continuing to innovate and adapt to the changing landscape of *e-sports*, this application has the potential to become an indispensable tool in the competitive gaming industry.

References

1. *Esports Part 1: What are Esports?* [Harvard International Review] [online]. 2020-04-24. [visited on 2023-04-02]. Available from: https://hir.harvard.edu/esports-part-1-what-are-esports/.

2. *What Is The Difference Between MOBA And Battle Royale Games — HP® Tech Takes* [online]. [visited on 2023-04-01]. Available from: https://www.hp.com/us-en/shop/tech-takes/moba-vs-battle-royale-games.

3. MORA-CANTALLOPS, Marçal; SICILIA, Miguel-Ángel. MOBA games: A literature review. *Entertainment Computing* [online]. 2018, vol. 26, pp. 128–138 [visited on 2023-04-03]. ISSN 1875-9521. Available from DOI: 10.1016/j.entcom.2018.02.005.

4. ROBSON, Matthew. *The 7 Most Popular eSports Leagues* [Game Rant] [online]. 2022-12-17. [visited on 2023-04-01]. Available from: https://gamerant.com/most-popular-esports-leagues/. Section: Lists.

5. *How to Play - League of Legends* [online]. [visited on 2023-04-01]. Available from: https://www.leagueoflegends.com/en-gb/how-to-play/.

6. *Summoner's Rift (League of Legends)* [League of Legends Wiki] [online]. 2023-04-02. [visited on 2023-04-02]. Available from: https://leagueoflegends.fandom.com/wiki/Summoner%27s_Rift_(League_of_Legends).

7. *Snowball* [League of Legends Wiki] [online]. 2013-05-30. [visited on 2023-04-13]. Available from: https://leagueoflegends.fandom.com/wiki/Snowball.

8. MOBALYTICS. *Understanding Scaling and How to Use it to Your Advantage in League of Legends* [Mobalytics] [online]. 2017-08-12. [visited on 2023-04-13]. Available from: https://mobalytics.gg/blog/understanding-scaling-use-advantage-league-legends/.

9. MOBALYTICS. *Wave Management Guide: Everything You Need to Know About Wave & Minion Control in League of Legends* [Mobalytics] [online]. 2021-02-08. [visited on 2023-04-14]. Available from: https://mobalytics.gg/blog/wave-management/.

10. *The Core Concepts of Each Lane in League of Legends* [Dignitas] [online]. 2022-05-11. [visited on 2023-04-11]. Available from: https://dignitas.gg/articles/the-core-concepts-of-each-lane-in-league-of-legends. Section: Guides.

11. *LoL Roles: How to Play Different Roles in League* [EarlyGame] [online]. [visited on 2023-04-11]. Available from: https://earlygame.com/lol/what-roles-exist.

12. *What Role Should I Play in League of Legends – Roles Explained* [online]. [visited on 2023-04-13]. Available from: https://blog.ggcircuit.com/what-role-should-i-play-in-league-of-legends.

13. *League of Legends Roles » How to choose the right role in LoL* [Esports.net] [online]. [visited on 2023-04-11]. Available from: https://www.esports.net/wiki/guides/league-of-legends-roles/.

14. BRIDGES, Stuart. *The different League of Legends roles explained* [Pinnacle] [online]. [visited on 2023-04-11]. Available from : http://www.pinnacle.com/fr/old-esports/betting-articles.

15. *Esports Part 4: Developer Control* [Harvard International Review] [online]. 2020-07-12. [visited on 2023-04-04]. Available from: https://hir.harvard.edu/esports-part-4-developer-control-the-implications-of-the-company-behind-riot-games/.

16. *eSports - Worldwide — Statista Market Forecast* [Statista] [online]. [visited on 2023-04-14]. Available from: https://www.statista.com/outlook/amo/esports/worldwide.

17. *Tier 2 Tournaments: Pre 2014* [Liquipedia Dota 2 Wiki] [online]. [visited on 2023-04-02]. Available from: https://liquipedia.net/dota2/Tier_2_Tournaments/Pre_2014.

18. *The International 2021* [Liquipedia Dota 2 Wiki] [online]. [visited on 2023-04-02]. Available from: https://liquipedia.net/dota2/The_International/2021.

19. PHOENIX. *The International 2022 FAQ: Everything you need to know* [Jaxon] [online]. 2022-09-25. [visited on 2023-04-02]. Available from: https://www.jaxon.gg/the-international-2022-faq-everything-you-need-to-know/.

20. *Season 1 World Championship* [Leaguepedia — League of Legends Esports Wiki] [online]. 2023-04-03. [visited on 2023-04-03]. Available from: https://lol.fandom.com/wiki/Season_1_World_Championship.

21. *League of Legends* World Championship. In: *Wikipedia* [online]. 2023 [visited on 2023-04-03]. Available from: https://en.wikipedia.org/w/index.php?title=League_of_Legends_World_Championship&oldid=1147795581. Page Version ID: 1147795581.

22. KONHÄUSNER, Peter; SEMMERAU, Sharon-Maria; GRUNERT, Marlon. Microtransactions in games – an analysis of a crowdfunding perspective. *Forum Scientiae Oeconomia* [online]. 2021, vol. 9, no. 4, pp. 31–58 [visited on 2023-04-04]. ISSN 2353-4435. Available from DOI: 10.23762/FSO_VOL9_NO4_2. Number: 4.

23. GOSLIN, Austen. *The 2018 League of Legends World Finals had nearly 100 million viewers* [The Rift Herald] [online]. 2018-12-11. [visited on 2023-04-03]. Available from: https://www.riftherald.com/2018/12/11/18136237/riot-2018-league-of-legends-world-finals-viewers-prize-pool.

24. *League of Legends Sets Record with 57 Million Viewers for Worlds* [EKGAMING] [online]. 2017-12-21. [visited on 2023-04-03]. Available from: https://ekgaming.com/2017/12/21/league-of-legends-sets-record-with-57-million-viewers-for-worlds/. Section: News.

25. *2019 World Championship Hits Record Viewership* [online]. 2019-12-17. [visited on 2023-04-04]. Available from: https://nexus.leagueoflegends.com/en-us/2019/12/2019-world-championship-hits-record-viewership/.

26. *Riot Games Esports Media Center - LoL Esports - Assets* [online]. [visited on 2023-04-03]. Available from: https://esports.riotgamesmedia.com/LoL-Esports.

27. *What is an API? - Application Programming Interfaces Explained - AWS* [Amazon Web Services, Inc.] [online]. [visited on 2023-04-01]. Available from: https://aws.amazon.com/what-is/api/.

28. *What is a REST API?* [online]. [visited on 2023-04-01]. Available from: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

29. *What is a REST API? — IBM* [online]. [visited on 2023-04-01]. Available from: https://www.ibm.com/topics/rest-apis.

30. *Riot Developer Portal* [online]. [visited on 2023-04-01]. Available from: https://developer.riotgames.com/docs/portal.

31. *Riot Developer Portal* [online]. [visited on 2023-04-01]. Available from: https://developer.riotgames.com/docs/lol.

32. *Leaguepedia API* [online]. [visited on 2023-04-01]. Available from: https://lol.fandom.com/wiki/Help:Leaguepedia_API.

33. SCHILDT, Herbert. *Java: The Complete Reference, Eleventh Edition*. 11th edition. New York: McGraw Hill, 2018. ISBN 978-1-260-44023-2.

34. GALLARDO, Raymond; HOMMEL, Scott; KANNAN, Sowmya; GORDON, Joni; ZAKHOUR, Sharon Biocca. *The Java Tutorial: A Short Course on the Basics*. 6th edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2014. ISBN 978-0-13-403408-9.

35. *Features of Java - Javatpoint* [online]. [visited on 2023-04-15]. Available from: https://www.javatpoint.com/features-of-java.

36. OTTINGER, Joseph B.; LINWOOD, Jeff; MINTER, Dave. *Beginning Hibernate: For Hibernate 5*. Apress, 2016. ISBN 978-1-4842-2319-2. Google-Books-ID: kOt6DQAAQBAJ.

37. *Your relational data. Objectively. - Hibernate ORM* [Hibernate] [online]. [visited on 2023-04-05]. Available from: https://hibernate.org/orm/.

38. *Documentation - 6.1 - Hibernate ORM* [Hibernate] [online]. [visited on 2023-04-05]. Available from: https://hibernate.org/orm/documentation/6.1/.

39. *Gson* [online]. Google, 2023 [visited on 2023-04-10]. Available from: https://github.com/google/gson/blob/b43ccee88927fa65d5e39a8ad4d0bebca7bf9994/UserGuide.md. original-date: 2015-03-19T18:21:20Z.

40. *Bucket4j 8.2.0 Reference* [online]. [visited on 2023-04-10]. Available from: https://bucket4j.com/8.2.0/toc.html.

41. *Java rate-limiting library based on token-bucket algorithm.* [online]. bucket4j, 2023 [visited on 2023-04-15]. Available from: https://github.com/bucket4j/bucket4j/blob/3f6da45c00134e10810c2a91e9bb624a7fac96b8/LICENSE.txt. original-date: 2014-10-12T14:19:18Z.

42. *Java Swing Tutorial - javatpoint* [www.javatpoint.com] [online]. [visited on 2023-04-10]. Available from: https://www.javatpoint.com/java-swing.

43. LOY, Marc; ECKSTEIN, Robert; WOOD, Dave; ELLIOTT, James; COLE, Brian. *Java Swing*. "O'Reilly Media, Inc.", 2002. ISBN 978-1-4493-3730-8. Google-Books-ID: fU1K9MxaWp0C.

44. ZUKOWSKI, John. *The Definitive Guide to Java Swing*. Apress, 2006. ISBN 978-1-4302-0033-8. Google-Books-ID: YPjZNlEgAMcC.

45. *Java Platform Standard Edition 8 Documentation* [online]. [visited on 2023-04-10]. Available from: https://docs.oracle.com/javase/8/docs/.

46. *What is HTML? Definition of Hypertext Markup Language - javatpoint* [www.javatpoint.com] [online]. [visited on 2023-04-15]. Available from: https://www.javatpoint.com/what-is-html.

47. *What is JavaScript? - Learn web development — MDN* [online]. 2023-03-05. [visited on 2023-04-15]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

48. *C3.js — D3-based reusable chart library* [online]. [visited on 2023-04-11]. Available from: https://c3js.org/.

49. *Instant (Java Platform SE 8)* [online]. [visited on 2023-04-21]. Available from: https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html.

List of Figures

List of Tables

List of Abbreviations

ADC *Attack Power Carry.* 7–10

AI *Artificial Intelligence.* 62

APAC *Asia-PACific.* 15, 72

APC *Ability Power Carry.* 7

API *Application Programming Interface.* , 2, 15–17, 25, 27, 29–31, 36, 46–49, 51, 57, 58, 60, 72

AWT *Abstract Windowing Toolkit.* 31, 32

CBLOL *Campeonato Brasileiro de LoL.* 15

CSS *Cascading Style Sheets.* 35, 53

DOTA2 *Defense of the Ancients 2.* 11

EMEA *Europe, the Middle East and Africa.* 15, 41, 63, 72

GUI *Graphical User Interface.* 31, 32, 38

HTML *HyperText Markup Language.* 35

HTTP *Hyper Text Transfer Protocol.* 16

IDE *Integrated Development Environment.* 1

JCA *Java EE Connector Architecture.* 24

JDBC *Java DataBase Connectivity.* 24

JFC *Java Foundation Classes.* 31

JNDI *Java Naming and Directory Interface.* 24

JS *JavaScript.* 35, 36, 53, 55

JSON *JavaScript Object Notation.* 16, 28, 29, 46–49, 51, 58

JSP *Jakarta Server Pages.* 24

JVM *Java Virtual Machine.* 23

KeSPA *Korean e-SPorts Association.* 10

LCK *LoL Champions Korea.* 15

LCS *League of Legends Championship Series.* 15

LEC *League of Legends European Championship*. 15

LJL *LoL Japan League*. 15

LLA *Liga Latinoamérica*. 15

LoL *League of Legends*. 1–3, 5, 7, 13, 14, 62, 63

LPL *LoL Pro League*. 15

MOBA *Multiplayer Online Battle Arena*. 2

MSI *Mid-Season Invitational*. 14

NBA *National Basketball Association*. 13

NFL *National Football League*. 14

NHL *National Hockey League*. 14

NPCs *Non-Player Characters*. 4

OOP *Object-Oriented Programming*. , 17, 22

ORM *Object-Relational Mapper*. 24

PCS *Pacific Championship Series*. 15

PDF *Portable Document Format*. 63

PLAF *Pluggable Look And Feel*. 32

PUUID *Player Universally Unique IDentifier*. 16

REST *Representational State Transfer*. 15, 16, 72

RMI *Remote Method Invocation*. 23

RTS *Real-Time Strategy*. 2

TCP/IP *Transmission Control Protocol/Internet Protocol*. 23

TI21 *The International 2021*. 11

URL *Uniform Resource Locator*. 23, 46–51, 57–60

VCS *Vietnam Championship Series*. 14, 15

XML *eXtensible Markup Language*. 16

XP *eXperience Points*. 6

A. Attachments

A.1 GitHub Repository

https://github.com/fatheus97/bpapp
Username: UHK-BLAHAFR1
Password: 4Aqb@CLTGWPMAFb

A.2 GitHub Release

https://github.com/fatheus97/bpapp/releases/tag/alpha
Username: UHK-BLAHAFR1
Password: 4Aqb@CLTGWPMAFb

A.3 Screenshots of the Application Flow

Name of file: `screenshots.pdf`

A.4 Example of HTML Output File

Name of file: `output.pdf`

A.5 Compressed Folder with the Application

Name of file: `application.zip`

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **František Bláha**

Studium: I2100929

Studijní program: B0688A140001 Informační management

Studijní obor: Informační management

Název bakalářské práce: **Desktopová aplikace na platformě Java pro analýzu dat v oblasti e-sportu**

Název bakalářské práce AJ: Desktop Java application for data analysis in e-sport

Cíl, metody, literatura, předpoklady:

Cílem práce je představení zvolených vývojářských technologií na platformě Java a jejich využití k vytvoření desktopové aplikace. Následující nástroje a knihovny budou prostudovány a porovnány s alternativami, aby mohly být v praktické části správně využity: Riot Games API, Amazon RDS, GSON, JDBC a knihovny integrované v balíčku Javy.

Praktická část práce se bude sestávat z aplikace, určené k tvorbě materiálů pro přípravu strategie na nadcházející soupeře e-sportového týmu. Aplikace bude vyvíjena ve spolupráci s e-sportovým týmem UHK a následně jim bude poskytnuta k využití.

[1] NOVAK, Andrew R, Kyle JM BENNETT, Matthew A PLUSS a Job FRANSEN. Performance analysis in esports: modelling performance at the 2018 League of Legends World Championship. *International Journal of Sports Science & Coaching* [online]. 2020, **15**(5–6), 809–817. ISSN 1747-9541. Dostupné z: doi:10.1177/1747954120932853

[2] NOVAK, Andrew, Kyle BENNETT, Matthew PLUSS a Job FRANSEN. *Performance analysis in esports: part 2 – modelling performance at the 2018 League of Legends World Championship* [online]. 2019. Dostupné z: doi:10.31236/osf.io/84fmy

[3] NOVAK, Andrew Roman, Kyle BENNETT, Matthew PLUSS a Job FRANSEN. *Performance analysis in esports: part 1 - the validity and reliability of match statistics and notational analysis in League of Legends* [online]. B.m.: SportRxiv. 9. říjen 2019 [vid. 2022-10-05]. Dostupné z: doi:10.31236/osf.io/sm3nj

[4] KARAKURT, Alkım. *Prediction of league of legends draft picks for competitive matches* [online]. B.m., 2021 [vid. 2022-10-05]. Thesis. Applied Data Science. Dostupné z: http://acikerisim.tedu.edu.tr/xmlui/handle/20.500.12485/875

[5] KICA, Artian, Andrew LA MANNA, Lindsay O'DONNELL a Tom PAOLILLO. *Analysis of Data Gathered from League of Legends and the Riot Games API* [online]. 15. březen 2016 [vid. 2022-10-05]. Dostupné z: https://web.wpi.edu/Pubs/E-project/Available/E-project-032516-161404/unrestricted/lolcrawler_Report.pdf

Zadávající pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: Mgr. Vojtěch Vorel, Ph.D.

Oponent: Ing. Karel Malý, Ph.D.

Datum zadání závěrečné práce: 26.1.2021