Loyalty Program For Urban Bicycle Promotion

António Maria de Sousa Nunes Marques Pinto

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors:  Prof. Paulo Jorge Pires Ferreira
              Prof. João Pedro Faria Mendonça Barreto

Examination Committee
Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor:  Prof. Paulo Jorge Pires Ferreira
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

June 2017
Dedicated to my family.
Acknowledgments

I would like to thank my supervisor, Professor Paulo Ferreira and Professor João Barreto for the support, guidance and feedback provided throughout this thesis. I also want to thank my parents, for their love, patience, and resilience. Not only during these last few months but also in the time spent at Instituto Superior Técnico. Finally, I want to acknowledge all the support provided by friends and colleagues. Thank you.
Resumo

Ciclismo como meio de locomoção alternativo em meios urbanos é cada vez mais atractivo. As vantagens ambientais e de saúde são um foco crescente da sociedade e as vantagens econômicas, em termos de pequena e grande escala, embora menos óbvias mas perfeitas para incentivar o uso de bicicletas.

Tendo isto em conta, um sistema em que lojistas se inscrevem e criam desafios e recompensas baseados à volta do uso da bicicleta, cumpriria o objectivo de incentivar o uso de bicicletas e aumentar a economia local.

O foco desta tese é o sistema que permite o lojista facilmente implementar desafios e recompensas personalizados para a sua loja e permite ciclistas visualizar os desafios, participar e completar os mesmos.

Palavras-chave: Ciclismo, Programa de fidelidade, Sistema de recompensas, Conteúdo gerado pelo utilizador
Abstract

Cycling as a means of locomotion alternative to the car is starting to become appealing in urban environments. The environmental and health advantages are a growing focus in society and the economic advantages, both locally and globally, while less obvious are perfect to promote bicycle use.

Taking this into account a system of challenges and rewards, where shop owners can register, created around the use of the bicycle as a means of transport, would fulfill the objective of encouraging the use of bicycle and increase local economy.

The focus of this thesis is the system that allows the shop owner to easily implement customized challenges and rewards for their stores and to allow cyclists to visualize, participate and complete challenges.

**Keywords:** Cycling, Loyalty programs, Reward systems, End-user content generation
# Contents

Acknowledgments ......................................................... v
Resumo ................................................................. vii
Abstract .............................................................. ix
List of Tables ............................................................ xiii
List of Figures ............................................................ xv
Nomenclature ............................................................ 1
Glossary ................................................................. 1

1 Introduction 1
   1.1 Motivation ..................................................... 1
   1.2 Cycle-to-Shop .................................................. 2
   1.3 Goal ............................................................ 3
   1.4 Existing Solutions ............................................. 4
   1.5 Thesis Outline ................................................ 4

2 Related Work 5
   2.1 Existing Solutions ............................................. 5
   2.2 End-user Programming ......................................... 6
   2.3 Programming-by-example ..................................... 7
      2.3.1 Systems .................................................. 7
      2.3.2 Advantages .............................................. 8
      2.3.3 Disadvantages ........................................... 8
   2.4 Natural language programming ................................ 9
      2.4.1 Systems .................................................. 9
      2.4.2 Advantages .............................................. 10
      2.4.3 Disadvantages ........................................... 11
   2.5 Visual Programming .......................................... 11
      2.5.1 Systems .................................................. 12
      2.5.2 Advantages .............................................. 15
      2.5.3 Disadvantages ........................................... 15
   2.6 Form Templates ................................................. 15
2.6.1 Systems .......................... 16
2.6.2 Advantages .................................. 17
2.6.3 Disadvantages .................................. 17
2.7 Custom Solution .......................... 18
2.7.1 Advantages .......................... 18
2.7.2 Disadvantages .................................. 18
2.8 Other Solutions .......................... 18
2.9 Solution Comparison and Conclusion .......................... 21

3 Architecture .......................... 23
3.0.1 Cycle-To-Shop Components .......................... 23
3.0.2 Use Cases .................................. 24
3.0.3 Shop Owner Use Cases .......................... 26
3.0.4 Shop Owner Application .......................... 29
3.0.5 Rewards .................................. 32
3.0.6 Interface .................................. 32
3.1 Tracking Server and Cyclist Application .......................... 34
3.1.1 Communication .................................. 34
3.1.2 Database .................................. 35

4 Implementation .......................... 37
4.1 Node.js .................................. 37
4.1.1 Choosing Node.js and alternatives .......................... 37
4.1.2 Shop Owner Login .................................. 39
4.1.3 Heuristic Approach for Interface Design .......................... 40
4.1.4 Database .................................. 41
4.2 Interaction with Cyclist application .......................... 42
4.3 Conclusion .................................. 43

5 Results .................................. 45
5.1 Requirement 1: Ease of Use .......................... 45
5.1.1 Experiment .................................. 45
5.1.2 Results .................................. 46
5.2 Requirement 2: Flexibility .......................... 48

6 Conclusions .................................. 51
6.1 Future Work .................................. 51

Bibliography .................................. 53
List of Tables

2.1 Comparison of possible approaches. ........................................ 22

5.1 Results of the first round of user testing. ............................. 47
5.2 Results of the second round of user testing. ......................... 47
5.3 Results of the third round of user testing. ............................ 48
List of Figures

3.1 Sequence of Cyclist Request Reward Code. ............................................ 25
3.2 Sequence of Cyclist Post Trajectory. ..................................................... 26
3.3 Sequence of Shop Owner Creating a Reward System. ............................... 27
3.4 Use Case Checkout Reward. ................................................................. 28
3.5 Collaboration diagram of Shop Owner Verify Code. ................................. 28
3.6 Main login Page. .................................................................................... 33
3.7 Main menu Page. ................................................................................... 33
3.8 Challenge Template selection list. ......................................................... 33
3.9 Challenge Template costumization menu. ............................................. 34
3.10 Reward Database. ............................................................................... 36
Chapter 1

Introduction

In recent years, bicycle use as a means of transport has become more popular, with around 200 million bicycles in the EU alone in 2000. While in some countries the use of bicycles as a main mean of transport is already happening, like Denmark and Netherlands, in others, steps are starting to being taken into account. Cities are trying to provide better cycling infrastructures; other steps being put into action are bicycle-centric tourism, e-bikes, and bike sharing.

This is due to bicycles being a healthy, cheap and environment-friendly means of transportation, which can be used for short routes with minimal effort. Not only the benefits but with urbanization, cities have high car congestion, especially during peak hours, creating a problem where people turned to cycling as the solution, since it is a faster and easier method of transportation for many along with public transportation.

These topics were addressed in the 2011 Transport White Paper, published by the European Commission, where its main goals are phasing out conventionally fuelled cars in urban areas and achieving $CO_2$ emission-free cities, along with other research showing that society is slowly but surely making a change towards clean transport methods.

1.1 Motivation

There are two main motivations for this project.

First is the environmental and social aspect, by promoting people to use a bicycle as an alternate means of transportation it will reduce the dependency of fossil fuels and emission of greenhouse gasses like $CO_2$ and increase peoples day to day fitness activities[? ? ]. Simply changing 5% of kilometers done by gas emitting vehicles to cycling would save about 22 million liters of fuel every year in New Zealand alone. Even when including the $CO_2$ emissions of food required by the increased cyclist dietary intake, the bicycle is still the lowest emitter of greenhouse gasses per passenger kilometer traveled, around ten times lower than a car and 5 times lower than a bus. The increase of fitness activities on the day to day life also increases happiness and reduces the probability of developing depression, this combined with the decrease of traffic and costs increases the productivity of people.
The second motivation is economical since unlike what most people think using the bicycle as a means of transportation actually has sizable economic benefits both in large scale and in small local economies. In large scale, using bicycles increases jobs, presently there are more than 650,000 jobs in the EU related to cycling, by doubling bicycle usage we could add more than 400,000 extra jobs. Not only the jobs but the health benefits amount decreases between 114 and 221 billion euros since mortality rate is reduced and overall health increases. Finally, road congestion in the EU costs nearly 100 billion euros every year.

More importantly for this project specifically are the local economy benefits, since this is where the investment is done and the promotion is made. Retailers do not have a notion and often underestimate a number of clients that go shopping by bicycle and overestimate a number of clients that use a car. We can see an example of this in Bristol, England, where a survey showed that retailers not only underestimated a number of customers that travel by bicycle to their shops and the distance they would travel but also overestimated by around 100% the share of customers that drive to their stores. At the moment, customers going shopping by bike accounts for 111 billion euros in EU; if this number doubles it would generate an increase of more than 27 billion euros in local retailers alone. When using bicycles people can easily shop on the way to their destination, and possibly become a repeat customer. This ease to stop and shop results in better customer loyalty, more money spent per month and increased local economy, in some cases, like in Seattle, it was shown that after developing bike lanes there was a 400% increase in sales volumes. Clients coming by bike spend more than those coming by car, be it during a certain time period or related to the parking space that has to be provided for them. Car drivers might spend more per visit, but they visit shops less often. Cyclists do their shopping locally and are more loyal customers.

1.2 Cycle-to-Shop

Cycle-to-Shop is a service, associated with the TRACE project, that aims to promote an increase in urban cycling. In order to achieve this, shop owners will be provided with the tools to create reward schemes focused on promoting cycling. The goal is to entice people with rewards from several shops and businesses, resulting in more cyclists. On the other hand shop owners would also benefit from an increased local economy and better customer loyalty.

Cycle-to-Shop works through the interaction of three elements: the cyclist mobile application, the reward server and the tracking server. In this thesis, we will focus specifically on the reward server.

The reward server interacts with the cyclist mobile application, by providing it with information like which shops have rewards, what rewards are they and what do the cyclists need to do to receive a reward. The cyclists are then incentivized to cycle more or visit the shop by bicycle more often as to fulfill the requirements to win a reward. Once a cyclist does complete a challenge, the reward server sends the cyclist a code as an authentication of his merit so that he can retrieve the reward from the shop owner. The reward server is to be hosted by a third party, such as a municipality, country or a private company. To be able to create reward schemes the shop owners need to first register to the Cycle-to-
Shop service. These reward schemes are composed of at least one challenge. A challenge is a task with requirements that a cyclist must complete receiving a reward. A challenge could be for example "visit my shop every day and get a discount" or "bike ten km before visiting my store and get a discount". Reward schemes, or loyalty programs, are usually used in marketing to get new customers and increase the loyalty of old customers. An example of a popular reward scheme in restaurants is giving a discount after a customer buys a certain amount of meals. The reward scheme would be defined by each shop and the rewards would encourage cyclists to visit stores and the stores get more clients.

1.3 Goal

The goal of this work is to design a system, called Cycle-to-Shop reward, that encourages the use of bicycles by using reward schemes.

While there are already some services that create reward schemes using a few different templates, they are too simple and limited. The Cycle-to-Shop reward server, unlike other existing solutions, aims to be generic and flexible to adapt to various communities and scenarios, like schools, museums, parks, and many others.

The challenge is in designing a system that allows non-programmers, like the shop owners, to define the reward schemes. There is also a balance to be struck with giving a simple system that everyone can use while not losing expressive power. This is important since the objective of this work is not a system to be used purely by shops but that can be adapted to any other structure or service provider.

The most important requirements are to provide a system that has a small learning time and have a low-cost implementation value. A user must pick it up and learn how to use it in a short time frame but it also must be cheap for small businesses to implement. This means the solution must shy away from complex programming languages. Two concepts to have in mind are: i) user acquisition, new customers gained and ii) user retention, users that keep using the service. These concepts are vital for the longevity of any product or service. By being as intuitive as possible, it will be easier for shop owners to start using cycle-to-shop, resulting in both high user acquisition and retention. On the other hand, while moving away from complex programming languages it must not lose their expressivity. This is to accommodate a vast array of possibilities and domains. The existing solutions for promoting cycling using rewards are very specific to a domain, for example, Ride2School[?] is specific to incentivizing students to ride bicycles. While there are some solutions that approach specific domains, there is no solution that provides a flexible approach that allows stores, schools, governments and others to create their own reward schemes.

Other non-functional requirements are scalability and security. Scalability is important since we want to encourage the biggest possible number of people to start using a bicycle and need a big number of registered stores to create an enticing platform with rewards for the cyclists. This means tens of thousands or hundreds of thousands of potential users daily, both shop owners and cyclists that have their own data.

A general idea of a number of cyclists can be taken from the ECF, in 2000 there were approximately
200 million bicycles in the EU, while only 10% of the bikes were used on any given day [? ]. In 2014, a total of 1.051 million new bikes have been sold in The Netherlands, 3.6 million in the UK, around 3 million in France and 4.1 million in Germany. All of these were an increase from the previous year by around 7%, with a possibility of exponential growth especially in Germany where a number of bicycles in use are approximately 72 million pieces.

A big number of cyclists also means that checking if users fulfilled a reward scheme requirements may be increasingly expensive. Security, on the other hand, is required since, in the case of big rewards, shop owners can not afford to be tricked by users pretending to have fulfilled the challenge requirements. This means that steps should be ensured that a cyclist cannot receive a reward without fulfilling the reward scheme requirements.

### 1.4 Existing Solutions

Currently, these systems are all focused in very specific situations and communities. For example, Ride2School[? ] is designed specifically to be used in schools since the teacher manages the counting of a few students. This method would not be easily applied to other communities with more users. So there is a gap to be filled by an application that is flexible enough that can have its reward schemes be tailor made to the audience while still being simple to use. This thesis is one of a three part system. Cycle-to-shop tries exactly to encourage people to bike while being relevant in any community. The three parts are, tracking the biker, storing the tracking data, and the one this thesis approaches, the definition of reward schemes by the user.

### 1.5 Thesis Outline

This document is organized as follows. In Chapter 2, it is described several methods of allowing end users to program and potentially create their own loyalty programs. Then, Chapter 3 presents the solution while explaining the use cases and overall design. Chapter 4 describes the implementation of the prototype. Section 5 details the evaluation of the system and Chapter 6 conclusions taken.
Chapter 2

Related Work

To create an easy-to-use system that allows non-programmers to create content we must delve into end-user programming\[1\]. From end-user programming, we can take away the difficulties the shop owners will face when creating their own reward schemes. The following sections further explore the current solutions in easy-to-use content creation.

2.1 Existing Solutions

Taking into account the goal is to design a system that encourages the use of bicycles by allowing shop owners to create a loyalty program that encourages cycling and its core requirements are ease of use, flexibility and low cost of implementation it is important to look at currently implemented solutions. Some examples are CykelScore in Denmark, Ride2School in Australia, among others. Ride2School HandsUp[2] is a system where each school simply counts how many students arrived at school on any given day and they enter the data through their Ride2School account. To incentivize this behavior, in each school term it is created a point ladder for ranking schools where the top school is rewarded. This method can be done since schools have a few students that can be counted by the teachers, while on stores this approach would be much harder. The reward scheme is also set up by the Ride2School organization so it can not be used in other domains like stores.

CykelScore[3], on the other hand, works by first registering the school, workspace on the project; by doing this, people can receive a chip to place on their bike. Besides the chip, the user must have an account on the website and an application on the smartphone. Therefore, an app is used to visualize the virtual checkpoints location and their range. While when using physical checkpoints it uses a chip on the bike that is detected by a previously placed electronic box. The points gained are used mostly as self-improvement tracking and comparison with colleagues/friends. Like Ride2School, the reward scheme is set up by an organization and in this case, it is always a lottery type, giving no control over the reward schemes to the shops or schools.

Other attempts at cycling reward programs are BikeMiles[4], yomp[5]; these also have the same problem as the previous two, they come with a specific reward scheme implemented, distance for points.
Meaning that shops, companies or schools cannot choose other types of activities that might be better for their cases.

Other non-cycling loyalty programs are shopkick [ ], flok [ ], Scoupy [ ], BellyBites [ ], Level Up! [ ], Meliuz [ ], Pirq [ ], BetterPoints [ ]. Even though these are not focused in promoting cycling, they still have the same problems as the previously mentioned programs: Low flexibility by being too specific, hard to use with complex interfaces and high implementation cost by forcing gadget or special readers.

Currently, these systems are all focused in very specific situations and communities. For example, Ride2School is designed specifically to be used in schools since the teacher manages the counting of a few students. This method will not be easily applied to other communities with more users. So there is a gap to be filled by an application that is flexible enough that can have its reward schemes be tailor made to the audience while still being simple to use. This thesis is one of a three part system. Cycle-to-shop tries exactly to encourage people to bike while being relevant in any community. The three parts are, tracking the biker, storing the tracking data, and the one this thesis approaches, the definition of reward schemes by the user.

2.2 End-user Programming

End-user programming is a field that allows end-users to program an application. In this case, we want to allow shop owners to create their own reward scheme logic. So, to be able to design a system that allows shop owners to create their own reward schemes, we must understand what are the end-users main difficulties and how to lessen them. What distinguishes a professional user from an end-user is mostly knowledge but also motivation, background, and knowledge. This means that end-users can be professionals in a technical field but not have in-depth programming knowledge, a researcher in an unrelated field or just a normal user. So end-user programming ends up being a practice where users usually have less technical knowledge [ ].

Surprisingly, even with less technical knowledge, end-users have the same difficulties as professional users, which are designing, testing and debugging their programs. These are especially important since there have been stories of severe consequences due to end-user errors. A famous example was a Texas oil firm lost millions of dollars due to an error in a spreadsheet formula [ ]. These difficulties together with trying to find a way to teach programming to beginner users, started a new wave of recent researches to find ways, which will be discussed later, to ease end-user programming. In the beginning, these researches have been focused on debugging and testing in spreadsheets, but recently there has been more with a focus on mobile, web development, and other fields. These new researches coupled with end-user programming being currently the most prevalent form of programming demonstrates the importance of this field.

One important aspect to take into account when creating these tools is the behavior of end-users. For this reason, there have been several studies and surveys done. End-users have a low percentage of reused code, especially code from other users. This may be the lack of tools, non-modular designs
making reuse harder, hard coded values or bad documentation [? ]. This is because end-users program without designing, in an erratic, iterative and opportunistic way. This behavior is shared with novice programmers, where it has been shown that the recurrent difficulties when learning how to program are large entities and abstract concepts. Another problem when learning to program is knowing when to apply what the students have learned. This is confirmed by both professors and students as a whole agreeing on how practical exercises are better for learning [? ].

Another problematic end-user behavior is not testing enough [? ]. Related with this, it was also shown that end-users have overconfidence in their programs, especially when using spreadsheets even though these are considered the most problematic [? ] [? ] . Because end-users have different motivations and backgrounds, tools that solve these problems need to be adapted from professional tools to end users with new approaches. An example of these new approaches is “What You See Is What You Test” methodology [? ] to help with debugging. WYSIWYG improved a number of end users that do debug and reduced the overconfidence in their own programs. The objective with these approaches is to improve end-user code quality without forcing the user actively to do it. So, when creating tools with these new approaches, there are three challenges to take into account: Understanding the end-users behaviors, take into account the specific domain, and finding out the appropriate level of expressiveness in the tools and language.

There are currently four approaches to making a flexible easy to use, end-user programmable system: programming-by-example, natural language programming, and templates. In the following pages these approaches will be explained, given examples of and dissected into pros and cons.

2.3 Programming-by-example

Programming-by-example (PBE) or by demonstration, is an approach that allows users to define a behavior without programming. This is achieved by the user inputting an example and the specific output for the example, instead of programming the behavior in general. For the system to capture this example, the user must first explicitly start it. The user then executes a sequence of actions that represent the desired behavior. The sequence of actions is used to turn an example input into an output. The sequence of actions, input and output are an example. From the input examples, it is then extrapolated the desired behavior and the result is a more general program that encompasses the original example intent. The generated program, given the example inputs, will output the correct results. This generated program is also then used, for other inputs other than the example inputs. PBE is an attractive approach for end-users since they naturally express themselves with examples.

2.3.1 Systems

Most of the uses in PBE have been seen in web data extraction and robot programming. WebCES [? ], CoScripter [? ], Internet Scrapbook [? ] allows the users to extract information from web pages through examples. They all work very similarly. On CoScripter a user, browsing in Firefox simply needs
a plugin. With the plugin the user can press a record button; Once pressed his actions are highlighted and recorded. Once finished CoScripter saves the actions as a script. These scripts are saved on a wiki where other users can change, comment and rate. However, the resulting scripts are usually very tightly connected to the specific website where the examples were performed.

Reform [? ], on the other hand, tries to allow the generated program to work on several similar websites besides the website used for the example. To do this it uses machine learning. This allows the user to point out any mistakes the program might do, and teach it for future cases. When tested with end-users, 86% of the tasks were successfully completed.

An interesting approach to PBE is seen in A Formal Framework for Combining Natural Instruction and Demonstration for End-User Programming [? ] where they used PBE in conjunction with natural programming. The user would first write a description of the program in natural language programming. Since natural languages are ambiguous, it would then create several hypotheses of possible results, which would be refined by consequent examples. The shop owner being able to start with a natural language description of their campaign and tweak it over time, although it sounds a simple approach at first, it actually results in other complications. For starters, it means that it will at first result in incorrect campaigns, this might result in loss of customers, and it is also not "one and done", requiring further tweaking from the shop owner.

### 2.3.2 Advantages

Explaining a concept by demonstration of an example is already inherent to people. This would allow the shop owners to easily input their promotion campaigns. In some of the systems presented, like web scraping, it is built on top of an already existing application. This could also be implemented in our system since shop keepers will be using a simple web browser, providing a feel of familiarity but now empowered with PBE so that they do not feel overpowered by novelty.

### 2.3.3 Disadvantages

A challenge present in PBE is the task of generalizing a generic program from a limited amount of examples that the system must perform. Because of this generalization, when using the generic program, it may incorrectly translate the behavior from the examples to a new input. This may result in campaigns that the shop owner did not intend. This mistake can result in loss of sales, which is not something a shop owner wants. The system must also be able to inform the user what is its current state in a meaningful way so that the user knows what to demonstrate next. Lastly, when the user later applies a new input to the produced generic program, the user should be able to rectify any mistakes. For biking related campaigns it is hard for the shop owner to provide examples with the location, distance traveled, and other factors using PBE. It should not be required for the shop owner to first bike a ten kilometers to be able to create a campaign where bikers need to bike ten kilometers, making this approach much less viable.
2.4 Natural language programming

Natural programming languages allow users to program using a language that the user uses every day, e.g. English. This approach allows end users to create rewards schemes with small to no additional knowledge. On the other hand, it has disambiguation problems, since words and phrases can have different interpretations depending on usage, context, culture and other factors. Even though the biggest challenge in programming is considered to be the interpretation of the problem and the designing of the solution, learning a new language is also a challenge for new programmers, especially because programming languages are very different from natural languages. While professional programmers might be used to programming languages, end-users also need to learn how to work with a strict syntax language, this is what natural language tries to fix.

Before going into more detail about natural languages, we must define its types. Natural programming language, might not be the ordinary English we are accustomed to; To avoid confusion *Foundations of the case for Natural-Language Programming* makes a distinction between two types of natural language, the active and the passive one. Passive natural language is a language that might look like natural language just enough so that anyone can instantly recognize it. On the other hand, unlike active a natural language, it has a rigid syntax so only people that are already experienced in it will be able to write applications using it. The active natural language, on the other hand, allows a user to write using a subset of a natural language for creating applications. So instead of the user only being able to use some pre-defined fixed amount of statements, in a specific way but a bundle of words that can use with the flexibility of natural languages and altered or extended easily. Passive natural language has very few pros; its readability is the main one, but it still maintains the rigidity of programming languages so we dismiss this type of natural languages. Another variation on natural programming languages is in using a keyword style; this takes advantage of the fact that most users are accustomed to using search engines, like Google, by inputting keywords. Just like when inputting keywords into a search engine, there is no strict syntax. From keywords like "add text button" the system would create a text button. When tested with users, who are not programmers, in *Translating Keyword Commands* the users were able to use the correct keywords for 84% of the tasks, and their first attempts had 72% success rate. The users were all experienced web users that used the web every day and used typing programs every day.

2.4.1 Systems

Some examples of environments that use a natural language for programming are PiE, Metafor, AliceNLP, among others.

**PiE** tries to lower the learning barrier of programming for novice programmers and non-programmers. It is built on top of LOGO, a graphics programming language designed for teaching purposes. PiE receives natural language as input and outputs a LOGO program. When tested, users were shown an image and a task to try and replicate it, both with and without the natural language, using LOGO. With natural language, it saved 43.8%, 21.8% and 22.4% of the time, respectively, for non-programmers, novices, and experienced programmers to draw the same graph.
Metafor [?] explored the idea of using a subset of English and let the user describe and write a story that is later turned into prototype code. This is done in a specialized interface that shows both the natural language story and the resulting code. The objective is not to replace normal programming with natural language but to assist new programmers in developing programming intuition. Another use for higher skilled programmers could be to brainstorm solutions to a given problem. When tested, non-programmers estimated 22% reduced time. It uses ambiguity to change representation or other implementation details that in natural language are not expressed directly. These changes are made dynamically every time the user adds more information about the ambiguous data. This takes some of the trouble of refactoring from the programmer.

In AliceNLP [?] the authors made a system that translates plain English into code ready to be executed using natural language programming technologies. AliceNLP is to be used as a programming tool for Alice, a 3d animation tool that is used in introductory programming classes. The objective is to allow the user to extend the basic functions provided by Alice with AliceNLP. During the user testing, users were provided with animations and had to describe them in natural language to try and reproduce them.

In *End User Empowerment in Human Centered Pervasive Computing* [?], they try to create an adaptive pervasive smart room system. This system is reactive and allows end-users to modify and extend the behavior of the system. This is done by combining natural language with programming by example. The tasks and the triggers are created by the user independently, so the user could choose for trigger "get home" to start a specific task like "turn on the lights" or choose any other trigger, making it modular in that way.

NL-play-in [?] is an interactive natural language interface to help user program, as a tool to improve a visual programming interface. The approach take to fix ambiguities was simple, the interface simply prompts the user to resolve any ambiguity.

Natural language programming can also be used in tools as aids for programming; A good example of this was made in *More Natural End-User Software Engineering* [?], where it was created a debugging tool called Whyline for java. It worked by allowing the user to ask "Why" and "Why not" questions; This made debugging a fun activity for non-programmers, increased productivity by 40% in controlled tests and when debugging with natural language debugger the novice programmers were faster while expert programmers were also able to debug twice as fast.

Another tool that uses natural language is Nlyze; It tries to help users by making a natural language interface to help spreadsheet programming in Excel. The motivation for this is that spreadsheet programs like Excel have several features that end users struggle to find and learn when to use. When testing, Nlyze produced the correct result from the input natural language code 94% of the times.

### 2.4.2 Advantages

When developing a program, the developer goes through several stages, first, it constructs a mental problem and solution, the second stage involves creating a version of that solution that the compiler can
understand, and finally the third is writing the program in the chosen language. If users could write in natural language, the second stage would be minimized if not completely removed since the constructed mental solution and the version of that solution made for a compiler would both be in natural language. The third stage would also be simplified since it is written in natural language, so the learning process is easier. So, it will allow the shop owner to easily write and create a campaign from his original campaign idea since natural language programming allows easier mapping from thought to programming.

Related to its readability, is the fact that Natural languages require less if no comments since it is self-documenting. Thanks to being self-documenting and in natural language, it makes any program instantly readable, allowing shop owners to easily read, understand and be able to edit any campaign.

2.4.3 Disadvantages

A challenge present in natural language is the difficulty in creating a domain simply from natural language. That is why several of the previous systems use UML [?] to specify what is an object, a list or another data structure. We can not ensure UML knowledge from shop owners, however, the domain could be a static one that the shop owner does not need to deal with. Even with pre-made UML domains, it would bind certain names and expressions to specific concepts that are meaningless to people that have no knowledge in programming, like classes, structures, and objects. If somehow those concepts were abstracted correctly to not matter to shop owners, the names, and expressions bound to them would be specific ones that shop owners would have to educate themselves on, by reading a manual and spending time learning about them. Since the focus is to make a simple solution to reward system creation, forcing a user to read a manual is counter-productive, especially because empirical research has largely found that users do not read the documentation.

Another similar problem is the vagueness that comes inherently with natural language flexibility since natural languages allow the user to be vague and not explicitly reference everything. While in theory, this is not the language fault, users tend to forget to specify facts needed, that more formal languages force to. For example, a shop owner when creating a campaign might specify its requirements and reward but forget to define the campaign ending date, this is because a natural language does not have a static specific syntax.

To fix these two problems it starts to become less like an active natural language and more like a passive natural language or a template form since there is a need for both a static domain and a strict syntax.

2.5 Visual Programming

A visual programming language is a language where each primitive element is represented with a visual element instead of a textual one. It is also a very high level of abstraction language since each visual element heavily abstracts from its logic. These visual elements are then connected to create a program. There are three types of visual languages: the flowcharts, the non-flowcharts, and the most popular,
the spreadsheets. Flowcharts are diagrams that are composed of boxes, of several types, that usually represent functions and arrows that display the logic flow of the program. Non-Flowcharts are diagrams that are simply composed of boxes, of various types, that connect with each other. Without arrows, the connections between boxes could be implemented to force the correct syntax by having boxes only connect if the following box has a matching symbol to the previous box. Spreadsheets are tabular forms that allow the user to specify the column and row meaning and have the cell content display its value. This is a field that does not have many empirical evidences, and most of the theory of its advantages come from the popular belief that visual elements are simply more user-friendly than text or its popularity. Between the few pieces of research that do exist, some conflict among them, indicating that the advantages of visual languages are not very clear cut. Fortunately, it is a field that is being researched more and more.

Visual programming is currently used massively in two fields. In teaching, mostly to teach programming to children and novices. In companies for the budget, cash flow and other types of calculations using spreadsheets. Some examples of teaching with visual languages can be seen in Scratch and Tynker, two projects that are primarily used by youth, ages 8-16 year to program apps and games, using a non-flowchart like a scripting language. Scratch has even been also implemented as a beginner computer science course at Norfolk State University, for at-risk students [? ]. To evaluate the results it was compared the grades and student retention between control and target groups. The control groups are simply students that had directly CS1 programming classes. On the other hand, the target group was at-risk students that had taken one semester of CS0 with Scratch. When comparing these, the results show that between two years, the target group shown better overall grades. On retention, around 60% of students that were exposed to Scratch kept going to the course while only 30% of the control group kept going. There were also surveys answered by the target group, where the students wrote that not only they enjoyed CS0 classes but it also provided them with better mindsets and a solid foundation for programming. We can then conclude that visual languages, specifically Scratch, have been applied with success in teaching classes.

2.5.1 Systems

Very similar to Scratch [? ] and Tynker [? ] there is Blockly [? ], a recent open source solution by Google. Non-flowchart visual languages are usually implemented as blocks that connect with each other. Each block has an underlying logic that directly maps to some text code. For example, an "if" block implements an "if" logic. Each block also has fields where the user can either fill in values or link other blocks. One interesting feature that non-flowchart languages like Blockly provide to end-users is the syntax error prevention. It is done thanks to the way blocks connect to each other. Just like jigsaw puzzles, each block piece has a small input shape and output shape. This means that a block cannot be connected to all other blocks, only the ones that make sense syntactically.

Two other interesting features provided by Blockly are both the creation of libraries and extension of the language. A user can save a complex combination of blocks as a single function and then use
it on other projects. Blockly allows the extension of the language by allowing the creation of custom blocks with their own custom logic. This extension is also done in Blockly, in what is called a Block Factory. Since the creation of new blocks is also made with blocks, it means that end-users do not need to request the help of a programmer when in need of extending the language. When the user is creating the new custom block, a code in JavaScript, Python, among others, is generated alongside the user visual language. The user then simply needs to copy and paste this code from the Block Factory to his program to add this new block specification. Due to the size of each block, creating large programs rapidly becomes cumbersome, and there are not many solutions to this. Since most programs are short, there are many limitations; For example, in Blockly there is no variable scope, all variables are global. There is also no easy way to handle external libraries currently.

MARBLS [?] is an end-user programming environment made to be used in Hospitals. Nowadays hospitals have systems that are aimed at helping nurses monitoring patients, alerting when something happens. These systems usually have rules input as condition/actions sequences. Typically these are defined in Cerner CLL [?] which is a language similar to SQL. Since Cerner CLL is a language that is not accessible to end-users, MARBLS goes for a visual language approach. MARBLS uses a non-flowchart language, very similar to Scratch and Blockly. Just like Scratch, it works as a jigsaw puzzle, where several blocks connect with the previous blocks establishing a sequence relation. Using colors, these blocks are divided into body blocks, operator blocks, function blocks and data blocks. MARBLS uses the same visual approach, that Scratch and Blockly have, to force an instinctive reaction to the user on what blocks can be followed by other blocks. This pre-emptively forces syntactically valid code, without displaying errors. Unlike other visual programming environments, it combines two work areas, that are two-way synchronized. Visual rule workspace, where the user can drag the blocks and define the logic of the rules. Visual query explorer, where the user can see data in a chart, corresponding to a variable. By selecting a value in this chart, the user can create conditions on the rule workspace with its value. Unfortunately, MARBLS is still in development, so the language is incomplete. It was also not tested in depth or empirically, taking only subjective comments from users into account.

Another well received visual language is G, running on LabView, a tool used for data acquisition. G is a flowchart language mostly used by electronic engineers. Unlike Blockly, G being a flowchart language has inherently improved debugging. This is because, with flowchart languages, a user can easily see the flow of the process and determine where it went wrong. On the other hand, being a flowchart language takes away the syntax error prevention that block languages have. The basic visual elements, of the G language, are small boxes, that represent functions, values or operators, and lines that connect boxes. Besides small boxes, LabView also provides a bigger box that can be used to add looping behavior. The wires carry an output value from a block to another block as an input. With its popularity over ten years, LabView grew a relatively well-sized community of scientists and engineers. In the study, 31 students of a science and engineering course worked with LabView. The objective was to find out if visual languages indeed provided better usability and readability when compared to textual languages. The users were provided challenges and they had to complete them both in LabViews visual language, G, and in a text language designed to be equivalent. The variables tested were accuracy and time. The results
showed that when using the visual language on parallelism and debugging challenges, the students had a clear advantage over text. This advantage was clear both in accuracy and in time. Another study involved 277 self-elected users. The objective was to find if the visual language in LabView was the feature that indeed made it easier to use. For this, the users were given a questionnaire with some open-format questions asking about Labview and how visual programming affected their mindset. They were asked which feature of LabView was the most advantageous when designing programs. Many of the responses showed a very high rating on visual feature versus other features. Obviously, it should not be assumed that respondents can judge their own cognitive processes accurately. However, the answers suggest that visual programming is a huge bonus for end-user programming.

GREAT is a business engine that uses visual language. It resulted from the lack of a good complete visual rule engine, noticed after looking into other popular rule engines like Drools, Visual Rules WebModeler(VRWM) and Microsoft Business Rule Composer. From these, the only solution that provided good visual capabilities was VRWM.

GREAT tried to improve on the visual rule engines by having automatic version control of the rules, the ability to create templates from the rules, and a user profile that works as a hub for both templates and versioning. As a visual programming language GREAT provides four base nodes: start, if, action, and end. Both if and action nodes have fields where the user must write the condition and resulting action in another business language; The user can then export the rules as a text programming language to feed into whatever program it is using. Besides visual rule editing, GREAT can also save rules as templates with several levels of granularity.

In a more generic research trying to and find out if visual programming allowed better program comprehension than textual programming languages or not. Two groups were tested: C programmers, and spreadsheet programmers. The result was that spreadsheet programmers had a better mental representation of both control and data flow, while C programmers had a better control flow than they had for data flow. When testing the reliability, from the inputted natural language they could create the correct output around 90% of the time, showing a high reliability. Other important studies were done by Saariluoma and Sajaniemi. These show that while using spreadsheets, users showed to go through a different mental process to acquire information. A possible reason for this could be the very structured layout that spreadsheets have.

The spreadsheet paradigm is considered by many to be the most used, by engineers to normal end-users. In fact, spreadsheet users are often end-users. A simple example of a popular spreadsheet program is Microsoft Office Excel. Spreadsheets are usually displayed in grids, composed by cells. These cells can have values, references to other cells or formulas. Spreadsheets are not only already well established but are very easy and quick to setup. By simply adding some values to the cells and later using a built-in sum operator, the user started and finished his program using common concepts. This allows users to start using them right away with minimal training.
2.5.2 Advantages

The concept behind visual languages is to, similarly to natural language programming, keep the end-user away from a strict syntax language. Unlike natural language, visual languages achieve this by going to a higher level of abstraction. This is done in most natural languages by, for example, avoiding variable initializations. Another advantage of visual languages is its expressivity. Since each of its visual elements could have several meanings through different variations of color, shape, icon the shop owner with a short introduction can easily distinguish these from each other.

2.5.3 Disadvantages

Deutsch Limit is a principle about one of the main limitations of visual languages. It is about the information density limitation that is inherent with visual languages. Unlike textual languages that can hold several thousand words on the screen, visual languages are far more limited with the ability of only display dozens of icons. With flowchart languages, high complexity and large scope problems usually create a high amount of clutter due to the wires crossing each other. This would not be the biggest problem since campaigns are supposed to be simple, but if the shop owner creates a more complex campaign it would result in a hard to read program due to clutter and low quantity of icons displayed.

One of the main problems of spreadsheets systems is its lack of visibility. Formulas spread between several cells and pages. Not only spread but also, usually, are hidden under the interface. These two combined with a lack of global vision makes debugging harder for end users; Resulting in the next problem, overconfidence in the created program. So, not only spreadsheets lead to errors and bad decisions but also make them harder to debug. In the case of shop campaigns, it results in angry customers and loss of sales, by creating an unrealistically easy or hard campaign. These errors are usually associated with the users being taught how to use the tools to use the spreadsheet but not how to correctly design programs in it.

2.6 Form Templates

Form Templates, like the name, indicates is a document with a strict layout with several boxes where the end-user can input several types of values. While the layout can change, it is never the duty of the end-user to change it, this is due to the fact that the type of each input box actually hides a logic that is hidden from the end-user, only known by the developer. This results in Form Templates being an approach where the developers can design all the implementation and keep it as a black box to the end-user, while still allowing customization of certain values, also picked by the developer. While it does reduce the flexibility of the end-user in terms of creating new logic, it significantly lowers the entry barrier by removing the logic creation and only opening to the user some values.

Even though the above solutions, are easy to learn when the user is willing, they are not instantly learned. When target user tests were made, the users expressed that they would rather have a simple approach that does not need a big time and work engagement. This requirement tells us that while the
previous approaches are more flexible, they are too involved, requiring not only learning how it works.

The Form Templates approach means that there is no flexibility, but it allows the user to not care about the logic since it is already done. By researching the most used shop challenges we can try and hit the needs for most shopkeepers without forcing them to learn visual programming or other complex approaches. These templates still have some customization in that they are form templates. The logic implementation is hidden from the user but still allows the shopkeeper to input values into several boxes that feed into the logic of the template.

To reduce the necessity of the end-user having control over the campaign logic there needs to be a good array of form templates already. For that, it is needed to implement the currently most popular loyalty programs.

2.6.1 Systems

when looking into customer loyalty services they all show the same or similar approaches, some kind of Form Template to allow the shop owner to create a campaign, a method to verify that the customer fulfilled the campaign requirements and a reward.

BetterPoints [?] has a similar objective to the one presented in this paper. It tries to encourage people to bike with challenges and rewards. Shops and other entities can register to the BetterPoints service, by registering they can set up their own challenges and rewards. The approach taken for the setup of their own challenges is a single Form Template page with several input boxes and checkboxes. In this page a shopkeeper can select the type of challenge, how much points does it give, and many other options, resulting in more than forty checkboxes. Even though it has the right idea, this makes the page cluttered and some options get mixed up between others. Another problem is the point system. Whenever a user completes a challenge he is rewarded with points. These points can then be traded off at any shop for a coupon. This results in behaviors like, a biker never visiting a shop or even passing close to it on his bike, and then after getting a coupon, using points from challenges that do not involve the shop in question, he can use this coupon on the shop. While the biker might be happy in the end, the shop owner did not get a customer that visits often but a customer that got a cheap deal and might not come back. To verify that the requirements of the campaign were fulfilled the user can be asked to scan QR codes on the establishments using his smartphone if the shop owner decided to when creating the campaign. This means that not only the logic behind the requirements of the campaign can be tweaked but also the verification method.

flok [? ], a customer loyalty program, uses a more minimalistic approach. It allows the shop owner to select between six campaigns or create a custom one. To complete the requirements for a campaign the user needs to scan a QR code present on the shop. This limits the types of campaigns. The user can only get a reward after completing several campaigns. These are tracked through a virtual punch card on a mobile application. Once it is complete, the user shows the smartphone to the shopkeeper.

BellyBites [?] has a single campaign that counts visits, that means that a shop owner that uses this loyalty program can only give rewards to clients that have a certain number of visits, that is the only
requirement. This is a Form Template gone too far down the side of strict logic and not allowing the end-user to customize the campaigns, resulting in no flexibility at all but the easiest to set up. BellyBites uses points specific to each shop, this is advantageous comparing to universal points since shopkeepers will not feel cheated by users that never interacted with their shops but still have several points from interacting with other shops. To verify the user visited a store it provides each new client with a QR code. It is the duty of the shopkeeper to have a specific tablet like device that can scan the users QR code and log his visit. After a number of visits, the user gets a reward. Both the number of visits and the reward are defined by the shopkeeper. Forcing the shop owner to have a device capable of scanning these codes and send it to the BellyBites service reduces the range of shop owners that can use this service.

LevelUp! [? ], like flok, also takes the minimalistic approach. It provides four types of campaigns: first visit, money spent, birthday and re-engagement. These basically only allow you to customize the reward the client gets. It means that the shopkeepers have no leeway in creating a campaign that fits their needs and the small range of campaigns further limits that.

### 2.6.2 Advantages

Form Templates means that the shop owner does not need to learn a programming language to create their own campaigns. Even if the shop owner is limited to the available templates it is a much better compromise. Not only that, but a shop owner does not want to lose a lot of time developing these loyalty programs. Instead, they want simple and fast creation methods that still yield results, and Form Templates fulfilled these concerns.

### 2.6.3 Disadvantages

By adding new templates for different challenges we can reduce the need of users having to create new challenges. On the other hand, what templates do not provide is the ability to mix and match features between challenges, due to their strict layout. Making templates for each combination possible not only would be work intensive but it would also increase the number of options making the choice harder for new users. Another problem with Form Templates is that they aim to support general needs, since due their rigid layout they can never fulfil the needs of all the shop owners, and it becomes a balance between only having the most popular options, and leaving other possibly important options to some shop owners, or having too many options, making the form visually filled with clutter.

A possibility is to create a separate screen for advanced users that want to have elements of some templates mixed. This while more complex than simple rigid templates is still simpler than visual programming.
2.7 Custom Solution

A custom solution is where each shop owner would contact and hire a developer to create a solution specifically crafted to its own requirements and clients needs. So in reality, this solution is unlike the others since the end-user, in this case, the shop owner has no hand in the logic behind the challenges.

This is a trade-off between time and money. By hiring a professional the shop owner spends less time implementing the reward system or learning how to implement it and instead simply pays the professional to create it.

2.7.1 Advantages

A custom website done by a professional, assuming the client requirements are correct, will provide great results since it will be specially crafted towards clients needs. A custom solution could very well be easy to use, providing an easy interface for the shop owner while still being a powerful tool.

2.7.2 Disadvantages

To start with, the shop owner has very little knowledge about software design and requirements, so when hiring a web developer the client would have to spend time and money to understand what kind of challenges and content, in general, they would need to be available on the website. Then, depending on the complexity, the development itself will take time that could span from weeks to months. Another variable that scales with complexity is cost, this can range from a few hundred to several thousand. It would also require a host of some kind, a shop owner with no technical knowledge would not know where to start and it would also add to the cost.

2.8 Other Solutions

Reward schemes are usually implemented as business rules, composed by conditions and actions. Business rules are strict and well-formed rules that are used to guide decisions in day-to-day business activity. Business rules provide both the criteria for making decisions and the result. These have also started to be implemented in business rules engines. Business rules engines are software systems that run and manage several business rules. The objective of a business engine is to have the rules defined outside of the main application code. Since the rules are defined in business engines it allows fast editing to the rules without changing code. Some examples of these are Visual Rules WebModeler(VRWM) and GREAT [? ]. These tools allow the end user to implement business rules using simple graphical elements. Both can be used to make simple rules like conditionals while VRWM can even create decision-tables. On the other hand, GREAT allows business rules templates to be created and shared between the users.

Currently, there is no clear standard method to define a business rule; some usual implementations are natural language or structured natural language, markup languages like XML [? ] or RuleML [? ].
modelling languages like UML and programming languages. We can observe some examples of these implementations in current systems like BUSINESS, Drools and others.

BUSINESS is a domain specific language for application development made specifically for end-users. It uses an office model to try and bridge the thought process between an office worker and program development. For this, it tries to make each instruction represent a physical action that could happen in an office. Data structures also map to entities in an office, like cabinets, resulting in instructions that an office worker might be used to, like pulling a folder from a file cabinet instead of loading a text file. It also tries to be a very similar language to English, making it a passive natural language. Even though it uses a language based on English and is combined with its office model, it is still a passive natural language. The fact that it is a language with a strict structure, means that even though end-users might be able to read it adequately, it still requires the end-user to be taught how to use this language. It also uses flow jumps similar to go-to which make the flow of the program harder to follow. While a company that used to hire programmers specifically for rule specification might find enough to teach office workers to use it, it is not easily picked up by an end-user, like a shop owner and the time and investment required to learn a language is too much. Its office model while helpful, does not translate well to shop owners. While a similar language could be created for shop owners and campaign creation, it still has faults due to its strict syntax just like a normal programming language.

Drools allows the usage of XML, a native rule language and decision tables in spreadsheets. Using XML is definitely not a good approach for end-users. While to experienced programmers it might seem like a reasonable approach, for end-users it is not. This is due to XML being a formal language with a very strict structure, which is needed to be known by heart by the user, making it hard to be used. Even worse is the fact that XML is not readily human readable. Drools support rule management with spreadsheets as decision tables. Drools advice is to use decision tables when you only want to create relationships between conditions and actions. This is because decision tables do not expose the rules directly, unlike templates. Besides just defining the business rules, Drools also allows defining the processes. This is done through Drools Flow. Drools Flow gives the user more control on how the rules are executed with a flow chart. A user, using Flow, can define the order of execution of rules or what rules to trigger based on certain events.

Jess is a rule engine that accepts rules in two formats. The first format is XML which is not a good approach for end-users. The second one is its own language, similar to Lisp. This language is far from approachable to end-users due to its syntax. To have the best efficiency, it needs to take into account the java application since it can load objects directly from java using what it calls "shadow facts".

OpenRules is an open source solution for business managers to employ their own rules. Open-Rules also allows several different approaches but all using spreadsheet programming, specifically on Excel. It allows the creation of tables on a spreadsheet with several functionalities, like decision rule tables, data tables, tables with Java code snippets that describe methods, among others. While some might be hard to implement by an end-user, data and decision tables are easily maintained. The easiest for end-users would be decision tables since they are recognizable by business users, these can easily be used to specify rules and their logic. Decision tables are especially useful when rules have multiple
conditions. This is because each condition is a row, and each action is a column. By selecting rows on a column, the user is selecting the conditions that are needed to trigger the action. It also allows an easy addition to new conditions to an action, by just adding a new row and selecting it on the correct column. The downfall when using decision tables is that the variables and their values for the conditions need to be defined somewhere and this is usually done by a programmer. OpenRules also allows non-technical users to design and implement a web form page without knowing how to use HTML, JavaScript. This is also done in a spreadsheet, where the user simply states a variable of a data type, and it auto generates fields that receive user input and save it in the chosen variable. The user can also define the visual attributes of the web form, like border and style.

Usually, a business expert creates all the rules in natural language and then manually translates them into a formal language. This translation takes time and is difficult. Due to the ambiguity of natural languages and the fact that rules must be well defined this translation must always occur when starting with a natural language. There are not many business engines that do this translation. Most usages of natural languages are either in passive natural languages or in template-like systems. A template is an approach where the programmers code all the logic and leave the only the value definition to the end user. Even though it is a very simple solution for end-users, it is not at all flexible and every time it is needed to be extended, only a programmer can do it since it is not easily done by an end-user. One of the few real uses of natural language is present in [? ], where they created a system that does translate the natural language to SBVR. SBVR is an adopted standard for formal and detailed natural languages for defining business rules. It establishes keywords and works on a fact-oriented approach. The solution presented receives two files as inputs, the natural language file that has the logic and an UML file that has the domain. From the UML it will extract classes, attributes, methods and relationships. From those, it will map them to SBVR concepts, for example, classes are mapped to nouns and operations are verbs. From the text file, the text is first lexically analyzed. It breaks the text into words and establishes relations between words in the same sentence. Then suffixes and prefixes are checked for and removed in each word. Using a parse tree it generates a syntax structure for the treated words. Finally, it analyses the words semantically and tags them with roles to later map easily to SVBR. It then maps the words to the SBVR concepts of the domain extracted from UML. Finally, it replaces any synonyms with SBVR keywords. Being able to write rules in natural language makes rule definition more accessible to end-users. On the other hand, the need to specify the domain using UML ensures that an end-user will have a hard time without help.

These business engines are mostly aimed at enterprise business managers and office workers. This is the main reason that most business rules environments come up with solutions that are not worried about non-technical end-users. Since business rules change constantly it is advantageous to have the business manager learn how to use a business rule engine versus have the programmer constantly change the code. So even though they are aimed at an end-user and not a programmer, they are not made accessible to anybody with minimal learning.
2.9 Solution Comparison and Conclusion

To select a solution they need to be compared to each other while taking into account the requirements.

First, we compare a custom solution fully constructed by a professional versus end-user programming. By having the challenges created by the end user, most of the implementation cost is gone compared to having a custom work done by a professional that would end up costing thousands. So the end-user programming solution is preferred.

On the end-user programming there are several solutions and to choose one they need to be compared taking into account the requirements. While the flexibility of the campaign creation and scalability are important requirements, the most important requirement to fulfill is the ease of usage for new users. This is because ease of usage will be the one factor that will drive user acquisition, more shop owners will adhere to cycle-to-shop, and user retention, more shop owners will keep using cycle-to-shop. From the Table 2.1 we can see that the best solution is Form Templates since it has the best ease of usage.

To minimize the flexibility problems two approaches are to be taken: create a wide array of different templates and have an advanced mode that allows shop owners to combine parts of each Form Template to create a customized campaign. For scalability, Form Templates are the best approach since it is only required to save an identification number of the template and its input values, while on the other approaches since they are more similar to programming, it would be needed to save the code done by each shop owner. Finally in terms of security, since the logic behind each reward scheme is all implemented on the server side, while on other implementations, the shop owner would be creating the logic and sending it to the back-end, leaving it vulnerable to all kinds of attacks, like for example code injection.

Templates allow the shop owners to instantly set their campaigns without having to learn any programming language or API. Even though they are not as flexible as the previous methods, the requirement of fine-tuned personal customization is not one as important as ease of usage since shop owners, as end-users, do not have the time investment or patience to spend on a fine-tuned system.

Since ease of use is the most imperative requirement, Visual, Natural Language and By Example Programming are not the preferred solutions, since they trade ease of usage for higher flexibility. Form Template and a Custom solution by hiring a developer, on the other hand, have high ease of usage. On the other hand, by hiring a developer it will have a high implementation cost while Form Templates do not. A Custom solution possibly even requires its own server hosting which adds another cost.

Another problem with the custom solution is that for each challenge it will require an extra cost to hire the developer.

This means the best approach to take for the current problem and user base is the template-based.
<table>
<thead>
<tr>
<th></th>
<th>Visual Programming</th>
<th>NL Programming</th>
<th>PBE</th>
<th>Form Template</th>
<th>Custom Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of usage</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Flexibility</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Implementation Cost</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of possible approaches *Since with a custom solution is implemented by a company, the requirement fulfilment may vary.
Chapter 3

Architecture

This chapter presents the Reward server architecture design. The overall system can be divided into three components: the cyclist mobile tracking application, the tracker database and the Reward server. Both the tracker database and the Reward server are hosted and managed by a third party. The Reward server fully implements the shop owner reward system web application and interacts with the other two components as part of the functional loyalty challenge-based programs implementation.

An extra component is the shop owner’s offline mobile application. This thesis will not tackle this component in depth since the offline application’s objective is to provide some of the functionalities of the Reward server without requiring an internet connection.

This chapter aims to explore the use cases of the overall system and how each component interacts with the Reward server. The Reward server has two main functionalities: allowing the shop owner to implement his own challenge-reward loyalty programs and interacting with the cyclist application to handle the cyclist’s challenge progress and completion. We will look further into these two functionalities by exploring the design of the challenge-reward on the web application side and how the reward server exchanges messages with the cyclist application.

3.0.1 Cycle-To-Shop Components

The first component, the cyclist mobile track application, has as its main purpose to track the user cycling routes. It also needs to communicate with the other two components. It sends messages to the tracker database with the routes, where the cyclist has been, the paths he took, etc. This does not mean the application needs to be connected to the internet constantly. Instead, the application simply requires GPS enabled when cycling, and once the cyclist has finished his trip, he can connect to the internet and upload the trip data to the tracking server. In case the cyclist does not have an internet connection the mobile application caches the request for later, following an opportunistic approach. On the other hand, the cyclist app also needs to send messages to the reward web server to query if the cyclist has fulfilled a challenge set up by the shop owner. This is used when the cyclist requests a reward or when he wants to see what was his challenge progress. Finally, it also needs to receive messages sent by the reward server to know what shops and challenges exist and display them to the cyclist. The list of shops and
challenges received by the application take into account the user position and a user specified radius. In this project, the mobile data was designed and implemented by a colleague in Android Studio.

The second component is the tracker database; this is where the tracking data from the cyclist is stored. It uses a graph-based database to easily save positional data and a small relational database to save simple data like kilometers traveled. This database also needs to be queried by the reward server in order to obtain all the cyclists’ data. This component, while used it is not the focus of this thesis.

Finally, the reward server has two main purposes: to communicate with the cyclist application and to allow the shop owner to set up his own reward system for his shop. When communicating with the cyclist application, the reward server needs to send data about nearby shops and challenges, display the progress in fulfilling the requirements of shop challenges and whenever a cyclist has fulfilled such requirements, send a code to the application as a verification method to the shop owner that the cyclist can receive a reward. For most of the previous actions, the reward server needs to compare the requirements of the challenge with the cyclist travel data that is saved on the tracker database. On the shop owner side, it must display several templates for reward system types, save the reward system customized by the shop owner and provide a way for the shop owner to check if a cyclist code is valid.

In the next section, the use cases for the Reward server will be explored, taking into account both the cyclist point-of-view and the shop owner.

3.0.2 Use Cases

Use cases allow a better understanding of the architecture and how each component and users interact. This section describes several use cases from both cyclist and shop owner perspectives.

Cyclist Application Use Cases

As previously stated, the reward server has two main purposes that will be further explained in this section: to communicate with the cyclist application and to allow the shop owner to set up his own reward system for his shop.

First, we will delve into the uses cases with the cyclist application. One of the first use cases is when the cyclist is currently cycling and opens his application wanting to see which shops near him have challenges, which challenges, its requirements and its rewards. This is a simple use case, where the application when opened sends a request to the reward server with the cyclist GPS location, and the reward server simply searches its database for nearby shops and answers the mobile application with the shop location and a list of challenges. If the cyclist application has no internet connection, then it will only show previously cached shops and challenges by defining them as favorites on the application. Caching the requests is not important in this specific use case since when the cyclist gets internet connection back, he might be in a different position, and the GPS request will not make sense.

The next use case is when the cyclist requests a reward from a certain challenge. First, the application sends a message to the reward server requesting a reward for a specific challenge. The reward
server will then compare the cyclist’s data with the challenge requirements. To retrieve the cyclist’s data the reward server queries the Trace Server and for the challenge requirements, the reward server queries its own database. If the cyclist has fulfilled the requirements, the reward server sends a code to the cyclist. When the cyclist goes to retrieve the reward he must show the code to the shop owner. The shop owner then can verify if the cyclist should get the reward by providing the code to the reward server. In case the shop owner has no internet access the code can still be verified through the shop owner’s mobile application. The shop owner is required to inquire the cyclist’s identification number since it is used as input on the application. The result is a code that will match the cyclist code if the cyclist has a valid code.

Figure 3.1: Sequence of Cyclist Request Reward Code.

This interaction can be seen in figure 3.2. In addition, what was explained in the previous paragraph, the Reward server verifies if a cyclist has fulfilled the challenge requirements. This is done by querying the Reward database that holds the challenge requirements and rewards, the Trace server that holds several data about the cyclist and compares them.

Finally, the last interaction, visually represented in figure 3.3, happens when the cyclist commits a newly cycled trajectory, prompting a response from the reward server with an updated progress on challenges. This interaction is not triggered with an explicit intent by the cyclist to request challenge
progresses to the reward server. Instead, the cyclist simply knows he needs to commit a recently cycled trajectory. When the trajectory is sent to the Trace server, a message to the Reward server is also sent. This message has a specific list of challenges that the cyclist intends to view the current progress. Then the reward server queries the Trace server to know the cyclist data, and queries its own Reward database for the requirements of the challenge, returning to the cyclist a list of challenges and their progress.

### 3.0.3 Shop Owner Use Cases

The reward server handles the other side, which is the part that allows the shop owner to set up his account and his loyalty programs by creating several challenges for the cyclists. This is done through web pages on a specific website hosted on the reward server. The two main use cases are the challenges definition and the code validation. The challenge definition is fairly straightforward. The shop owner, on his browser, logs into his account using his Google credentials. Then, on the main page, he selects "New Challenge" chooses a template from several, inputs values on a form and submits it.
As seen in figure 3.4, the Reward server acts as a normal web server, receiving normal HTTP requests and deploying HTML pages. In this case, the server sends three pages: the main page, templates list, and the template input form. These are all predefined pages made with HTML files. The shop owner can select one template type from the several ones available on the template list page. The template input form has elements that allow the shop owner to select values he finds adequate to his challenge program. When the template values are sent to the Reward server, they are saved to the Reward database.

Illustrated in figure 3.5 is the second use case, where the cyclist requests a reward code once they complete a challenge. The cyclist receives the code and shows it to the shop owner. The shop owner then submits it to the Reward server in order to verify its validity. If the code is valid, the shop owner can then give the cyclist the reward. A more in-depth visualization is presented in figure 3.6, where the use case is represented as a collaboration diagram.

Other use cases are:

- Type 1: The shop owner defines his shop profile on Reward server.
- Type 2: Deleting a challenge.
Type 3: Multiple combined challenge definition.

Simply defining challenges is not enough, those challenges must also be assigned to a shop. For
that, the shop owner must first login on the website and on the main page select the option to define shop. In there it will request the shop name, a description, a GPS position, a beaconID and a shop logo. This data will also be the one sent to the cyclist application to be displayed.

When deleting a challenge, the shop owner would logs in into his account and then in main web page select the challenges page. There it is displayed a list of all challenges that the shop owner created. The shop owner can then select one to delete it.

Finally, there is an alternative method to creating a challenge. Once a shop owner explored the main templates and has grasped how the reward server and interactions with cyclists work, he can choose to create a more complex challenge. A complex challenge is created by combining several of the basic templates.

For example, if the reward server only had two simple templates like "A cyclist is required to cycle a certain number of kilometres between two dates." and "A cyclist is required to visit my shop a certain amount of times," but a shop owner wanted a challenge with both of those requirements, such as "A cyclist is required to cycle 5 kilometres between today and tomorrow and visit my shop twice.", it would not be possible to do by only using the simple templates. That is the gap that the complex challenge fills, and it allows the shop owner to select both the requirements and combine them into a single challenge. All that the shop owner needs to do is: login as usual, but instead of selecting a template, the shop owner must select advanced templates, and then choose both template types that have the requirements he is interested in combining, in the above example are "limited calendar travel" and "shop visits". Finally, the shop owner just needs to fill the input form and submit.

3.0.4 Shop Owner Application

One of the vital steps is the creation of the challenge-reward programs done by the shop owner. This is done through the web app that the shop owner can access. In the web app, the shop owner can also manage the challenges and verify if a cyclist has finished a challenge and deserves a reward. This step was explored in the previous section and in figure 3.2. The question then becomes, what are the templates available to the shop owner and how were they developed. In the following sections, it is explained how were the challenge templates, the interface, and the rewards, chosen and designed.

Challenge Templates Design

A loyalty program will attract new guests and turn the already customers into more loyal ones, and in this case, it will also try to influence their behavior by rewarding cycling. There are several types of loyalty programs, classified by the rewards and requirements.

The main types of loyalty programs are [? ]:

- Type 1: Members receive a price discount off a static amount at the register. Popular implementations are usually a discount of a percentage of the total price or a discount of a specific monetary value.
• Type 2: Members receive one unit free when they purchase n units.

• Type 3: Tiered rewards. Once a customer receives a reward, his following rewards will be of increasing value.

• Type 4: Customer relationship with frequent special offers.

When planning, several program types were discussed. The first one was the simplest one, once you register to the Reward server you provide a static discount to some products. While this is simple, it offers no flexibility, and it does not encourage the use of bicycles since many shops and restaurants offer similar or equal discounts.

The next one was a challenge type system, where shop owners would create a challenge with certain requirements related both to cycling and their own business. This offers a bigger flexibility than the previous static discount and also uses gamification to make it more interesting for cyclists.

As rewards for these challenges, we thought of a bankable points system, where cyclists collect points depending on their actions and trade them for specific rewards. These points can either be universal or specific to each shop. In the case of universal points, there is a risk of actually decreasing the reason for shops to sign in since cyclists could cycle around their homes and then use the points in the shops even though the cyclist did not visit the shop or its vicinities. The result would be a happy cyclist but an unhappy shop owner since it was a single interaction, possibly with no profit to the shop owner, with no loyalty increase.

Another problem with a point system is the possibility of having different shop owners that would have different point amounts as rewards. A shop could reward a single point for a simple action but require dozens of points for rewards, while others could reward dozens of points but require hundreds of points for rewards. If points worked in between shops, a cyclist could be rewarded a fairly high amount of points from one shop and trade for multiple rewards from a shop with low-cost rewards. So instead of points as currency, having the rewards tied to requirements avoids this problem. For the case of shop-specific points, while it does not have the previous problems it would be harder for the cyclist. Instead of managing a single currency, the cyclists would be forced to manage several currencies with different amounts and rates for each shop.

The reward server provides several templates to allow the shop owner to customize his own reward challenge. Since templates are an easy approach for shop owners to create their own challenges, as concluded in Chapter 2, it is important to choose which templates to create. These templates were chosen and implemented by hand and are divided into two parts, the visual web page, and the functional code. So for any new challenge template to be introduced these two parts need to be implemented.

The template method itself can be done in two ways. Either a single template, that has a multitude of options to customize into one of the several types of challenges, or several simple templates with a single challenge type in mind. A similar solution discussed in Chapter 2, called BetterPoints, had a single template that was several pages long which made it difficult to go through and understand what each option does. Limiting each template to a unique simple challenge type results in a more intuitive
The templates were chosen with the intent of promoting bicycle use and to the shop owner a way of increasing customers and customer loyalty. It also took into account other possible avenues like museums, schools, and others.

To choose the templates, current popular challenges from existing loyalty programs were taken into account. The simplest one is simply to promote cycling. Once a cyclist does a certain number of kilometers, chosen by the shop owner, he can get a reward. The intent of this challenge is to promote events since it simply requires the cyclist to travel and nothing more.

The next template is like the previous one, but the shop owner can choose a city where the cyclist must ride. The objective is the same as that of the previous template, to promote a more active lifestyle but in a specific city. This can be used to promote an event of a specific municipal center for example.

The next template is “Cycling between two dates”, which makes it very much like the first one but with a date restriction.

“Cycle and win a raffle”, is a template that allows shop owners to create a raffle, giving a more expensive reward to a single user randomly.

“Visit the shop’s owner shop while cycling”. The visits are tagged by the application when it detects a special beacon on the shop. This template can be used as a visit frequency loyalty program, where the cyclist that visits the shop the most receives better rewards. It can also be used to emulate the type one loyalty programs, by having the discount coupon as a reward. In theory, the shop owner can also only allow the cyclist to detect the beacon after the performing a transaction or using a service, causing this template to work as a type two loyalty program since the cyclist is only rewarded after buying something n times. Which is very similar to how stamp cards work.

“Happy hour”, which is a popular discount program to attract clients in bars. The cyclist must visit the shop, very much like the previous template, but in a restricted time frame. It can be used by restaurants to promote client visits during hours where there are not many clients.

Finally, the last template is “Cycle and the number of previously completed challenges”. This allows shop owners to create the type three of tiered challenges by creating some simple challenges, then have some harder challenges that require a certain number of previously completed challenges.

Challenge Template Architecture

Challenges are organized on a two-file-per-challenge basis. Each challenge is implemented in two parts: the visual part, and the logical part. The visual part is what appears for the shop owner, containing the interface where the shop owner inputs his values and sends it to the server. The logical part contains the functionality that is called when the reward server needs to verify if a cyclist has finished a challenge. The logic needed to verify if a cyclist has finished a challenge encapsulates: which queries to call on the TraceDB and the RewardDatabase, which and how to compare the values and outputting if a cyclist has finished the challenge. The logic can also output a progression value for how much the cyclist has completed the challenge instead of a simple boolean state of completed or not.
3.0.5 Rewards

Once the cyclist requests a reward code and the reward server verifies that the cyclist has completed the challenge, it creates a reward code. The reward code is generated by hashing three values: the cyclist identification number, the challenge identification number and the day of the request. This generates a simple four letter code. It is important to be simple since the cyclist will have to easily display or tell the code to the shop owner. The shop owner will also be busy with his usual activities and other customers, so this verification should be simple and short. A simple string is the most straightforward method of authentication, user-friendly for the non-computer-savvy shop owners and it does not require extra equipment like barcodes and QR codes.

The three values used are needed to create a unique code for each person-challenge combination on a specific day. This is important because if the shop owner does not have an internet connection, he can use a separate shop owner specific mobile application. This mobile application also receives the challenge id number, the cyclist id number and the current day and generates a four letter code. If both the shop owner app generated code and the cyclist server generated code match, then it means the cyclist should receive the prize. Unfortunately, due to this requirement, it means that both the offline shop owner application and the reward server need to have the same reward code generation and it must be deterministic thus rendering salt-like techniques to improve the hash security impossible.

Once the cyclist gives the code, and the shop owner verifies its validity through the server or his mobile app the cyclist receives the reward. The reward is saved and displayed as a simple word, as to avoid extra costs for the shop owner that would come by forcing a specific hardware piece that reads and registers specific products barcodes as rewards. The barcode method is also less flexible since it does not allow services, discounts or products that are not sold in that shop. This means that a reward can be pretty much anything from a flat discount on a restaurant, a free meal, free night in a hotel, a free trip, and others.

3.0.6 Interface

When the shop owner first enters the website, he is presented with the main page that includes a small explanation of the service provided and a Sign in button using Google account. This is presented on figure 3.6, where the main page displays a button for sign in through Google.

Once he signs up the shop owner is presented with his options as shown on the page presented on figure 3.7. Those options are: create a new challenge, view current shop owner challenges, verify cyclist reward code and set shop profile.

Once the shop owner selects to create a new challenge on the main menu, he is sent to the challenge template menu in Figure 3.8. In this menu, the shop owner can select what type of challenge he wants to create from the list previously mentioned.

Once the shop owner selects it, he is faced with the input form that lets the shop owner customize the challenges’ values. All of the input forms request a title, description, and reward for each challenge since those are shared for all challenges, while the challenge requirements vary depending on the type.
In the case of the Figure 3.9, it requests how many kilometers a cyclist should travel before completing the challenge.

Once the shop owner confirms the challenge he is redirected back to the main menu page.
3.1 Tracking Server and Cyclist Application

Once the shop owner creates his challenges program, the cyclist needs to be able to view them, follow his progress, complete the challenges and receive the reward. These actions complete the interaction cycle between the shop owner and the cyclist and are handled by the reward server via interactions with cyclist application and tracking server.

3.1.1 Communication

The reward server needs to receive data from several sources: the shop owner, the cyclist mobile application, and the geospatial Trace database.

The Trace database was already created and therefore already had a messaging protocol defined. To query the Trace database and receive a cyclist route data a message has to use a RESTful API and go through the HTTP protocol. The challenge program creation by the shop owner has to be simple and easy to use. Because of this, the shop owner challenge creation application was decided to be made as a web application, accessible in a browser, so using HTTP was a given, and therefore HTTP verbs as well.

The communication with the cyclist mobile application for the sake of consistency also works through HTTP verbs. Another advantage of using HTTP as a communication protocol is that it is implementation independent and therefore platform independent. By using HTTP, the mobile application can be made to work natively, as a web application or a hybrid. The hybrid solution is an application implemented with web technologies like HTML, CSS, and Javascript but hosted inside a native application that works as a full-screen web browser.

Since there is some sensitive information, like the cyclist position or the reward code, messages are sent through HTTPS, Hyper Text Transfer Protocol Secure. HTTPS is actually HTTP over a secure protocol that ciphers the communication, which is usually either TLS or SSL. HTTPS provides security by ensuring authentication, encryption and data integrity. These are implemented by the secure protocol
extra handshake at the start of the connection, where the server sends a certificate, this certificate has a key, allowing the server and the client to share a secret and establish a secure connection. This certificate is generated from a certificate authority which ensures authentication.

This is important, since shop owners with potentially big rewards, like a car or a free vacation, will be a target of attacks like man-in-the-middle attacks or simple packet sniffing. By having the sent reward code encrypted, an attacker can not capture it and pretend it is his. Data integrity ensures an attacker with a fake reward code can not give it to the shop owner, waiting for him to request a verification on the browser and have someone tamper the server message turning a “this code is invalid” into “this code is valid” kind of message.

The cyclist’s GPS location is sent when the cyclist queries the reward server about nearby shops and challenges. This GPS data is also sensitive data since an attacker can over time create a route based on usual GPS locations and find a cyclist’s real location. Since the message is two-way encrypted, the cyclist is also protected from packet sniffing.

From the side of the cyclist mobile application, most messages are cached once it loses internet connection. This is especially important for when it commits a new bicycle trip to the Trace database, requests a challenge progression update and when he requests a reward code.

### 3.1.2 Database

Each shop’s relevant data and challenges need to be stored so that the cyclist can later receive them. These are stored in a database organized as presented in Figure 3.10.

Each shop owner has associated a shop profile, which the shop owner inputs on the website. The shop profile consists of the shop name, shop GPS latitude, and longitude, identification number of the low range beacon used by the cyclist mobile app and an image logo reference.

Each shop owner can create his own challenges. Challenges are saved in separate tables. On the challenges table, the metadata is saved of each challenge. These are comprised by the challenge name and description, date of set up, type between simple template or a combination of templates, unique id number, and reward for completion. For each of these entries, there are one or more entries on the challengeTemplates table.

This table holds, for each challenge, the template id number, so that later, when the cyclist requests a reward code, depending on the template type, it can use the correct function to verify if the cyclist has completed the challenge or not. It also holds the requirement values chosen by the shop owner, which are required to assess if a cyclist has completed the challenge or not.

Once a challenge is completed and a cyclist requests a reward code, the code is generated and it is saved in the challengeCompletion table that works as a whitelist. In the table, the challenge id number, the cyclist id number, and the reward code are all saved. This table also avoids the same person using the same reward code several times or sharing it with friends as to have multiple uses. Each code entry is only saved for a day.

Once the reward code is shown to the shop owner and verified to be correct its entry on the table
is deleted. In its place, it is either added to the rewardUsed if the challenge had not been completed by the cyclist before or incremented. The date of the last time it was completed is so that it resets the progress of the challenge, and the next time a cyclist requests a reward code, the server will query the Trace database how many kilometers has the cyclist done since the last time he completed the challenge. This avoids the same cyclist requesting several times a reward code for the same challenge. This table is also used for the challenge template, that requires the cyclist to complete a certain number of challenges since it counts how many times each challenge has been completed.

For the raffle challenge, there are two extra tables. One saves each cyclist entry on live raffle challenge, and once they end, an entry is selected randomly and saved on the finishedRaffles table while all other entries in the previous table for that challenge are deleted.

The users’ table is used to translate the identification used by the third party doing the login authentication, in this case through Google account, to another id value to be used only inside the reward database.

Figure 3.10: Reward Database which holds shop and challenge data.
Chapter 4

Implementation

This chapter presents the implementation details of the Reward server, a component of cycle-to-shop that handles the creation of reward schemes, processes cyclist data in order to see how far along a challenge he is and confirms if a cyclist can collect a reward. The section will tackle implementation decisions like which technology was picked for the development of the web server, and how the authentication is done and the message architecture is structured.

4.1 Node.js

As referenced in chapter three, the Reward server is to be implemented as a web server. This allows the shop owner to create his reward scheme in an environment familiar to him, the web browser. It also allows the cyclist app to be easily ported from one mobile platform to another. One tool currently being used a lot to develop web servers is Node.js.

Node.js is an asynchronous event-driven JavaScript runtime that is specially designed to build scalable network applications. So unlike the more common servers that go with the concurrency model where multiple threads are used, these design choices are perfect for scalability and optimized throughput. Some current examples of companies using Node.js as a web server solution are LinkedIn, Microsoft, Netflix, PayPal, and Yahoo!. Node is also easy to work in part due to its module-centric nature, allowing developers to easily use community published modules, and also because it uses JavaScript, meaning that the developer can use the same language both for client and server side.

While Node.js is enough to create a complete web server, the Express framework for Node.js was used to make development faster and easier. Not only it is a fast, minimalist framework specific for web server development but it also provides multiple useful middleware modules, like Passport which was used for the shop owner authentication.

4.1.1 Choosing Node.js and alternatives

One of the most popular web server solutions is Apache, which is exclusively a configurable HTTP server. Apache serves static files like images and allows server-side scripting through PHP. While
Apache can use other languages for server-side scripting, that is done through the use of third-party frameworks. Another disadvantage that Apache has over Node.js is that it is thread-based, resulting in Apache being more inefficient and harder to develop in. This is due to possible deadlocks and thread management, which Node.js does not have.

Nginx is, very much like Apache, a popular HTTP web server that uses PHP. Unlike Apache, it is not thread based but in fact it is asynchronous like Node.js.

In the 2015 study \[?\], several web server technologies were compared, taking into account the fact that their architecture is either synchronous or asynchronous and how they perform. In this paper, it is concluded that asynchronous architectures are far superior to the synchronous architectures due to performance differences in answering client requests under high concurrency. This is due to the limited queue size, basically the number of threads, which causes long response times. It also shows that just increasing the number of threads in the thread-based synchronous architecture is simply not enough, and instead it will rise the multi-threading overhead.

When comparing the HTTP server approaches, in the evaluation study done \[?\], the Apache/PHP approach was proved to be several times more inefficient when handling a high amount of network traffic. Comparing Apache with Nginx, we conclude that Nginx is several times faster in IO operations. Node.js, on the other hand, exceeds both several times. In the very same study, the web server tools were also evaluated regarding CPU consumption and memory usage. When comparing CPU usage, Apache and Nginx had fairly similar values, while on the other hand, Node.js had 1.5 times more CPU usage. When comparing memory usage, Node.js has the best memory usage, followed by Apache and finally Nginx. Even though Nginx uses more memory, Apache is the only one that starts failing requests. Another test done in the study was a hashing test, as a way to test computational overhead. In this test, not only Nginx loses most of his performance from the previous test CPU test but Node.js outperforms both Apache and Nginx by a factor of 2.5. This is due to both Nginx and Apache using PHP which does not handle data processing as Node.js. In the very end of the study, the title question “Is end-to-end JavaScript a viable option for building modern web apps?” is answered with a positive “yes, and highly advisable.”

Similar results can be observed in the evaluation \[?\], where Node.js was compared with PHP/Apache and Python. Several tests were performed, ranging from a simple "Hello World" test, a more CPU intensive test like calculating a number in the Fibonacci sequence and querying a database. Node.js was superior to the alternatives in all tests. In the simple message test, Node.js performed twice as better than PHP and six to seven times better than Python. Not only that but Node.js can hold more users than both, resulting in better scaling. With the Fibonacci test, all three web technologies decrease in performance significantly, and even though Node.js performs better than the remaining two, it shows that all three are not great solutions for servers with CPU intensive actions. In conclusion it is worth noting that Node.js surpasses the traditional PHP, especially with high concurrency cases.

With this information, Node.js was the selected technology due to: its asynchronous architecture that results in a very scalable server by nature; the performance results when comparing to Apache, PHP and Python; it is module based which makes it easy to develop to and the fact that uses Javascript,
allowing the same developer to easily develop both the client and the server side.

### 4.1.2 Shop Owner Login

For the shop owner to create challenges, associate them with his shop profile, and associate his shop profile with his real life shop, the shop owner requires all this data to be tied to an account of his own.

The shop owner login is based on Social login, which is a form of single sign-on using a third party to create and authenticate the account. The login in the reward server uses Google as the identification provider, but it can easily be expanded to others like Facebook, Twitter, LinkedIn, Instagram, and many others.

This is done through the Passport middleware. Passport is an authentication middleware for Node. It is designed to serve a singular purpose: authenticate requests. Passport allows the application to select which third-party API to use for authentication through separate modules called strategies that act very much like plug and play modules. This allows a fast addition of other sign-on platforms besides Google, like Facebook or even a local strategy with username and password.

To use Google as an identity provider, the reward server and URLs are registered as an application with Google. Google then issues a client ID and client secret, which need to be provided to the strategy. These are used for OAuth and OpenID Connect, the two popular single sign-on protocols.

Once the authentication finishes successfully, a session id will be established and saved in a cookie on the user’s browser, so that then each following request will use that session id.

The reward server uses OpenID Connect as its authentication protocol which is built on top of OAuth2.0.

### What is OAuth?

OAuth is a standard protocol that allows users to authorize API access to web, desktop or mobile applications. Once access has been granted, the authorized application can utilize the API on behalf of the user. OAuth has also emerged as a popular mechanism for delegated authentication. While OAuth has been used for authentication, it is an authorization protocol, which makes it dangerous to use as authentication [? ]. Authorization is a workflow that only ensures that a specific user has permissions for something by possession of an access token, but this access token is completely opaque and has no information about the actual user identity. Because OAuth intent is purely authorization, it also resulted in a weaker security by relying purely on TLS.

An example of a OAuth2 use case might look like this: Bob signs up for a new account in a website and is offered the option to see which of his friends are already registered on the website. There’s a button labeled “Import contacts from Facebook.” Bob clicks that button and is redirected to Facebook to log in. Bob successfully logs in and is asked if he wants to share his Facebook friend list with the website. Once he accepts the website receives an authorization token from Facebook giving the website permission to access Bob’s friend list.
What is OpenID Connect?

OpenID is an open standard for authentication, promoted by the non-profit OpenID Foundation [? ].
A shop owner must have an OpenID account from an OpenID identity provider, in this case, Google. The shop owner will then use that account to sign into the website without having to register.

OpenID Connect is a delegated authentication protocol built with OAuth 2.0 as its basis. A delegated authentication protocol means that one site is simply outsourcing its authentication needs to another pre-selected site. It uses simple REST/JSON messages between the identity provider and the service the user is using. It is easy for developers to integrate and users to use, when comparing to other authentication protocols. It is simple for developers since they do not have to store and manage their user’s passwords while still providing a secure identity of the user. It is simpler for the users because they now do not have to register to the service and remember a new set of password and identity name.

OpenID Connect allows for clients of all types, including browser-based JavaScript and native mobile apps, as well as launch sign-in flows and receive verifiable assertions about the identity of signed-in users.

Besides the fact that OpenID provides user identity through identity tokens, which are essentially JSON web tokens, it also adds a security layer on top of OAuth 2.0. This security layer is done through encryption, using a public key on the web token.

4.1.3 Heuristic Approach for Interface Design

When developing the reward server interface, ten usability heuristics were followed as guidelines [? ]. The ten heuristics are:

- Visibility of system status. Every single web page has a title that displays the current purpose of the page, ensuring that the shop owner always knows what is going on. Small pop ups are also sent in some steps in order to provide feedback.

- Match between system and the real world. All language is simple and relatable, by using real world language like challenges, cyclists and shop. Challenge templates also relate directly with real world actions, making them easier to understand.

- User control and freedom. All web pages have a back button that allows the shop owner to return to the previous page.

- Consistency and standards. For instance the following buttons are all consistent with other application’s buttons: “Back”, “Next”, and “Submit”.

- Error prevention. A shop owner cannot create a challenge before registering his shop first. This not only prevents errors but also directs the shop owner towards registering his shop as his first action. A shop owner also cannot progress on the advanced challenge creation without selecting two or more templates to combine.
• Recognition rather than recall. Due to the simplicity of the template method and that after selecting a template the shop owner is presented a form specific to the chosen template, there is no case of recall.

• Flexibility and efficiency of use. The advanced combinable challenges allow the advanced users to create more flexible challenges. Catering both to new and veteran users.

• Aesthetic and minimalist design. This was especially used in the challenge creation interface. By having a minimalistic design, it reduces the amount of information at once, allowing the shop owner to easily make decisions of what templates to pick in order to create his desired challenge.

• Help users recognize, diagnose, and recover from errors.

• Help and documentation.

4.1.4 Database

Whenever a new shop owner signs on to cycle-to-shop, his business data needs to be saved, such as his shop name, location and description. Every time a shop owner creates a new challenge, all of the challenge parts like the name, description, requirements and rewards also need to be stored. All of this data is stored in a database.

MySQL, a relational database, was the database system used to save the shop owners shop profiles, the challenges they create and other data, which was explained in the Architecture section. MySQL has joins which simplify queries while document databases like MongoDB do not, requiring extra code, that will clutter the server code and make separation of concerns between database and server code.

On the other hand, a document DB like Mongo would be a good alternative since it allows for a more dynamic schema. It was also built with sharding and replication in mind. Conversely, it does not have transactions, it is not ACID-compliant, and cannot create relationships between document entities. This means that if a shop profile is deleted, it will not delete its challenges. One way of fixing this would be to have the all the shop profiles and its challenges in the same document, but this would result in a very large document since every data that has some dependency would need to be aggregated into the same document.

Graph databases like Cassandra shine when there are deep-level relationships between entities, as for instance in social media, where it is important to determine friends of the user, friends of friends, etc. This is not the case since most of the new data saved in the database will be new challenges, which do not increase the depth level of relationships between entities.

MySQL was chosen due to previous experience with the technology and with relational databases as well, making it much easier to develop since there was no need to learn a new technology. Not only learning a new database type would have taken extra time and effort but also the development of code that handles joins and other functionality is provided by MySQL by default.

While the non-rigid schema that MongoDB and other document databases provide would have been useful, specifically for the different types of challenges, that hold different amounts and types of data
with MySQL, it can be worked around it. In this implementation, the challenge requirement values are held in a single column and serialized as a JSON array. This allows new challenges to be introduced without changing or creating a new table or database. The disadvantage is that if a single value of several challenges is needed, it cannot be simply queried, instead it needs to retrieve all arrays, deserialize them and then organize and process the data needed. Fortunately, this is never the case since challenges requirements are only needed purely for display to the cyclist and to confirm if the cyclist has finished a challenge, in which case all requirements are needed. This would also be a problem if the shop owner wanted to alter his challenge, but luckily this is not allowed to avoid "Bait-and-switch" type fraud where the shop owner alters the challenge requirements halfway to make the challenge harder or changing the reward itself.

Another possible approach would be to have a table for each challenge type; this would only have the drawback of having to add a new table every time a new challenge type is to be added.

4.2 Interaction with Cyclist application

The interactions between the cyclist mobile application and the reward server are done through HTTP messages following a RESTful API.

REST, Representational state transfer, is a style of software architecture that is very popular with web services and is widely used by the World Wide Web. Its two main characteristics are the limited interface with a set of well-known operations and how it focuses on resources instead of operations. It works especially well in a web environment like a web server, due to HTTP already having a well-known, constrained methods known as HTTP verbs (GET, POST, DELETE, PUT).

When comparing a REST style, which is resource oriented, with an RPC-style architecture, which is process oriented, REST shows several advantages:

- Reduced complexity in the interface, unlike RPC-style approaches where each one has its own operations and requires a discovery contract.
- Unlike standard RPC style, it is a stateless style, meaning that each message has all the information needed.
- Reduced coupling between the client and the server due to the lack of a service interface contract.
- Reduced parsing and packaging overhead due to the use of HTTP verbs.

A big advantage of using REST over using something like XML-based SOAP messages is that it has a much lower overhead. While this difference might seem inconsequential for home computers, the cyclist will be using an application on a mobile phone, which has lower performance. Not only mobile phones have a wireless connection unlike most home computers that have wired connections but it also has limited battery lifetime, limited processor speed and a more volatile wireless network connection. One of the reasons for such difference is the very small message size, which is smaller than conventional SOAP. The response time is also smaller, due to easy to process messages. Less message size and
response time means less processing and transmission time which leads to lower power consumption, and faster web service.

REST has several advantages like [? ]:

- Improving scalability by being stateless, which means that the server has no need to maintain a communication state for each client, reducing resources and increasing server simplicity.

- A simplified uniform interface that makes client application development simpler.

- Resources identified via URIs, which make them both self-descriptive and easily readable by humans.

- Provides equivalent functionality when compared to alternative approaches to communication.

- Does not require a separate resource discovery mechanism, due to the use of hyperlinks in content.

When the reward server requires querying the cyclist cycling data he needs to query the TraceDB. TraceDB is queried through HTTP messages using the GET verb. A GET verb is used in messages that are purely read request of data. These requests are done through an RESTful-style API.

### 4.3 Conclusion

In this chapter, several technologies were presented, leveraging the pros and cons of each and the chosen technology for implementation. The technologies presented tried to solve several problems like the implementation of a web server, how to allow the shop owner to easily login and authenticate him and store the shop and challenge data. In the section about the communication between the Reward server and the other two components, the cyclist application and the Trace server, the main subject was about the message architecture style called REST, its advantages and why it was chosen.

In the next chapter, an evaluation of the system is presented.
Chapter 5

Results

In this Chapter, the experiments and results of the evaluation of this thesis are presented. The objective of these tests is to determine if the Cycle-to-Shop Reward server achieves the requirements presented in Chapter 1 of ease of use and flexibility. Ultimately, we present the experimental results obtained and corresponding conclusions.

5.1 Requirement 1 : Ease of Use

Ease of use is one of the most vital requirements since it is the deciding factor for user retention. This section examines the experiment and the results, to try and verify how easy to use the template approach and the interface is.

5.1.1 Experiment

This test tries to determine how easy it is for users to understand and use the interface and template forms. Each test subject is first given a small explanation of what are the roles of the shop owner, the reward schemes and the cyclist. The test subject also has as much time as he wants to first read the main page. Each test subject is then presented with seven tasks that encompass the shop owner’s use cases. For each task, the number of mouse clicks are counted and also the time it took to complete the task. Since test subjects were encouraged to “think aloud” during the task, any opinion or mistake committed were also noted. The objective of this experiment is to attempt to quantify the usability of the interface, pinpoint any interface mistake and verify how easy it is for users to understand how to create a challenge from templates.

The tasks are:

- Task 1: Sign on with Google and create a store called "MyRestaurant", in Av. Rovisco Pais 1 Lisboa, with “This is my restaurant” as it's description and 21334512 as the beaconID number.

- Task 2: Setup a simple challenge where if someone cycles to your store 3 times he gets a 10% discount. The challenge is called "Cycle To Me" with a description of "Everyone should cycle".
• Task 3: Setup a simple challenge where if someone cycles 10 kms in April he gets a free meal. The challenge is called “Cycle Alot” with a description of “Everyone should cycle”.

• Task 4: Go see your setup challenges and delete the “Cycle A lot”/cycling 10 km one.

• Task 5: Setup an advanced challenge where if someone cycles to your store and wins a raffle he gets a 5euro discount. The challenge is called “Visit and Raffle” with a description of “Try your luck”.

• Task 6: Setup an advanced challenge where if someone cycles 6kms in your city and visits during happy hour gets a free drink. The challenge is called “Cycle and drink” with a description of “Be happy cycling”.

• Task 7: A cyclist enters the store and tells you his reward code is 6772, verify if it is true.

The first task is the initial contact with the website and encompasses both the sign on using Google and the setup of the shop. The two following tasks require the subject to create a challenge from the description by picking the correct template. For each pair of challenge creation tasks, 2-3 and 5-6, the first of the two is more straightforward than the second. The fourth task is simply to view the previously created challenges and delete one of them. The fifth and sixth tasks are similar to the second and third, but it requires the test subject to understand how the combined templates work. These also increase in difficulty. The final task is to verify if a cyclist code is valid or not.

Testing was done in rounds, and between each round, the interface was improved taking into account the results and user feedback. Each round tests five users, with three cycles done total. This is due to how usability problem detection shows diminishing returns when testing more than five users [? ?]. Because of the diminishing returns, it is better to do three rounds of five users with improvements in between rather than testing fifteen users at once [? ].

5.1.2 Results

From the results of the first round of tests shown in table 5.1, some conclusions can be taken.

The first task has no errors and no extra mouse clicks implying that it is an easy to achieve even though it is long.

For the two next tasks, where the user is required to create a new challenge, they presented similar times and a low amount of extra clicks. The second challenge creation task had two users select the wrong template. A hypothesis was proposed that since this task is not as straightforward the template label needed to be changed to be simpler.

The fourth task was short, but some users had some trouble understanding how to delete a challenge after entering the correct page that lists existing challenges.

The fifth and sixth tasks are the advanced challenges which were the challenges with the most mistakes and time spent. On the first of the two, most users had difficulty understanding the concept of combining templates into a new challenge. This is represented in the time spent and mistakes made where the users did not combine challenges but instead picked a single challenge. On the second
advanced challenge creation, since the users now had some understanding of how the template combination works, users were faster and performed fewer errors and fewer mouse clicks.

The seventh task is simple and most users can complete it in a small time with no errors and no extra clicks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Avg. Time</th>
<th>Avg. Extra Clicks</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:51</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1:11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1:22</td>
<td>0.8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0:37</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2:17</td>
<td>1.6</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1:42</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0:30</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Results of the first round of user testing.

Between the tests, most of the improvements done on the interface were changes to text in an attempt to have users recognize faster what the interface buttons and labels mean. After the changes were implemented the second round of tests were made and the results can be seen in table 5.2.

The first task stayed the same, both with similar time, no extra clicks and no extra errors.

The second and third task show similar times but have reduced errors. Possibly from the change in challenge templates labels, making it easier for the users to understand which template to pick.

The fourth task also stayed overall the same, with a small improvement overall. This improvement was after changing the interface to have obvious buttons to delete challenges. The only user error was also immediately corrected by the user without the tester pointing it out.

The fifth and sixth task still show the difference in time between the user's first contact with the template combination and the second time using it. The number of errors stayed the same.

The seventh task was unchanged and the results stayed overall the same.

<table>
<thead>
<tr>
<th>Task</th>
<th>Avg. Time</th>
<th>Avg. Extra Clicks</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1:00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1:21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0:16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2:59</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1:35</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0:15</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Results of the second round of user testing.

The biggest changes in the interface between the second and third were focused on the advanced challenge creation menu. By taking into account the difficulty users had with understanding the combination of templates and the heuristics presented in Chapter 3 some improvements were implemented. The biggest help was blocking users from progressing until they choose two or more templates. This forced users to pick several templates and then understand the resulting form how the combination works.

For the fourth task, a bit more information was provided on the screen to avoid users from deleting the wrong challenge.
<table>
<thead>
<tr>
<th>Task</th>
<th>Avg. Time</th>
<th>Avg. Extra Clicks</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1:05</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1:21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0:18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1:59</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1:26</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0:18</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: Results of the third round of user testing.

From these results, we concluded that is fairly fast for users to set up their own challenges. Users had no problems with the interface with exception of the advanced challenges where the first contact is not easy to understand and might require a different approach to the interface. From user feedback and notes that were taken thanks to the “think aloud” methodology, the template and forms interface were simple to understand and easy to use.

5.2 Requirement 2: Flexibility

While usability is vital to not scare away potential shop owners, flexibility is still required to allow shop owners to implement the reward schemes of their choosing. With templates, this means striking the balance between not overwhelming shop owners with multiple templates and choices while still providing useful and meaningful options.

As referenced in Chapter 3, the templates created to take into account the popular reward schemes from several fields.

There are currently six popular implementations of reward schemes [?]. These are: Point based, Tiers, Premium, Customer values, Business partnership and Gamification. These six are more specific implementations of the four types presented in chapter 3. From these six, four are easily implemented using Reward servers templates. These are the: Point based, since both the distance traveled could be thought of as points, but the shop owner can also define points as a reward. Tiers can be easily created by using the template that takes into account how many other challenges the cyclist has finished. This makes possible the creation of a reward scheme where only the most loyal cyclists have the bigger benefits. Customers values are partly implicit by the nature of cycle-to-shop since it will inspire customers that value health and the environment. By creating rewards tailored to these customers the reward scheme is even more based on customer values. Gamification like the customer values is also implicit in the reward server. This is due to the server working with challenges and progression values, very much like games do with missions and experience. The two remaining reward schemes can not be implemented using the Reward scheme.

When comparing with the reward schemes in existing solutions, the Reward server templates allow most of the reward schemes implemented in these existing solutions to be reproduced. From the total eleven existing solutions, there is a total of fifteen reward schemes. These are:

- Four implementations of “Customer must travel a specific number of km”.

Six implementations of "Customer must visit the shop a specific number of times".

Three implementations of "Visit the shop for the first time".

One implementation of "Raffle".

One implementation of "Happy hour".

One implementation of "Liking the shop Facebook page".

One implementation of "Customer Birthday".

From these, the shop owner can easily implement the two most popular just using the templates provided: "Count the distance traveled by cyclist" and the template "Count the number of times the cyclist came to your shop". The third most popular, while not possible through the simple templates, it is possible when using the advanced challenge creation. The process is the combination of the number of visits template with "Count the number of times the cyclist completed other challenges", allowing the shop owner to create a challenge that triggers if the cyclist visits once and has never completed a challenge with the shop owner, which would be the first visit. The next two challenges have their own templates, making those also simple to be created. The remaining two, on the other hand, do not have a way to be implemented by the shop owner. Instead of these last two, it was preferred to expand the basic "distance traveled" template since the main objective is to motivate people to cycle more.

Besides the normal template selection, the advanced challenges further expand the Reward server flexibility. The advanced challenges allow combining several templates into a single challenge. From the seven existing templates, 127 unique challenges can be created as seen in the expression 5.1.

\[ \sum_{i=1}^{n=7} n^C_i = 127 \]  

This allows a vaster array of challenges to be created when comparing to the simple template approach. Since the advanced challenge creation is aimed towards veteran users it can also hold more than the normal seven templates. This results in an exponential increase in unique challenges, while not flooding the advanced users with information.

It can be concluded that while somewhat limited to the templates, the Reward server is able to create the currently popular reward schemes and provide a wider array of challenges through the combination of templates.
Chapter 6

Conclusions

The objective of this thesis was to provide an easy way for shop owners to create reward schemes in order to get new customers. These reward schemes, composed of challenges, would be focused at cyclists, in order to promote people to cycle more. While some applications already exist that attempt to allow shop owners to create reward schemes, these are either too complicated or too restricted. Overall the implemented solutions do not conform to this thesis requirements. Several solutions were explored in Chapter 2, in ways to allow shop owners to easily create their own reward schemes. These were programming-by-example, natural language programming, Visual programming, and Templates. The template forms strike the best balance between usability and flexibility while having a low cost to implement for the shop owner. The Reward server was then implemented as a simple web application with templates. Once implemented it was then user tested. Experimental results prove that the template approach for the Reward server achieves the usability requirement while maintaining a good amount of flexibility.

6.1 Future Work

In a real world versions, there are several tools and functionalities that would be deemed useful that were not implemented since they were out of scope. One of these features would be to provide the third party hosting the Reward server with an interface. This interface would allow the third party to verify if a shop owner is, in fact, a shop owner or to control and limit the shop owner challenges deployed. Besides control, the interface allows the possibility of deploying a premium version with a possibility of better features for shop owners that register. Another useful feature is the ability to see cycling activity on a heat map, to allow the shop owners and the hosting third party to better plan which challenge to create or where to deploy resources.

A technical improvement that can be done is database sharding. This is helpful in the case of very high scalability since splitting the database over several machines improves load balancing. Another technical aspect is security. A separate version of the cyclist reward code generation. This version would not create a four letter code but instead a more complex and secure code. It would also not be
used on the off-line shop owner mobile application in order to keep it secure.
Bibliography