Titol:	*"Tipping offers e-commerce project"*
Alumne:	*Pol Miró Omella*
Data:	*15-06-2012*
Director/Ponent:	*Ernest Teniente*
Departament:	*Enginyeria del Software*
Empresa:	*Rewardli Inc., San Francisco, USA*
Titulació:	*Enginyeria en Informàtica*
Centre:	*Facultat d'Informàtica de Barcelona (FIB)*
Universitat:	*Universitat Politècnica de Catalunya (UPC) BarcelonaTech*

DADES DEL PROJECTE

Títol del projecte: ***"Tipping offers e-commerce project"***

Nom de l'alumne: ***Pol Miró Omella***
Titulació: ***Enginyeria en Informàtica***
Crèdits: ***37,5***
Director/Ponent: ***Ernest Teniente***
Departament: ***Enginyeria del Software***
Empresa: ***Rewardli Inc., San Francisco, USA***

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: ***Maria Ribera Sancho***

Vocal: ***Juan Carlos Cruellas***

Secretari: ***Ernest Teniente***

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Index

1 PREFACE **10**

2 INTRODUCTION **12**

2.1 THE COMPANY 14

2.2 THE TEAM 15

2.3 GOALS 17

2.3.1 Give access to wholesale offers for SMB. *17*

2.3.2 Grow the user base *17*

2.3.3 Provide merchants with a quick release of stocks *17*

2.4 SCOPE 18

2.4.1 Personal Responsibility *18*

2.4.2 Contributions *18*

2.4.3 Other *19*

2.5 ACTORS OF THE SYSTEM 20

2.5.1 Administrator *20*

2.5.2 Shopper *20*

2.6 METHODOLOGY 21

2.6.1 Stand-up meetings *22*

2.6.2 User stories *22*

2.6.3 Task-board *23*

2.6.4 Group sharing tools *24*

2.6.5 Version Control System *25*

2.6.6 Flow of work *27*

2.6.7 Server administration *28*

2.6.8 Continuous integration *29*

2.6.9 Continuous deployment *31*

2.6.10 Iterations *32*

3 FIRST ITERATION - TIPPING OFFERS 34
3.1 SPECIFICATION 35
3.1.1 Functional Requirements 35
3.1.2 Non-functional requirements 36
3.1.3 Use Cases Diagram 41
3.1.4 Use cases specification 42
3.1.5 Conceptual data model 45
3.1.6 System Sequence diagrams 48
3.1.7 Operation contracts 51
3.2 DESIGN 56
3.2.1 Patterns 56
3.2.2 Normalized Database Diagram 59
4 SECOND ITERATION - ORDERS 61
4.1 SPECIFICATION 63
4.1.1 Functional Requirements 63
4.1.2 Non-functional requirements 64
4.1.3 Use Cases Diagram 65
4.1.4 Use cases specification 66
4.1.5 Conceptual data model 69
4.1.6 System Sequence Diagrams 71
4.1.7 Operation Contracts 73
4.2 DESIGN 77
4.2.1 Normalized Database Diagram 77
5 THIRD ITERATION - SUBSCRIPTIONS AND REDEMPTIONS 78
5.1 SPECIFICATION 79
5.1.1 Functional Requirements 79
5.1.2 Use Cases Diagram 80
5.1.3 Use Cases Specification 81
5.1.4 Conceptual Data Model 86
5.1.5 System Sequence Diagrams 88
5.1.6 Operation Contracts 90
5.2 DESIGN 94

5.2.1 Normalized Database Diagram *94*

6 IMPLEMENTATION 95

6.1 BACKEND 96

6.1.1 Ruby *96*

6.1.2 ..... *96*

6.1.3 Ruby on Rails *97*

6.1.4 Hosting Server *98*

6.1.5 PostgresSQL *98*

6.1.6 Rspec *98*

6.2 FRONTEND 99

6.2.1 HTML with HAML *99*

6.2.2 CSS with SASS *100*

6.2.3 JavaScript *101*

6.2.4 jQuery *101*

6.3 TESTING TECHNOLOGIES 102

7 PLANNING, MONITORING AND BUDGET 103

7.1 PLANNING DESCRIPTION 103

7.1.1 Iteration 1 – Tipping offers *103*

7.1.2 Iteration 2 - Orders *104*

7.1.3 Iteration 3 – Subscriptions and redemption strategies *104*

7.2 BUDGET STUDY 105

7.2.1 Hardware / Hosting *105*

7.2.2 Software *106*

7.2.3 Human Resources *106*

7.2.4 Total budget *108*

8 CONCLUSIONS 109

8.1 ACHIEVED GOALS 109

8.2 FUTURE EXTENSIONS 110

8.3 PERSONAL EVALUATION 111

9 BIBLIOGRAPHY 113
9.1 BOOKS 113
9.2 REFERENCES 113
10 GLOSSARY 115
11 ANNEX 118
11.1 DEMO 118
11.1.1 Tipping offer page 119
11.1.2 Payment page 120
11.1.3 Order administrator page 121
11.1.4 Administrator new tipping offer page 122

1 Preface

The reason I started this project came at the end of my exchange studies in Sweden in December of 2010. After probably the most interesting and revealing six months of my life, I had the realization that I wanted to get to know another part of the world.

After my time in Sweden I had a semester of courses and the final project left of my Superior Engineering degree in Computer Science. If looking for a job abroad was not challenging enough, I also had to find a company where I could develop my Final Degree Project.

However after a lot of research, effort, and a bit of luck, I found an opportunity I could not pass up. I applied to the "*Jovenes con Futuro*" program from the Spanish company *StepOne*. Out of pool of more than 500 Spanish students, and after several personal and competence interviews, I was one of the four chosen for this unique opportunity to work in a startup company in California. Shortly after receiving the news I found myself buying my one-way ticket to the heart of San Francisco to start this new adventure.

My main goal with this program was to get exposure to the startup technology industry in Silicon Valley as well as gain working experience. However, things are not always as good as they seem. The company I was hired for was sub-contracting all the product development in India. By the time I arrived the product development had already been started and there had not been a tech leader locally hired. Obviously, at one point everything became a heap of absurdities. The worst of it all was that after a few months of work, I realized I had hardly learned a single thing.

This was a major turning point for me and I decided things could not continue as they were. Instead of coming back home and start over, I decided to give this city another chance. In a brief period of time, I got hired by another company and extended my program for few months more.

This new job has offered me a young work environment and a close small team to work with. Working side by side with the Chief Technology Officer of the company is a great learning experience. The company's agile development methodology together with the use of the latest web technologies has also given me plenty of new knowledge.

Moreover, working in a shared space in San Francisco has turned out to be a great chance to meet interesting people and attend multiple technology events.

Overall I can conclude that despite starting 4 months later and at a different company than I had expected, I am confident to say that I am now working for a company that it is giving me a great experience, both academically and professionally. That company is *Rewardli*.

2 Introduction

Rewardli is a company that lets small and medium businesses obtain prices only available to big companies by providing a listing of deals and perks. The main type of offers in *Rewardli* until this project started were cash back offers. A *Rewardli* user can get a discount on many different merchants depending on his buying power. The discount that the business can get depends on the buying power. The buying power is calculated with a formula that takes into account the previous purchases and the connections this user has with other users in *Rewardli*.

Even though the cash back offers provide good discounts to the users, they resulted a bit difficult to understand for users and they did not generate the initial anticipated engagement.

Rewardli also offers a free service deal site for membership-based groups. This is a place where groups can post and share deals among their members. Examples of these membership-based groups are incubators like '500 Startups' and 'Techstars' or language support groups like "Columbus Ruby Group" or even companies like "Sprouter" or "Grasshoper". By registering in *Rewardli* they can create their own co-branded site and gather many kind of perks for their members there. These sites are great tools of engagement for groups. Members that sign up for the co-branded group can also benefit from *Rewardli*'s public deals.

When trying to figure out a new way for *Rewardli* to help small to medium businesses achieve better prices, the tipping offer concept was contrived.

A tipping offer is an offer that varies its price depending on the amount of people who have bought it. Each tipping offer expires at a specific time. Whatever the price is at that specific moment, will be the price that all the buyers will pay.

Rewardli can negotiate a better price for everybody as purchases are made for a group. Obviously, this kind of offer can work better on products that can sell at scale.

Some potential advantages shine for these kinds of offers. They might give users a reason to share the offer with people they know who would be interested in it. The reason is simple, by doing this; they will get a better price for themselves.

On the other hand, merchants can also be interested in offering tipping offers in *Rewardli*. They can get direct access to valuable potential customers and at the same time, release stock they have accumulated.

The following paper with detail the process followed to develop the tipping offers concept in the company.

2.1 The Company

Rewardli is a startup company based in the San Francisco, California. Co-founded by George Favvas and Jean-Sebastien Boulanger, it aims to allow small and medium businesses leverage the buying power in their groups so that they can get better deals on the products and services they need.

Both George and Jean shaped their business idea in Montréal where they are originally from. The company was seed-funded by Real Ventures, a venture capital and private equity partnership also from Montréal. *Rewardli* graduated from Dave McClure's 500 Startups accelerator in 2011 and is funded by a set of different groups of investors including 500 Startups, Real Ventures, Initio Group, Kima Ventures, and business angels. The company started its business publicly in September 2011, at the TechCrunch disrupt event in San Francisco, CA.

Small and medium businesses can sign up to the *Rewardli* platform to automatically benefit from all the perks it offers. Office material, software, accounting and payment services are some examples of the endless categories of perks that can be found and redeemed with only a few clicks.

The *Rewardli* website has also been popularly known for letting groups and organizations build their own groups to get exclusive deals within the website. Just by filling a form they can start adding their own private deals. This way all the members of the group will have place to easily find the deals the group offers. On the other hand, *Rewardli* tries to get discounts for them using its business connections.

Rewardli is actively trying to improve its service to its users by looking for new merchants and testing innovative approaches to the market. The tipping offers, which is the main focus of this project, is a clear example of a groundbreaking way of approaching groups of customers and merchants online.

2.2 The Team

There are currently four people working at the startup. Each one of them brings a different background and skills to complement the team. George and Jean met while working together for a different company and joined forces later to start this new business. Following is a short description of the background of the group.

George Favvas

Co-founder and Chief Executive Officer

George is an experienced product manager and business development executive. He previously worked at *TotalNet* and *RadialPoint*. He also founded two startups, *Reflexity* and *SmartHippo*.

George is the business side of the company. He manages most of the contact with the merchants.

Jean-Sebastien Boulanger

Co-founder and Chief Technology Officer

Jean is a software engineer with more than ten years of experience in architecture of web applications. He previously worked at White Label Dating and also co-founded *Swakes* and *Codapay* startups.

Jean is the one in charge of the technology side of the company. He developed the first version of the site all by himself.

Hamish Macpherson

Web designer and front-end developer

Hamish is web designer with more than five years of experience. He worked at *RadialPoint* where he first met George. He has worked as a freelance designer as well.

Hamish is the artist of the company. He also makes the users get the best of the possible experiences from interface of the site. Hamish works in the project from Montréal.

Pol Miró *(author)*

Future Software engineer

I am young software engineer from Barcelona. I went to San Francisco looking in research of exposure to the startup ecosystem in Silicon Valley. I was granted the chance thanks to *StepOne* and its program "Jovenes con futuro", of which I am very thankful.

I previously worked as a front-end developer at *Offerslot* during six months right after my lasts courses at Polytechnic University of Catalonia. During my studies I always enjoyed starting new projects and learning new things by myself.

2.3 Goals

Setting the goals of the project is an important task to be able to evaluate its success. The tipping offers have three clear goals hereby described.

2.3.1 Give access to wholesale offers for SMB.

Small and medium businesses do not have the buying power of big companies. It is hard or even impossible for them to achieve the prices that big companies can negotiate with suppliers. We think that tipping offers can be a path to help small and businesses benefit from them by grouping their purchases together.

2.3.2 Grow the user base

Rewardli wants to improve the current deals with better prices and reach out more merchants. The user base plays a leverage role when negotiating with the merchants. This is the reason why *Rewardli* is at this moment in time trying to increase the number of users in the platform. Tipping offers have a potential sharing factor by the users that might increase the traffic and bring newcomers to the site.

2.3.3 Provide merchants with a quick release of stocks

Accumulated stock is always an important expense for merchants. Tipping offers last for a short period of time and can accumulate a group of buyers for a product. This can produce a quick boost on the sales of the merchants. This way they can easily liberate accumulated stock.

2.4 Scope

The project scope has been difficult to bound because of the agile and incremental development methodology. Nevertheless I will try to scope which features belong to this project and which do not (even if they do belong to the company product).

The execution of the project started with me giving some proposals of the specification and design to Jean. After discussing about them and clearing the details, he validated them and I proceeded with the implementation of the different sections.

In *Rewardli*, the team works in a very collaborative way. It is usually said that team members in startups have to wear different hats every day. Therefore it is somehow difficult to draw a perfect line about the responsibilities in the project. Everybody has his main role but works on many different things at the same time.

Here is a distribution and list of the different parts of this project and my role in them.

2.4.1 Personal Responsibility

- The requirements analysis, specification and design of the project.
- The continuous integration and the continuous deployment methodologies.
- The design of the integration test suite.

2.4.2 Contributions

- Jean and I implemented the backend of project. The project here documented had to be integrated with the rest of the site features, for example the co-branded websites. Therefore we had to coordinate the efforts. Jean usually took care of the integration with pre-existing tasks (co-branded sites, groups integration, etc.). These sections have been purposely not included into the scope of this project.

- Jean and I developed the unit tests suite as well during development phases. The reason is that most of the time we approach the low level implementation with test-driven development.

2.4.3 Other

- George and Hamish did the mockups of the views of the different flows.
- Our amazing designer Hamish implemented the visual design.
- Jean chose most of the stack architecture of the project during the creation of the platform. However, the old and the new components will all be analyzed in the implementation part of this memory.

2.5 Actors of the system

Having defined the goals and the scope of the project it is time to clearly describe the actors that interact with the system.

2.5.1 Administrator

The administrator is responsible of managing the whole platform. He manages all kind of offers, groups, the different actors and their granted access to groups. He will also have to deal with the orders and provide support and help to the users.

2.5.2 Shopper

He is the main user of the system. The shopper has its own account in the platform. He wants to browse and share the different deals of *Rewardli* and take advantage of them. He wants to look for tipping offers as well as order them. This actor can also have these roles in the system.

Group member

A shopper can join private deal sites from different groups and become a group member. Besides being able to deal with the public offers, a group member can also browse and take advantage of the offers of the private group he belongs to.

Group admin

The group admin is in charge of managing the members and the private offers of the group of which he is manager. He is the responsible to grant and revoke access to his group. He can also show and hide offers to the group depending on what he thinks is going to be interesting for his group.

2.6 Methodology

The team follows a simplified methodology derived from SCRUM. SCRUM is very commonly used in startups. A flexible and fast methodology is key for the success of the project. The goal is to learn about what the user wants and what he does not. Perhaps a feature that initially seemed to be very important is not actually well received by the users community. There is sometimes no other way to know it than to actually try with a demonstrable product.

Scrum is based in sprints (iterations). A sprint is formed a set of tasks that need to be fully completed to achieve its goal. Usually sprints last from one week to a maximum of one month. In a sprint, one or more features of the product are completed and can usually be deployed to production. The requirements are completely frozen during the sprint. The product can only change requirements in between sprints.

The short iterations give continuous feedback from the users. This causes the product to change all the time depending on the user response. The development methodology embraces the new requirements and the changes in the requirements. Because of the small size of the team the methodology the methodology is followed more flexibility and not word by word. The following items sum up some of the most important habits and procedures we follow:

2.6.1 Stand-up meetings

Everyday at 10am a “stand-up” meeting is held. Every team component presents a quick review of the work that he has done the day before. Afterwards, the individual tells to the rest of the team the tasks he will be working on during that day.

Any blockers that may interfere with his work are mentioned and discussed so that they can be prevented or at least reduced to the minimum consequences. A blocker is an obstacle that stops a piece of work from being finished. Avoiding blockers lets the teamwork efficiently and in a synchronized manner.

Besides that, stand-up meetings are also a source of motivation to the team by giving an overall perception of how the project is moving forward. Usually stand-up meetings do not take more than 10 or 15 minutes.

2.6.2 User stories

The project was divided and organized in stories. A story consists of few lines of text describing a requirement from the point of view of the user experience. In order to provide a proper structure and a more detailed format to the specification of the project in the memory, the author decided to document the user stories in the format of use cases.

By dividing the project in stories, we ensure that the final goal is achieved from the point of view of the user’s experience. It also avoids having to finish the project as a whole before trying it out. The user stories also help to identify atomic features that can be deployed atomically.

Once the requirements of the story are defined, they are divided into small tasks that can be performed by a team member. This division allows the work to be parallelized among the team members.

Completing all the tasks for a story ensures all the requirements for the story have been met.

2.6.3 Task-board

A task-board is a history of the state of all the stories' tasks. The task-board lets the team visualize the current state of the project at a glance. Some of the features of the task-board include: knowing which tasks are being performed, knowing who is performing that specific task, previewing the upcoming tasks in the project and reviewing the completed stories.

There are many online task-board tools, all following a similar interface. In *Rewardli* we particularly use AgileZen for this project. In the following screenshot we can see that stories are mainly divided in three columns.

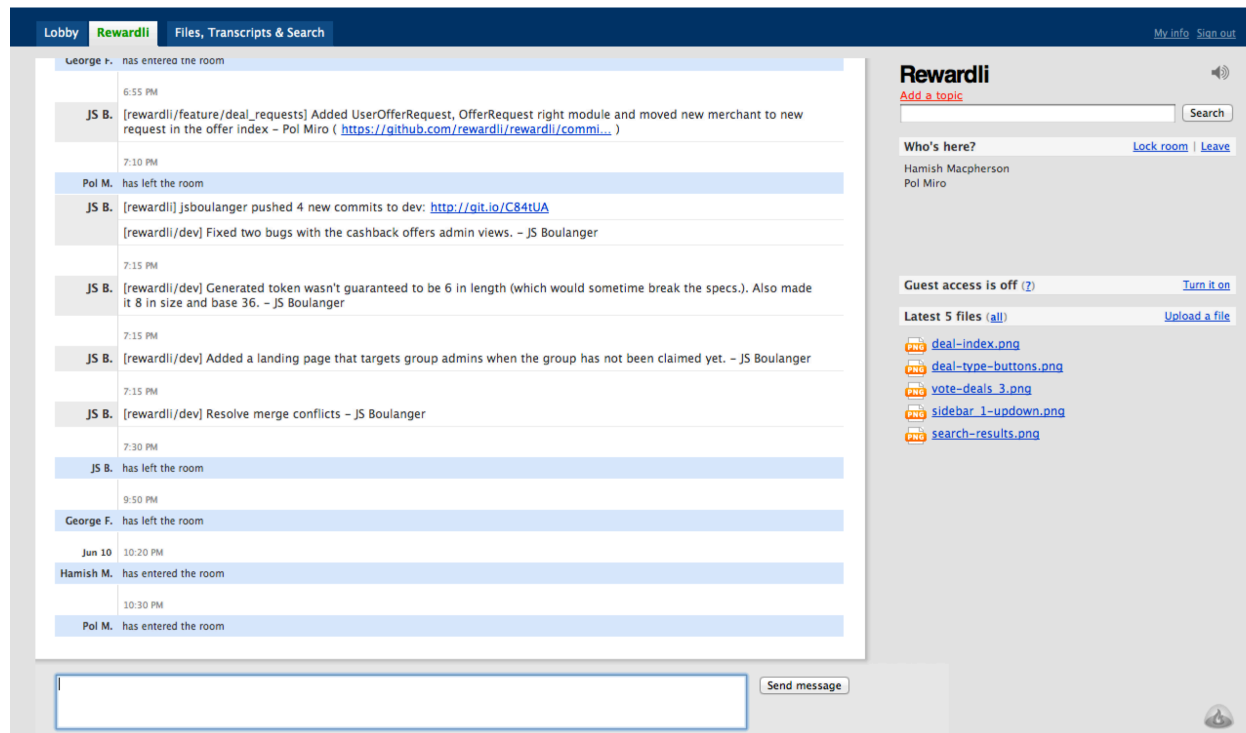
- Ready: These are the tasks that have no blockers to be initiated.
- Working: Displays the tasks that each team member is currently working on.
- Completed: Shows the finished stories.
- Archive/Backlog: Separate too new/too old stories from the main view.

2.6.4 Group sharing tools

Communication is vital to achieve speed in the project development. The easier the communication is the more efficient and effective the team will be. To solve simple blockers, as well as to improve long distance communication we used different sharing tools to communicate with each other.

One of the team members worked remotely from Canada; therefore it is even more obvious that we needed a good set of online communication tools.

We have specifically used Campfire, a group chat and sharing forum for the team. Campfire offers a group-chat as well as a place where any kind of files can be shared: documents, mocks, images, etc.



2.6.5 Version Control System

The version control of the project application is managed with Git. Git is a powerful distributed version control system. Each person in the team works in his own private repository in his machine. Whenever that team member wants to synchronize the work he has done, he can push the changes to the shared repository. Git takes care of most of the merging. Obviously, as most of the versioning tools, if Git cannot figure out how to merge two edited versions from different repositories, it will ask the user to do it.

- Git saves a full history of all the changes and development in the project. I had mostly work with SVN control version system during my degree and discovering Git has made working with a version control system much more comfortable. I would like to name the two main differences that have provided me with this experience:

- Git branching is much more easy and powerful than SVN branching. Git lets you switch branches very fast and also stores the complete history of each branch separately. By creating and merging new branches for each unit task the committed code history stays clean for future reviews.

- Git is distributed while SVN is centralized. Git still can have a central repository but it is intended to be distributed which means that there are multiple repositories in each of the machines and they do not have to exclusively merge in the central repository. As an example this gives a lot of flexibility to users easily work without having reached the central repository for days.

Of course, as many other version control systems, Git has branch capabilities. For this specific project, we followed a common pattern for organizing the code in branches. There were three main branches: master, staging and development.

- Development branch: This the branch over which the team develops. It has the latest changes, updates and features. It can also be “unstable” although for team-working purposes, we always try to keep it as stable as possible. Otherwise we can bother and step on each other.
- Staging branch: This branch is supposed to be the same as in the staging server. When a sprint is complete and the development branch is stable the changes are pushed into this branch. After that, usually all the changes are put into the staging server, which is the final testing environment before the production server. This way we have a better guarantee of the stability of the project and assure that no incompatibility issues will arise with the hosting server machines.
- Master branch: Just as the staging branch, the master branch holds the last version in production server. It is used to deploy to the production server.

The shared repository is privately hosted online at *Github*. The team can access and work from it from anywhere. It is a good practice to try to keep the local version updated to avoid conflicts in the future with the local repository.

2.6.6 Flow of work

The team uses a development tool called Git Flow. Git Flow is a branch strategy tool that extends Git using a subset of Git commands in a specific way. It helps keeping a clean and clear repository while developing new features, manage different versions, and apply hotfixes very quickly and in a clean way to production.

Each time a story/feature is started, it is developed in a new different branch. Once the story is finished and the code is tested stable, the branch is deleted and merged back to the development branch.

Git Flow also manages the versions of the code that are in production and staging. It can name the versions of the code each time they are deployed to the servers. In case anything unstable rises up, the last known working version of the code can be redeployed in a matter of minutes.

Finally Git Flow offers the option to create hot-fixes branches. When a bug rises up in production environment you can fix the bug right away with a hot-fix branch. This branch works directly on the master production code and after being submitted merges with it and also merges back to the development branch, so that the bug is fixed in both places at the same time.

2.6.7 Server administration

Although choosing hosting belongs to the design of the project I need to point out that *Heroku* has turned out to be a very useful tool in the project methodology. Managing a server can be a very hard and tedious task that requires a lot of time and usually a dedicated worker to supervise. *Heroku* kills that myth by offering a set of tools to manage your server in an extremely easy way.

Man could think that the simplicity of the tools mean very simple and not powerful servers, but this is far from reality. *Heroku* brings together a very powerful and flexible hosting service on the cloud as well as the support and tools to manage it with the most comfortable experience.

2.6.8 Continuous integration

As mentioned several times in this document, development speed is a very important variable. Continuous integration is a set of principles and processes that ensure the quality of the product as well as improve the speed in the current iteration.

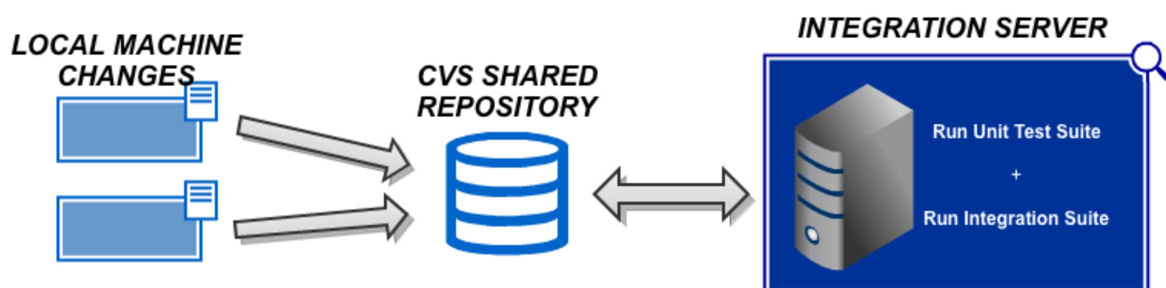
Instead of applying the quality control when the product is completely finished, continuous integration aims to make this control each time new steps are taken. This way quality assurance is done more often.

Continuous development also helps to reduce the amount of work that shows up when integrating changes of different individuals at the end of their tasks.

In a common development environment, the different parts developers do are integrated in the end. It becomes hard and tedious and many conflicts can arise. It sometimes requires all the team members to work in this integration tasks.

On the other side, with continuous integration, the task of integrating different parts all together is small and done multiple times. All the developers submit their changes to the shared repository nearly everyday.

A remote server is watching this repository, and anytime a changed is pushed, it runs all the unit and integration test suites. In case a test fails, the system automatically pings the developer to let him know that he has broken "the product".

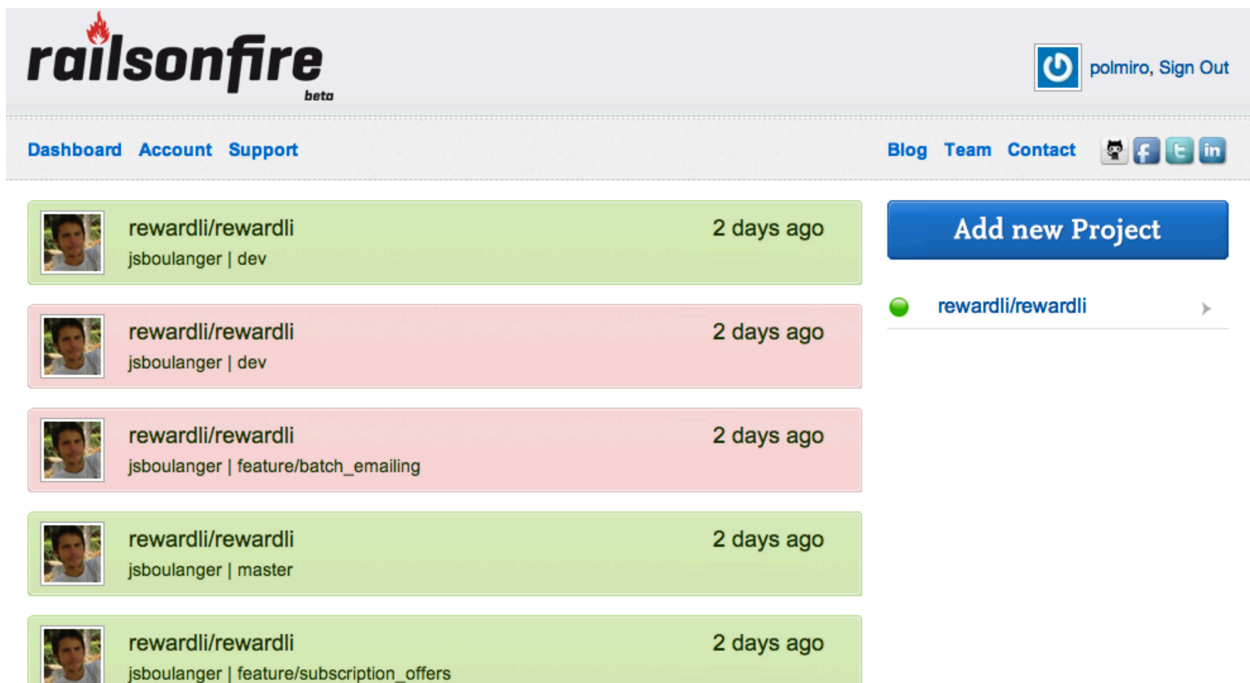


We did several times face many conflicts between different team members work. When a team member was working in a feature while another one was working in a different thing, some test of the project failed without anybody even noticing. This is the reason we decided to follow a continuous integration process.

After some research and having in mind we wanted a continuous integration structure at the lowest cost in terms of time and money, I decided to give a try to a brand new integration software called Railsonfire. This tool provides a super-easy and great integration with Github (our shared code repository) and Heroku (our hosting provider)

Railsonfire provided us with a server that watches our shared repository code. Anytime a change is pushed to this repository, Railsonfire starts running all the tests and lets us know when and whom broke the tests.

In the following screenshot we can see the platform already set up.



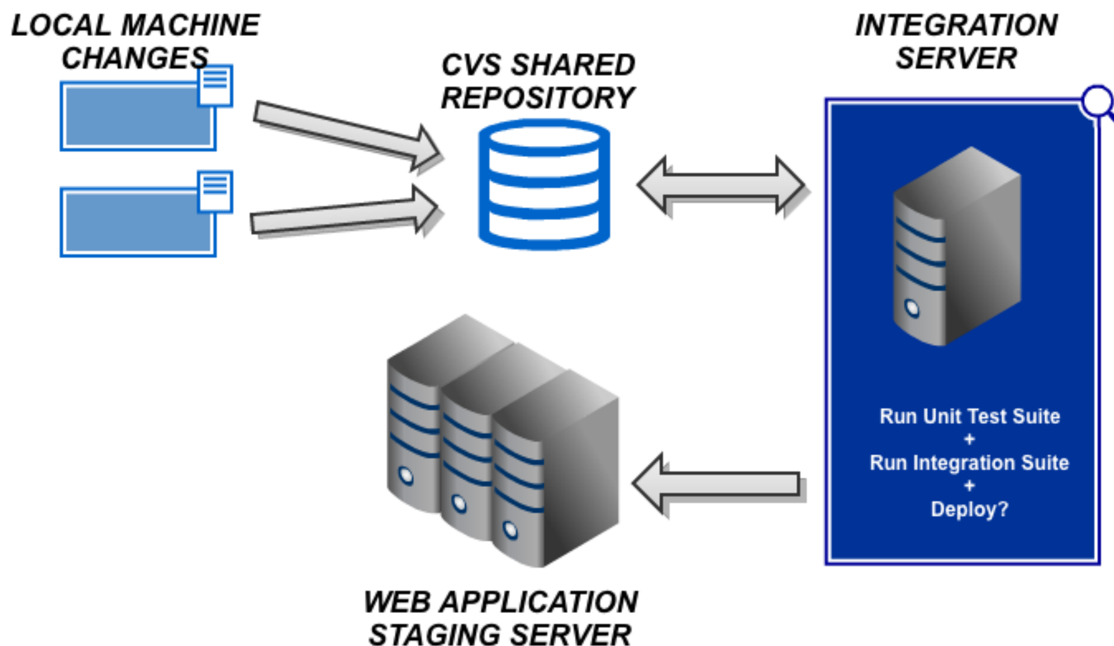
2.6.9 Continuous deployment

Continuous deployment is a methodology that goes one step further than continuous deployment. Not only the tests are run each time changes are pushed but they are also deployed to the servers.

At some point during the development of the project, we realized that we were pushing new changes and features to production several times a week. This became an important loss of development time for the developers.

We use continuous deployment for our staging server. This means that our staging server is synchronized with the last version of the application in the shared repository of the team.

We also use the Railsonfire service to configure the continuous deployment methodology. Only when the tests pass, and therefore the quality of the product is somehow assured, Railsonfire automatically deploys the version to the staging server.



2.6.10 Iterations

The project specification and design is divided into several iterations. The methodology behind this project is intended to be agile and embrace new requirements and changes within the product. The tipping offers idea evolved with different features and particularities with time.

There is a clear philosophy behind this iterative process. When creating a new product, you do not know how it is going to be understood or how well it is going to work. Thinking it behind the scenes too much is vain and useless, the real knowledge is there in the outside world.

By implementing the product in iterations we can analyze which features are well accepted and understood and which are not. We can get continuous feedback and shape the product according to it.

So the first thing when creating a product with this methodology is to try to get a product out and working the earliest the better. It does not matter if it does not initially have all the little details and requirements you want. Mainly for two reasons: first you might be wrong because your customers do not think the same, second because you do not have endless resources.

Of course you can try to get feedback before hand and implement your product in one shot. But how can you get a reliable feedback without even having a prototype?

Each one of the iterations, adds, removes or tweaks the existing requirements and has to produce a standalone and product unit that can be tried out there in its own.

This is not just the common process of splitting a project in several iterations. The product that presents at the end of each one has to be something by itself, not a part of a whole. We could think about it as a continuous prototyping.

Each one of the iterations in the documentation is divided in two parts, the specification and the design. Some parts analyzed in the first iteration apply to all the other

ones. For example, most of the non-functional requirements or the use of a specific data management analyzed in the first iteration will also apply to the next ones.

The memory tries to give a view of the whole project although the documentation does not describe all of the phases in full detail. The implementation phase has been separated after all the iterations because it has been analyzed in a general way independent from the iterations.

Finally I have to say I never realized how important the test phase (or testing in general) is for the development of a product during my engineering career. I had always underestimated it. And I am not only about referring to quality assurance, which is what always comes first to the mind when talking about testing.

Having a good test suite (in terms of integration tests and unit tests) is key for the efficiency of agile development. The outcome of the development team drops down a lot if the test suite is not maintained properly.

However, the test phase is not a phase as interesting as the specification and design for the purpose of this memory and therefore it will be described a bit in a higher level in comparison to the rest of the phases.

3 First Iteration - Tipping Offers

The first iteration of the project is focused in creating the basics to have tipping offers.

The tipping offers had to be integrated together with the existing kind of offers called “Cash back Offers”. That means that the system would support different kind of offers with different properties. First of all I will describe the particularities of any offer:

- An offer has a title, a headline, a description, some SEO keywords for the head of the page and an image URL
- An offer can be exclusive when it belongs to one or more groups, public otherwise.
- An exclusive offer can only be viewed by users that are members of one of the groups of the offer or administrators.
- An offer belongs to a merchant (who has many categories).

This iteration is trying to get tipping offers out there the simplest way possible. Therefore all tipping offers created in this iteration are public by default. This way we do not have to care about groups, group admins and group members, just shoppers in general.

These are the particularities that make tipping offers different from common offers.

- Tipping offers have some special conditions that the merchant provides. For example “only valid from July to September”. They also have some frequently asked questions (“FAQ”) that can be added to clear up any questions about the offer.
- Tipping offers are only available in a limited time.
- Tipping offers price is not static. It depends on the number of purchases that have been made to that tipping offer. The more people buy, the lower the price. This is represented in a price scale (to simplify the prices to the users). So for example when 10 people buy the offer, the price is 15$. When 25 have bought the offer the price becomes 10$. The buyer pays the current price and is refunded the different when the tipping offer is “closed”.

3.1 Specification

3.1.1 Functional Requirements

The shoppers will want to look for tipping offers. To do so, they might want to just to **look at all the tipping offers** in a straightforward way seeing the name, a little description about the tipping offer and perhaps the current price as well.

Sometimes they will already know what they are looking for. Therefore they will want to **find tipping offers for a specific product category**. Some examples are human resources, payment platforms, hosting and office material.

Once they find one they might be interested in, they will want to **look at the detailed information about the tipping offer**. They will want to read the terms and conditions, perhaps have a look at the frequently asked question.

On the other end, *Rewardli* will make agreements with merchants and will want to create tipping offers for their shoppers. The administrator is the responsible for **taking care of the tipping offers**.

The administrator has to be able to **register a tipping offer** in order to put it on sale. He will have to describe the different characteristics of the tipping offer. Some examples are the title, the description, when it is supposed to start and when it is supposed to end or the different prices it can have.

He will also have to be able to **change the characteristics of a tipping offer** as well in case there are some corrections to make or some new information to add to it. For example if the administrator receives a question repeatedly from multiple users, he might want to update the frequently asked questions information for that tipping offer.

In some cases the merchant might want to cancel an offer. That could be due to different reasons. Perhaps he did not like the results of the offer or ran out of stock. In this case the administrator will have to be able to **stop and remove a tipping offer**.

3.1.2 Non-functional requirements

After analyzing the required functionalities in the first iteration, we can proceed to analyze the non-functional requirements. Without them, the success of the project is not guaranteed. We have to take into account that a lot of the non-functional requirements are cross-related. Scalability depends on performance and efficiency e.g.: a website with a bad performance or efficiency is most likely not to be scalable; a website with a bad performance will not have a good usability. It is important not to lose track of any of them.

Usability

Merchants and buyers are the key users in the platform. Those users can be inexperienced with technology. We need to make an intuitive design. This will increase the chances of success of the users using the application. The most important points are:

- Learnability. The users have to be able to find offers the first time they use it, otherwise they might never come back. The interface has to be easy and simple to understand. A complicated interface will frustrate the user and blow up a possible sale.
- Efficiency. Once the users learn how to use the platform, they must be able to perform their goals in the fastest way possible. Each extra step increases the chance the user will leave the website, and once again, lose a possible sale. If it takes too long to find an offer the shoppers will just go away.
- Memorability. A successful user will remember the process finding an offer if it was easy to learn. This requirement will be achieved by succeeding both in learnability and efficiency.

Horizontal Scalable

Every startup is expected to scale and this will produce a huge increase in the website traffic. The system has to be ready for it; otherwise the users will have a slow unpleasant interaction that will cause a higher bounce rate.

The system has to be ready to support multiple users at the same time. The number of nodes will be continuously increased as traffic grows.

Modifiability

As a general rule of thumb, software is intended to be modularized and encapsulated so that it can be reused in the future. Further than that, startup products continuously evolve. The changes in the product are welcome and likely to happen in the future. This is the reason it is even more important than in any other kind of organization that the product is changeable.

Performance

We want to provide our users with a pleasant experience during their time on our site. Slow response time will harm the user experience. Furthermore, the site is expected to receive increasing traffic. Performance is a permanent requirement.

Testability

A solid base of tests for the application is basic to ensure the quality of the product. A good testing also helps modifiability by ensuring that new changes are not breaking previous behaviors.

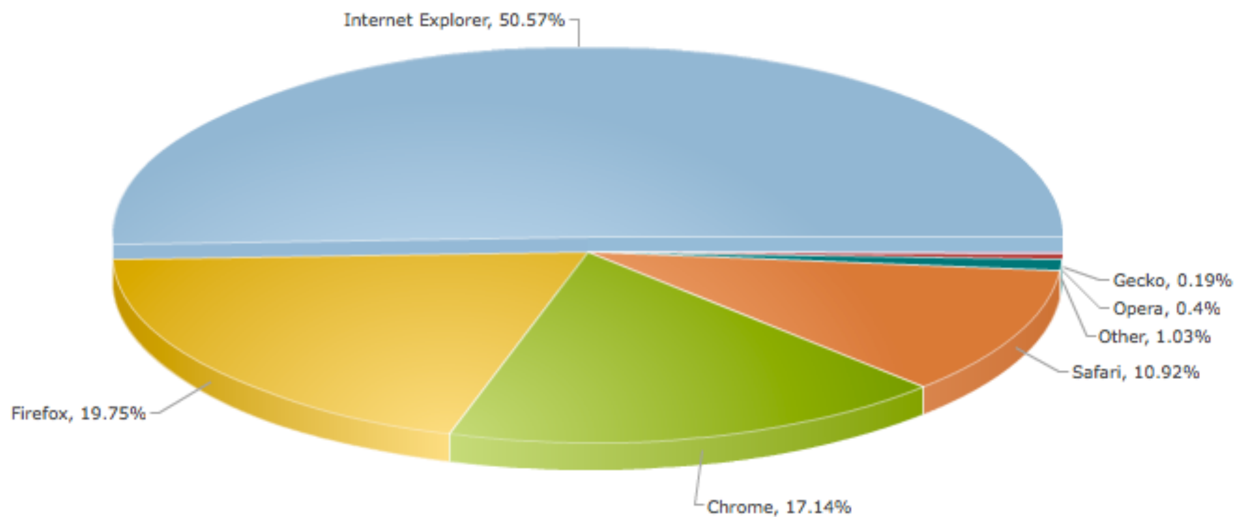
Cross-Browser compatibility

There are many different browsers, each one with much different compatibility. Ideally any person with access to a browser should able to browse and use the site

correctly. However, resources are not free and achieve a full compatibility is an expensive goal if not even impossible.

Very soon we realized it did not make any sense to try to achieve this goal and here are the reason why.

First thing to do is have a general overview of the market share of the browsers. Statowl is a reliable company that provides World Wide Web browser market share statistics. Here is a simple graph that describes the share of the browsers from November 2011 to April 2012.

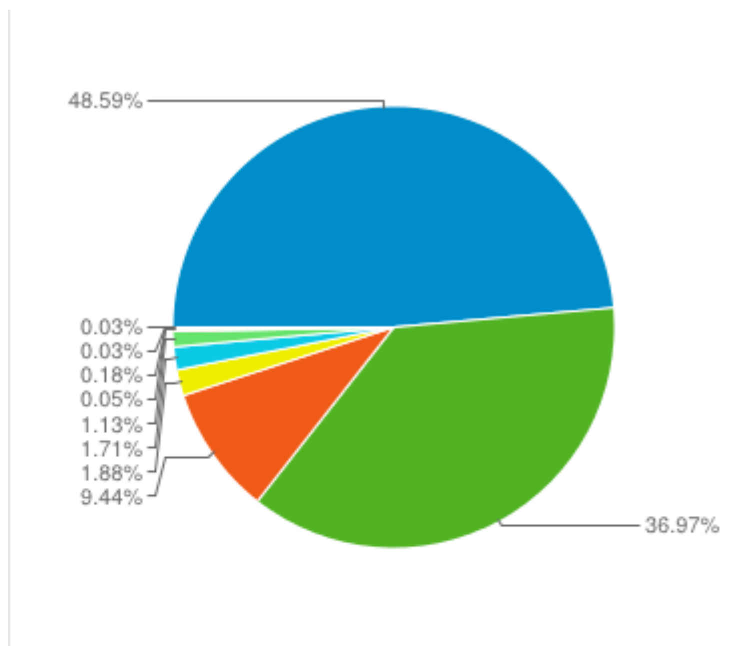


An individual can easily make the wrong decision by following this graphic. The most important part when choosing which browser to support is to study the profile of the users.

By looking at the profile of the users that had been landing to the site in the past we can take a meaningful decision.

Our account with the Google Analytics service provided us with the profile of the browsers our users had installed in their machines. Our customers are mostly in the bay area and have a good experience with technologies and most of them are Apple users. Here is a summary of the statistics extracted:

	Browser	Visits	Visits
1.	Chrome	1,935	48.59%
2.	Safari	1,472	36.97%
3.	Firefox	376	9.44%
4.	Internet Explorer	75	1.88%
5.	Mozilla Compatible Agent	68	1.71%
6.	Android Browser	45	1.13%
7.	Opera	7	0.18%
8.	RockMelt	2	0.05%
9.	IE with Chrome Frame	1	0.03%
10.	Opera Mini	1	0.03%



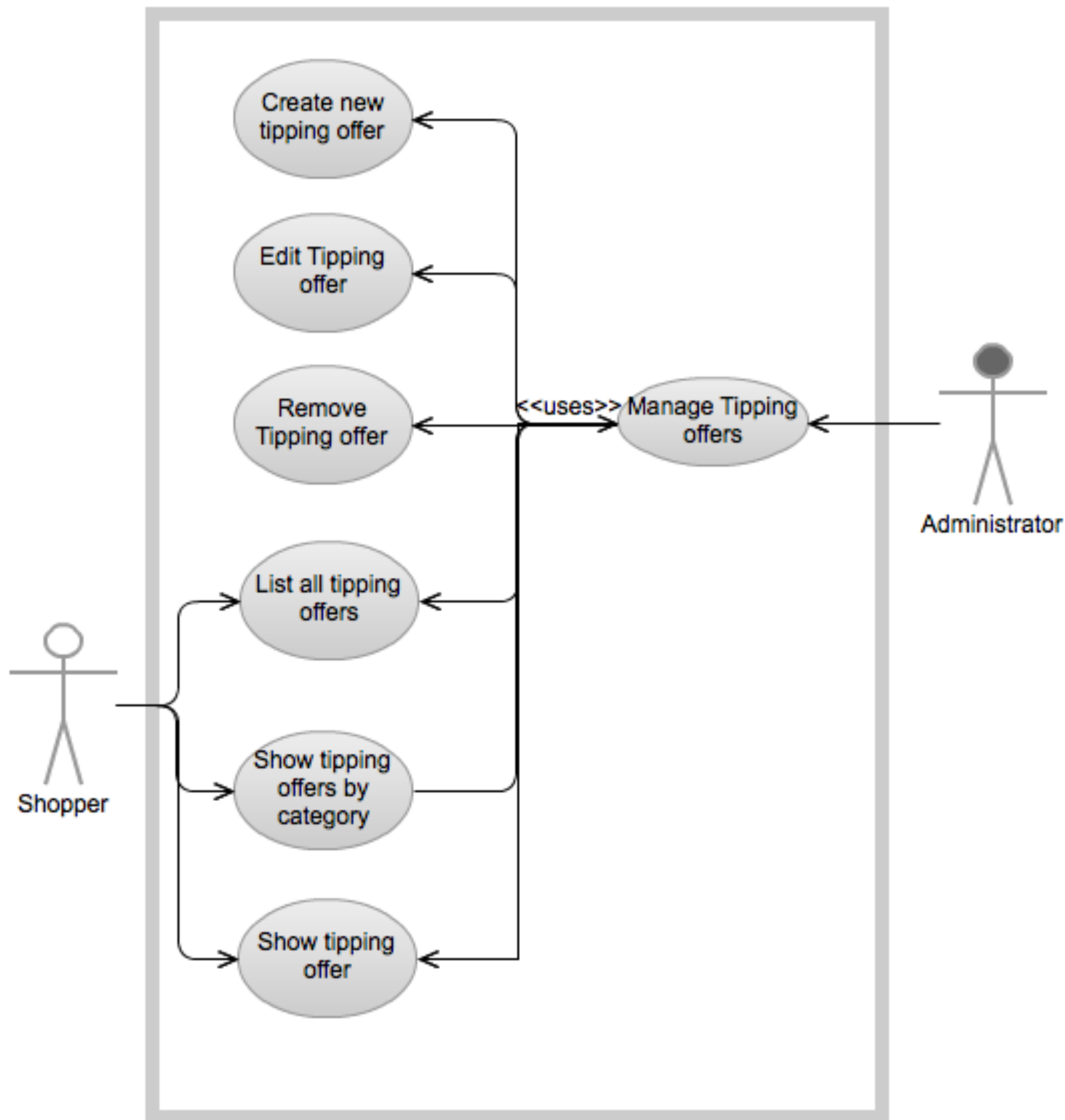
We can easily point out that only a 2% of the visitors are actually using Internet Explorer. A stroke of luck! Internet Explorer is by consensus, the most difficult browser to support because it does not follow many of the W3C standards. Google analytics also allows doing a more specific breakdown with the versions of the browsers as well, an 85% of the users with Internet Explorer browser were using a version equal or above 9 which is quite new.

With all this information in mind, and knowing how difficult it is to achieve compatibility with each versions of the browsers, this is the final list *Rewardli* fully support:

- Mozilla Firefox 3.6+
- IE9+
- Chrome 6+
- Safari 5+
- Opera 8+

3.1.3 Use Cases Diagram

The following diagram represents the use cases found after analyzing the requirements.



3.1.4 Use cases specification

The next section will specify in detail the use cases that have appeared in the requirements analysis. Each one of them will describe in a table format the actors, the goal, and a brief summary about the interaction and the typical course of events of the use case. In case that there are alternative events they will be shown after the typical course of events and reference at which moment they happen.

New Tipping Offer

Use case:	New tipping offer
Actors:	Administrator
Goal:	Create a new tipping offer
Summary:	The administrator wants to create a new tipping offer for a specific interval of time.
Typical course of events:	1. The use case starts when administrator requests to create a new tipping offer. 2. The system shows the different options to create a tipping offer to the administrator. 3. The administrator describes the tipping offer. 4. The administrator enters the different price levels and completes the creation. 4. The system records the information about the tipping offer and confirms it to the administrator.
Alternative course:	4. The description or times given by the administrator are not valid. The system does record the tipping offer. It asks the administrator to try again.

Remove Tipping Offer

Use case:	Remove tipping offer
Actors:	Administrator
Goal:	Remove an existing tipping offer from the system.
Summary:	The administrator removes an existing tipping offer.
Typical course of events:	1. The administrator requests to remove a specific offer of the system. 2. The system removes the offer and confirms the action to the administrator.

Edit Tipping Offer

Use case:	Edit tipping offer
Actors:	Administrator
Goal:	Change the information or interval of time of an existing tipping offer
Summary:	The administrator wants to change some of the information about an existing tipping offer
Typical course of events:	1. The administrator requests the system to edit a specific offer. 2. The system shows the existing information about the offer. 3. The administrator changes some of the information or the time the offer is scheduled. 4. The system confirms the changes of tipping offer and reschedules the tipping offer if necessary.
Alternative course:	4.When the administrator submits invalid changes for the tipping offer, the system does not record the changes. It also warns the administrator about it and requests him to try again.

Show All Tipping Offers

Use case:	Show all tipping offers
Actors:	Administrator, Shopper
Goal:	Show the information of all the offers
Summary:	A list with the basic information of the tipping offers in the system that is shown to the actor.
Typical course of events:	1. The actor requests the system to view a list of the existing tipping offers. 2. The system shows the basic information of each one of the offers in the system.

Show Tipping Offers By Category

Use case:	Show tipping offers by category
Actors:	Shopper
Goal:	Show a list of basic information of the tipping offers of the system of a specific category to the shopper.
Summary:	A list of the tipping offers of a specific category is shown to the shopper.
Typical course of events:	1. The shopper requests the categories of tipping offers to the system. 2. The system shows the different categories existing in the system to the shopper. 3. The shopper chooses one of the categories 4. The system shows a list wit the basic information about each one of the tipping offers that belong to a merchant that has the specified category and shows them tot he hopper.

Show Tipping Offer

Use case:	Show tipping offer
Actors:	Administrator, Shopper
Goal:	Show the detailed information of a tipping offer
Summary:	Detailed information of to a specific tipping offer is shown.
Typical course of events:	1. The actor requests the information of a specific tipping offer. 2. The system shows all the detailed information related to the tipping offer.

3.1.5 Conceptual data model

The conceptual data model represents the data related to the current iteration. Taking into account the previous existence of a type of offer called "cash back offers" previously described, this is the refactored conceptual data model. There might also be some entities not directly related to it, which help to put the diagram in context with the existing developed platform.

The following are the definitions of the different entities that appear in the conceptual model along with a brief description:

- **Merchant.** Merchant describes a business that sells a product in *Rewardli*. They have a name, a description of their business and a logo URL. An example would be the merchant with the name "BestBuy", has a description "Computer, phone and technology retailor".

- **Category.** The categories provide a general classification for merchants. A merchant can belong to many categories. For example "BestBuy" could belong to the categories "Computers", "Communications" as well as "Entertainment".

- **User.** This entity describes the actors of the system. Users have a name and a contact email. Users can have different roles. They can be administrators of the system or common shopper or group administrator. Those roles are described in the users of the system section.

- **Group** *(contextual)***.** A group describes an organization that has access to exclusive deals in *Rewardli*. Groups can be managed by a group admin users and system administrator users.

- **Offer.** An offer is the abstract entity that holds the information that any kind of offer can have. Specifically all offers have a name, a description and some other static information that describes them. As an example, a Best Buy offer could be: “20% discount in computer purchases”.

- **Cash back Offer** *(contextual)***.** Cash back offers are the main and unique kind of offer in *Rewardli* before this project. Particularly, cash back offers give a discount percentage to apply to some kind of purchases for a particular merchant.

- **Tipping Offer.** Tipping offers have frequently asked questions that clears up all doubts a shopper could potentially have. It also has some terms and conditions that apply to the specific offer. Tipping offers have a scale of prices that is defined by tipping offer levels. Tipping offers also have a specific interval of lifetime that is defined by a start date and an end date. Depending on the current time, it can have three different statuses:

- **Pending Offer.** The offer has not yet started.

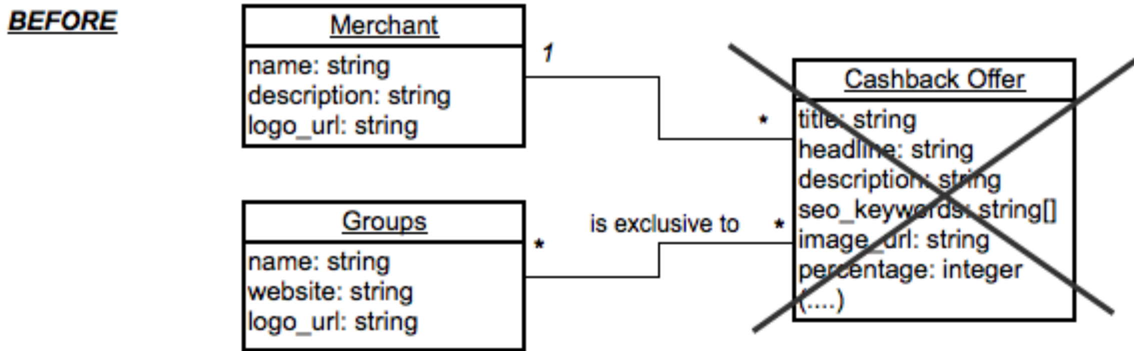
- **Active Offer.** The offer is started and purchases are open to shoppers.

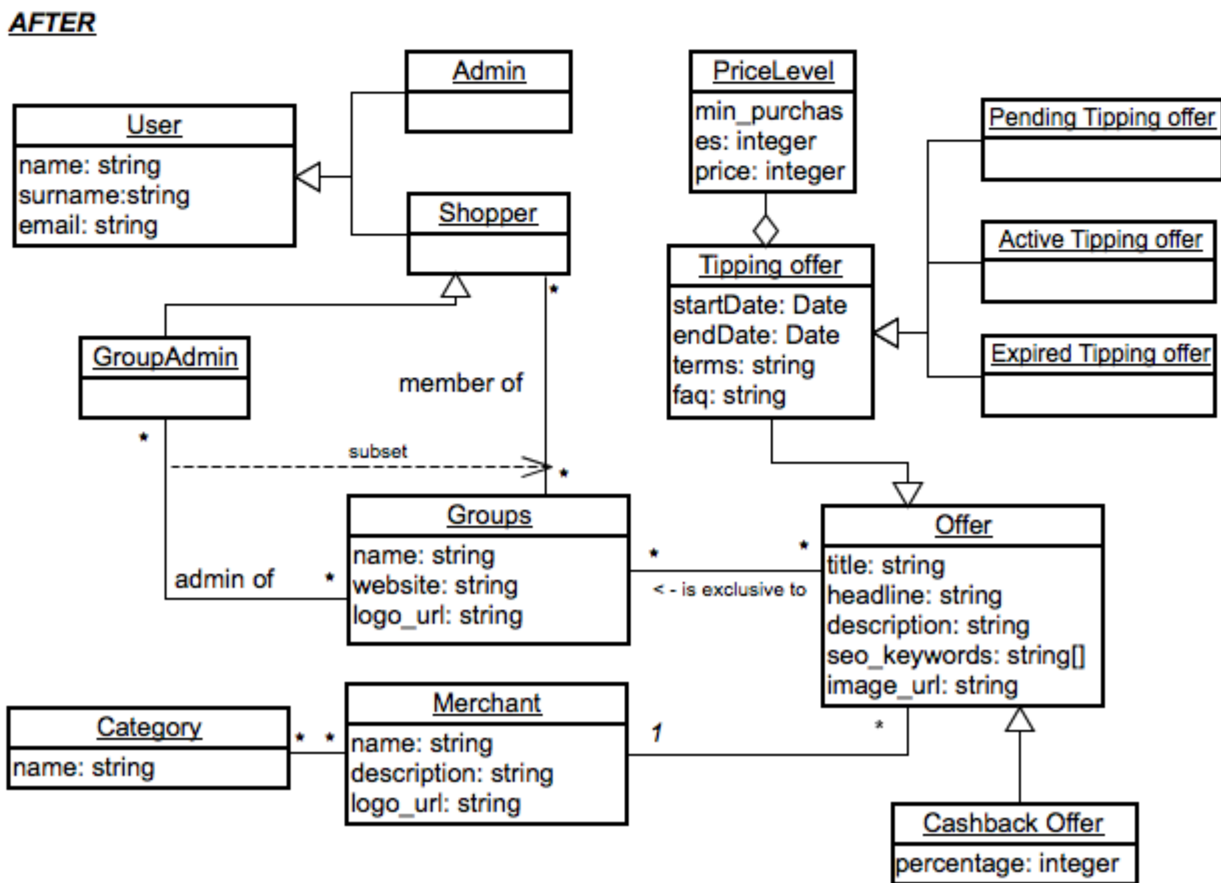
- **Expired Offer.** The offer is closed and no more purchases are allowed.

- **Price Level.** The price level represents the cost a tipping offer in relation with the purchases that have been made. They are defined by a minimum of purchases and a

price. The current price of a tipping offer corresponds to the price level with the minimum purchases surpassed. This is an example of price levels a tipping offer could have:

PRICE	MINIMUM PURCHASES
$100	**0**
$950	**5**
$900	**20**
$800	**50**

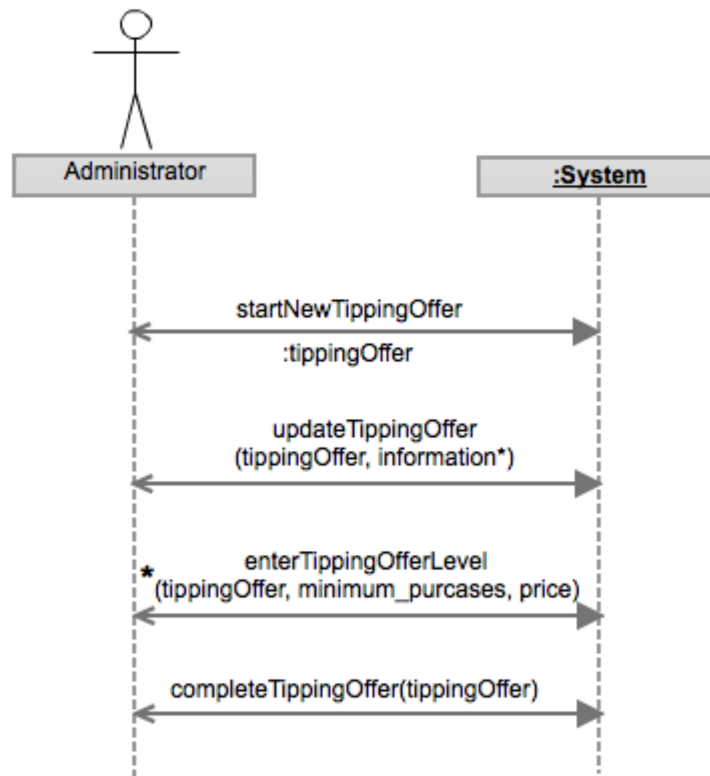




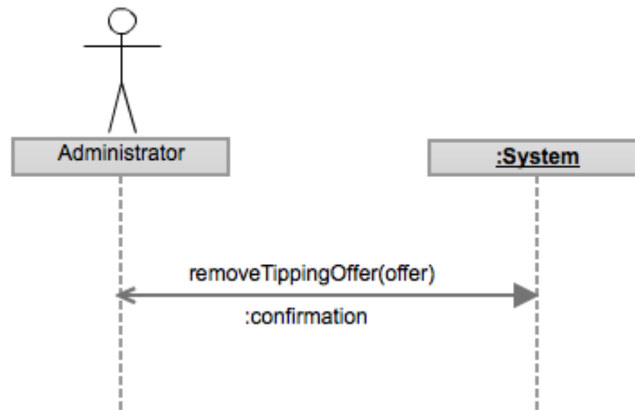
3.1.6 System Sequence diagrams

The next section shows for each of the essential use cases the system sequence diagram that represents the different interactions between the user and the system.

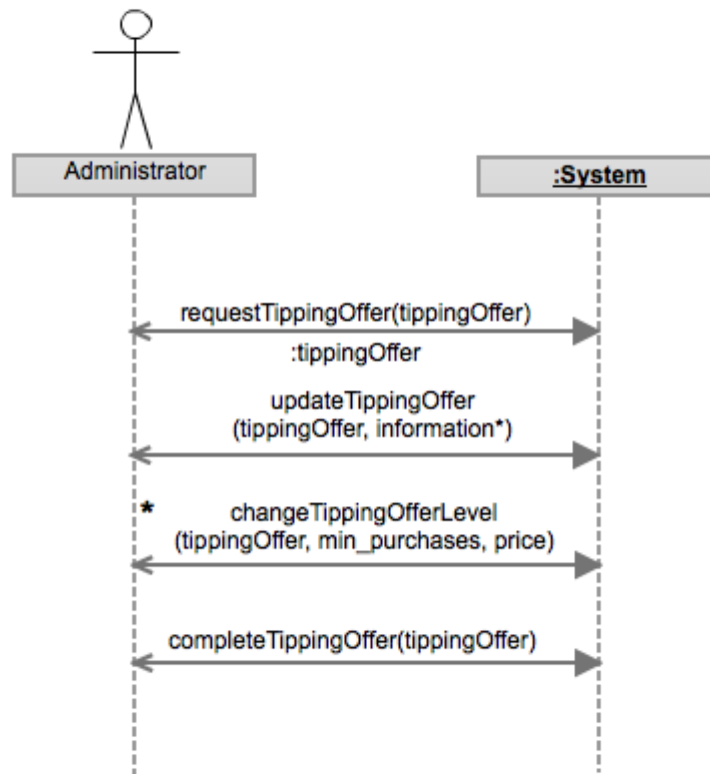
New Tipping Offer



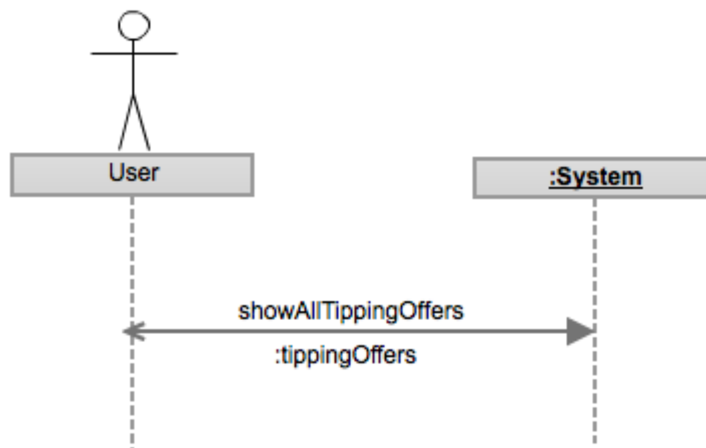
Remove Tipping Offer



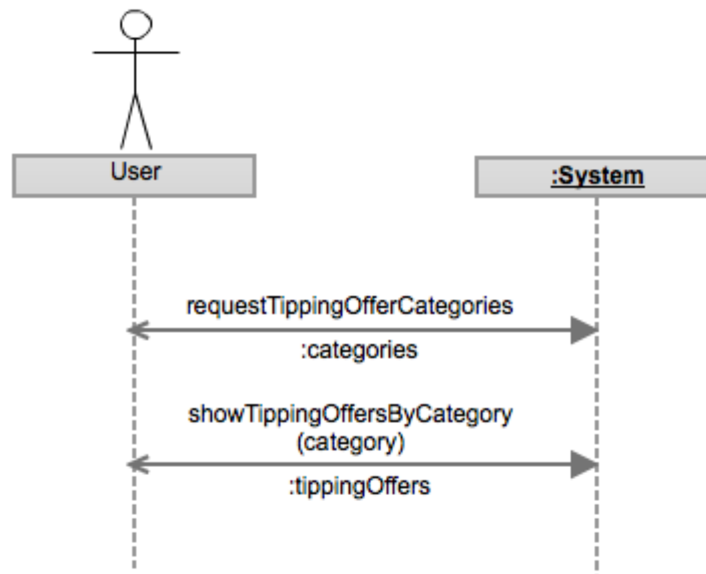
Edit Tipping Offer



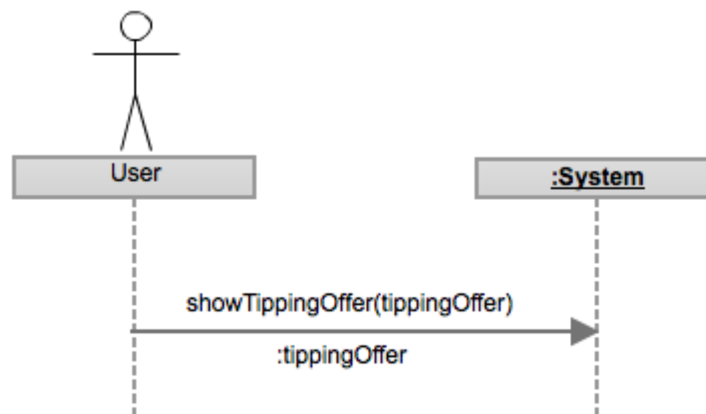
Show All Tipping Offers



Show Tipping Offers By Category



Show Tipping Offer



3.1.7 Operation contracts

The operations that have appeared from the system sequence diagrams are here grouped as operations of the system. Following this diagram that represents them. The operations that have been analyzed in detail have been highlighted in bold style. The two

not highlighted have been disregarded due to their similarity with one of the other operations.

SYSTEM
startNewTippingOffer() :tippingOffer
updateTippingOffer(tippingOffer, information*)
enterTippingOfferLevel(tippingOffer, price, minimum_purchases)
completeTippingOffer(tippingOffer)
requestTippingOffer(tippingOffer) :tippingOffer
updateTippingOfferLevel(tippingOffer, level)
removeTippingOffer(offer) :confirmation
showAllTippingOffers() :tippingOffers
requestTippingOfferCategories(category) :tippingOffers
showTippingOffersByCategory(category) :tippingOffers
showTippingOffer(offer) :tippingOffer

Name: **startNewTippingOffer() :tippingOffeer**

Responsibilities: start the creation of a new tipping offer

Exceptions: ---

Preconditions: ---

Postconditions: A new TippingOffer instance ‘t’ is created

Output: t

Name: **updateTippingOffer(tippingOffer, information*)**

Responsibilities: save information to a tipping offer

Exceptions: if the title in information* is empty show error

Preconditions: information is a set of information of the offer (title, description...) and the title is present

Postcondicions: tippingOffer has the attributes in information*

Name: **enterTippingOfferLevel(tippingOffer, min_purchases, price)**

Responsibilities: associate a level to a tipping offer

Exceptions: if a level exists with the same minimum number of purchases, shown an error

Preconditions: there is no level in the tippingOffer with the same minimum number of purchases

Postcondicions: register an instance of PriceLevel p with min_purchases = minimum_purchases and price = price associated to tippingOffer

Name: **completeTippingOffer(tippingOffer)**

Responsibilities: Finish the creation of the tipping offer

Exceptions: ---

Preconditions: ---

Postconditions: the tippingOffer has been registered in the system

Name: **removeTippingOffer(offer)**

Responsibilities: remove the tipping offer record

Exceptions: if offer does not exist show error

Preconditions: offer exists

Postconditions: offer does not exist anymore in the system

Name: **showAllTippingOffers() :tippingOffers**

Responsibilities: List all tipping offers

Exceptions: ---

Preconditions: ---

Postconditions: 'l' is a list of all the tipping offers with their basic information (title, headline, description, startDate and endDate)

Output l

Name: **requestTippingOfferCategories() :categories**

Responsibilities: list all existing categories

Exceptions: ---

Preconditions: ---

Postconditions: 'c' is a list of all categories that exist in the system

Output c

Name: **showTippingOffersByCategory(category) :tippingOffers**

Responsibilities: list all tipping offers whose merchant belongs to a category

Exceptions: if category does not exist show error

Preconditions: category exists

Postconditions: 'l' is a list of the tipping offers that exist whose associated merchant belongs to the category 'category'

Output l

Name: **showTippingOffer(offer) :categories**

Responsibilities: Give all the information of a tipping offer

Exceptions: if offer does not exist show error

Preconditions: offer does not exist

Postconditions: 't' is an TippingOffer instance of offer

Output t

3.2 Design

The fast pace in the development and the continuous changes of the requirements makes the design very changeable with each one of the iterations. Therefore it is not one of the main interests of this project to get into a lower design level of it as it would only represent a short-term picture of the current status.

The main general architecture patterns used will be described in detail as well as the database normalization process for each one of the iterations. The database normalization is one of the most important processes because all the design and implementation will work around it and it is one of the most permanent things in the design.

3.2.1 Patterns

The design of this project uses many design patterns to solve common problems in web development. The following part of the memory does not intend to reflect and explain all of the patterns used but at least mention and describe the most important ones.

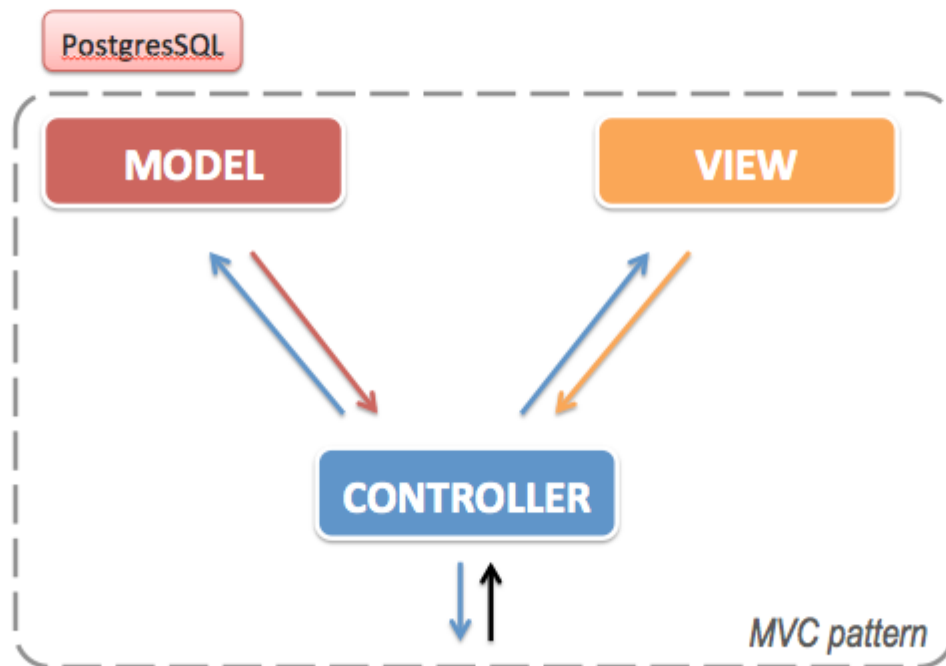
Model - View - Controller

This pattern is very common in web development design. A short summary of this pattern, it divides the application in three different parts:

- Models. Their responsibility is to save the data state of the application. They are based around the data and not around behavior.
- Views. As the name suggests they are responsible to show the data in different ways.
- Controllers. They receive the user inputs and use the models to generate the output and delegate the data to the views.

By following this paradigm, we ensure that a certain level of modularity, flexibility and encapsulation that helps with the continuous evolution of the site.

Let’s say that in a closer future, the site will have to support a visual design for mobile devices. We could easily support that without having to touch a line of code from the models or the controllers, we would just have to manipulate the views.



The same scenario takes place for example when a new user with different permissions has to be able to manage the site. Just by manipulating the model (adding the corresponding data and domain logic data of the model) and the controllers (which would ensure the permissions are correct) we would easily be able to integrate this new requirement.

The level of complexity that brings to the system is justified by the amount of advantages that we get. Furthermore, by using a framework like Ruby on Rails, the MVC architecture is already prepared and ready to be worked on.

Active Record

Active record specifies a way to store the objects in memory in the relational database of the application. In Active record, each class or entity that exists in the normalized is associated with a specific table in the database. Each one of the object instances is mapped in a row of the table of its class.

Furthermore, each one of the objects has its own storage methods to insert, update or remove himself from the database. The main advantage of active record is that it is a very simple way to manage data persistence. It is implemented very easily and it is quite flexible.

There are several issues to take into account when implementing the Active record pattern. The objects are aware of their own persistence that ideally should be an independent responsibility and separated from the object.

Another alternative that was taken into account was implementing a Data mapper pattern for data persistence in the SQL database. This pattern separates completely the storage from the original object, which guarantees a level of independence from the data storage.

The former was chosen basically because of its simplicity. A Data mapper pattern would have been an over-engineered solution that would add another layer of complexity to the application.

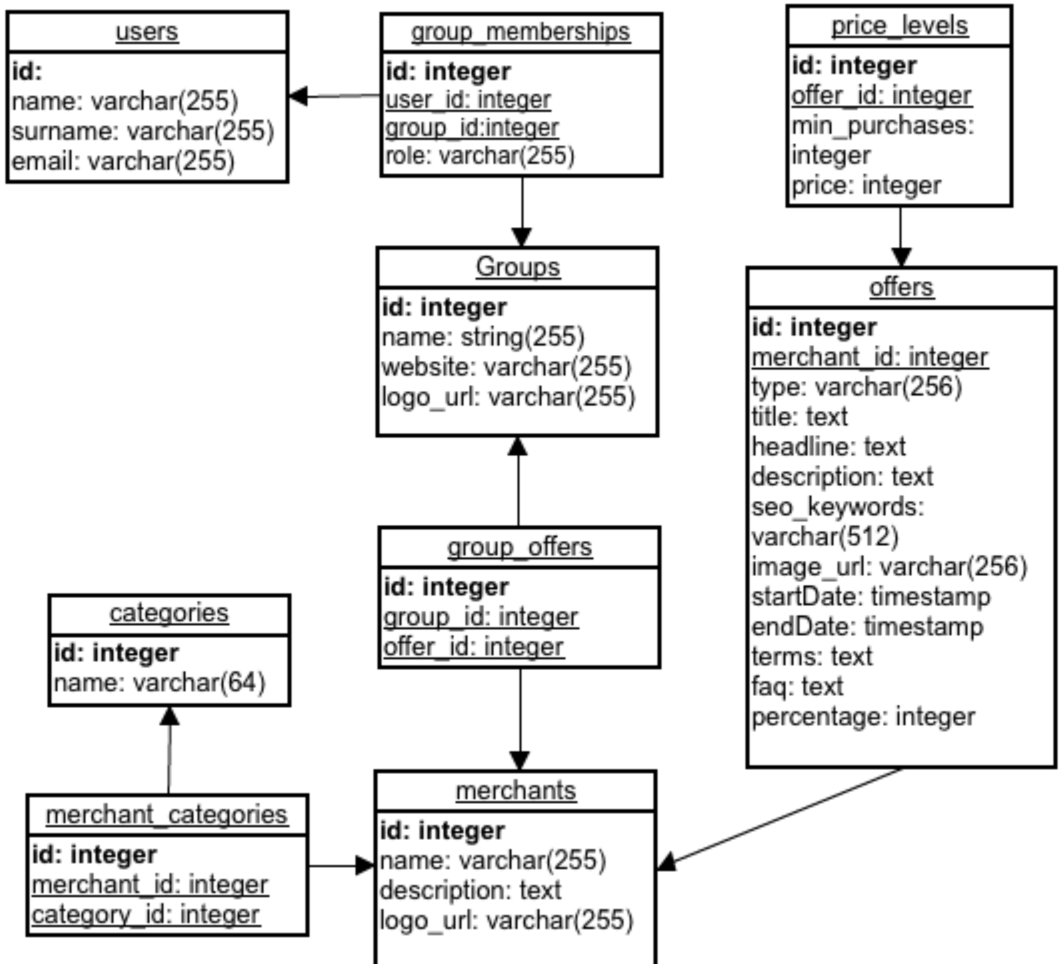
The team usually had a good knowledge of the issues behind the Active record pattern. Following a set of 'best practices' when using the Active record was enough to end up with a good persistence design.

Active record is implemented in the Rails framework, which provides a superclass where the developer can make the inheritance from: ActiveRecord::Base. When a model created inherits from this class and the table of the class is properly defined in the database following the convention, rails takes care of all the SQL logic applying the Active record

3.2.2 Normalized Database Diagram

The normalization of the database has been done according to the utilization of the Ruby on Rails framework. As it is explained in the Implementation chapter of this memory, Ruby on Rails favors convention over configuration. Some of the procedures followed to normalize the database have been the following:

- **One table for each model:** Active Record pattern together with the rails convention derives into creating one table for each of the models with the name pluralized.
- **Single Table Inheritance:** The users and their different roles as well as the different type of offers are set into one single table in the database schema. This leads to a bit of loss of space because of unused fields for each object but brings a lot of simplicity and integrates perfectly in the Ruby on Rails framework
- **Id as primary key:** Using an id for each of the tables lets Rails manage the records with more efficiency.
- **Alternative primary keys:** To ensure the data consistency and robustness we can apply those indexes to the alternative primary keys.


users
id:
name: varchar(255)
surname: varchar(255)
email: varchar(255)
group_memberships
id: integer
user_id: integer
group_id:integer
role: varchar(255)
price_levels
id: integer
offer_id: integer
min_purchases:
integer
price: integer
Groups
id: integer
name: string(255)
website: varchar(255)
logo_url: varchar(255)
offers
id: integer
merchant_id: integer
type: varchar(256)
title: text
headline: text
description: text
seo_keywords:
varchar(512)
image_url: varchar(256)
startDate: timestamp
endDate: timestamp
terms: text
faq: text
percentage: integer
group_offers
id: integer
group_id: integer
offer_id: integer
categories
id: integer
name: varchar(64)
merchant_categories
id: integer
merchant_id: integer
category_id: integer
merchants
id: integer
name: varchar(255)
description: text
logo_url: varchar(255)

4 Second Iteration - Orders

After having developed the base to support tipping offers, we analyzed the user response. We had a few successful tipping offers and we some users gave us really good feedback about it. It was time to take a next step.

The management of the orders was not practical at all. As it was mentioned at the beginning of the first iteration, the shoppers used the system only in an informational way. The orders of the items were done manually contacting the administrator. They proceeded with a payment and received a redemption code. Then the administrator had to manually update the price in case it had to be changed. This was really tedious and not scalable. Therefore it became a priority.

Moreover some of the providers asked for a total maximum number of purchases that could be sold. That was because they did not want to sell more too many if the deal was very good. It could affect their gains too much. Another similar issue was that as the tipping offer where sometimes used as special one time deals. The providers wanted to limit the deals on a per shopper basis.

If we were going to manage the orders automatically in the system, it was a good moment as well to embrace those two new requirements about the maximum total purchases and the maximum purchases per shopper.

We also would have to take care of the redemption method. We should save some redemption information provided by the merchant for each tipping offer (instructions about how to redeem the purchases basically) and also provide shopper with a redemption code. We decided to start with a very simple redeeming method. We would generate random redemption codes (one for each purchased offer) and send them both to the merchant and to the shopper so that the merchant could verify the redemption.

In conclusion, this iteration would free the administrator from being responsible of managing the orders, the payments, the purchases limits and simplify a bit the redemption of the tipping offers.

4.1 Specification

4.1.1 Functional Requirements

The shoppers will have to be able to **order an amount of tipping offer**. Let's say they want a product that is sold as a tipping offer that is called "a user tests at usertesting.com" for 10$. The shopper might want to get 3 of them at the same time.

The shopper will pay the current price for the tipping offer through credit card. Once the purchase is paid, the shopper will be provided with a redemption code. Then he will be able to contact the seller merchant and redeem the offer by telling the merchant his redemption code. Of course the merchant will need to have a list of the redemption codes in order to verify its authenticity.

The shopper might want to **check the history of all his purchases** made in the past. Perhaps to look for the corresponding redemption code or just to double check the price paid.

Once the tipping offer finishes, the administrator will have to **refund the tipped price** of each purchase if it is the case.

He will also have to take care of unexpected events with the orders and attend possible enquires from the shoppers. Therefore he will need to **see all the orders that shoppers** have made and **take care of possible cancellations**. In case of a cancellation the price paid will be refunded but the count of the purchases for that tipping offer will not be changed. This could increase the complexity and normal behavior of a tipping offer (prices could go up!).

4.1.2 Non-functional requirements

In addition to the more general non-functional requirements we will need to take into account two new ones that come from managing the orders through the system. This process has to be secured and a proper customer service to support the users has to be provided.

These are the mentioned requirements and their specification:

Secure

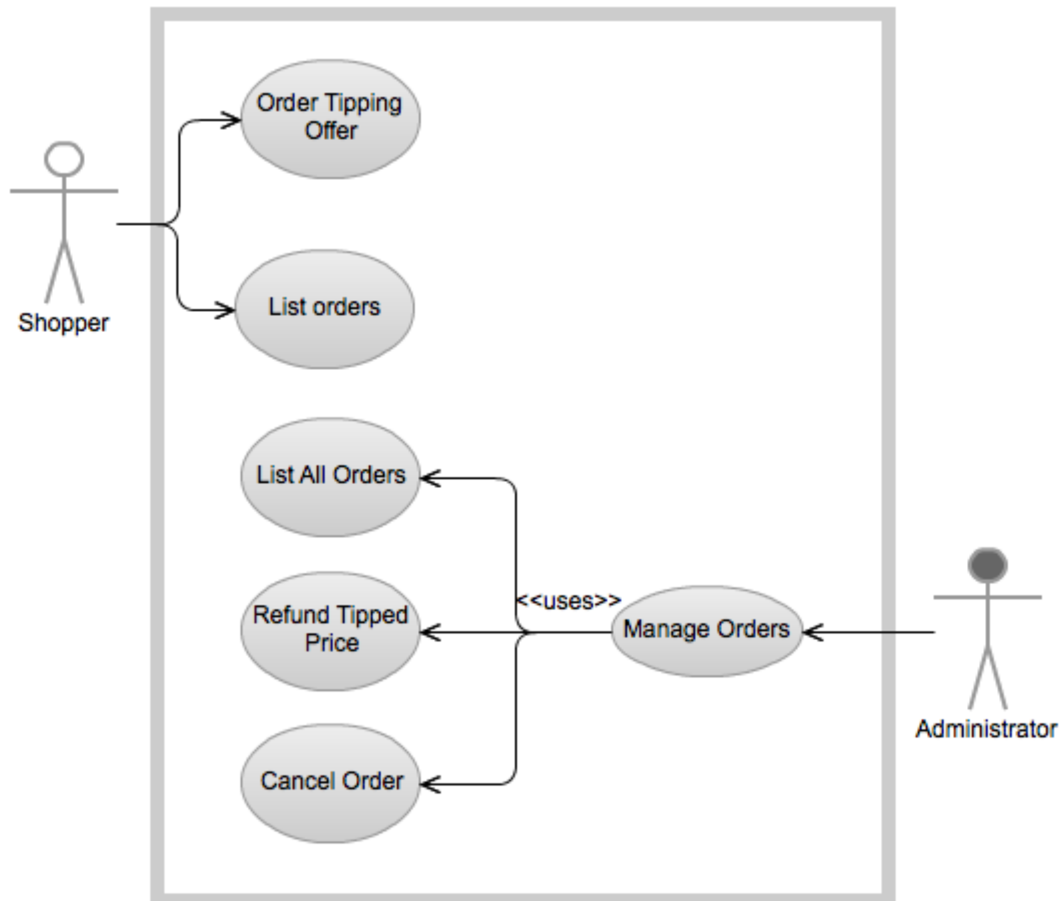
This requirement is specific to the only some of the use cases, for example the ones involving payments. We need to assure that the payment information the shopper is providing is safe during the transaction and after it has happened in case it is saved in the platform. The communications and the data between the system and the shoppers involving important data will have to be encrypted and protected to avoid possible fraudulent attacks.

Supportability

The platform will have to provide support to the users regarding to the order process and possible refunds. Therefore some kind of communication bridge between the user and the administrator will have to be created.

4.1.3 Use Cases Diagram

The following is the representation of the use cases that have been found during the requirements analysis of the second iteration.



4.1.4 Use cases specification

This is the detailed specification of the use cases of the second iteration. The format is the same as the one followed in the first iteration.

Order Tipping Offer

Use case:	Order tipping offer
Actors:	Shopper
Goal:	Order and pay a tipping offer
Summary:	The shopper is interested in one tipping offer and wants to make a purchase of an amount of the tipping offer.
Typical course of events:	1. The shopper requests to order an amount of active tipping offer. 2 . The shopper proceeds with the payment with credit card. 3. The system saves the payment and shows the order confirmation to the shopper with the payment information and the redemption information. 4. The system verifies if the price of the offer has tipped and decreases the current price of the offer. In case the maximum number of purchases is reached the offer is marked as sold out. 4. The actor can bring the redemption code to the vendor to redeem the deal.
Alternative course:	2. If the offer is not active or has reached the maximum total of purchases or the actor has reached his maximum number of purchases for that offer the system stops the purchases and notifies the shopper about the limit reached. 3. If the system can't succeed with the payment the purchases are stopped and the shopper is notified about the problem.

List Orders

Use case:	List orders
Actors:	Shopper
Goal:	Show the information of all orders the shopper has made.
Summary:	A list with the information of the orders the actor has made is shown to the actor
Typical course of events:	1. The actor requests the system to view a list of his orders. 2. The system shows the information of each one of the orders (obviously including the redemption information and codes) the actor has made.

List All Orders

Use case:	List all orders
Actors:	Administrator
Goal:	Show the information of all orders in the system
Summary:	A list with the information of the orders saved in the system is shown to the administrator
Typical course of events:	1. The administrator requests the system to view a list of all the orders. 2. The system shows the information of each one of the orders saved in the system and the assigned codes as well.

Refund Tipped Price

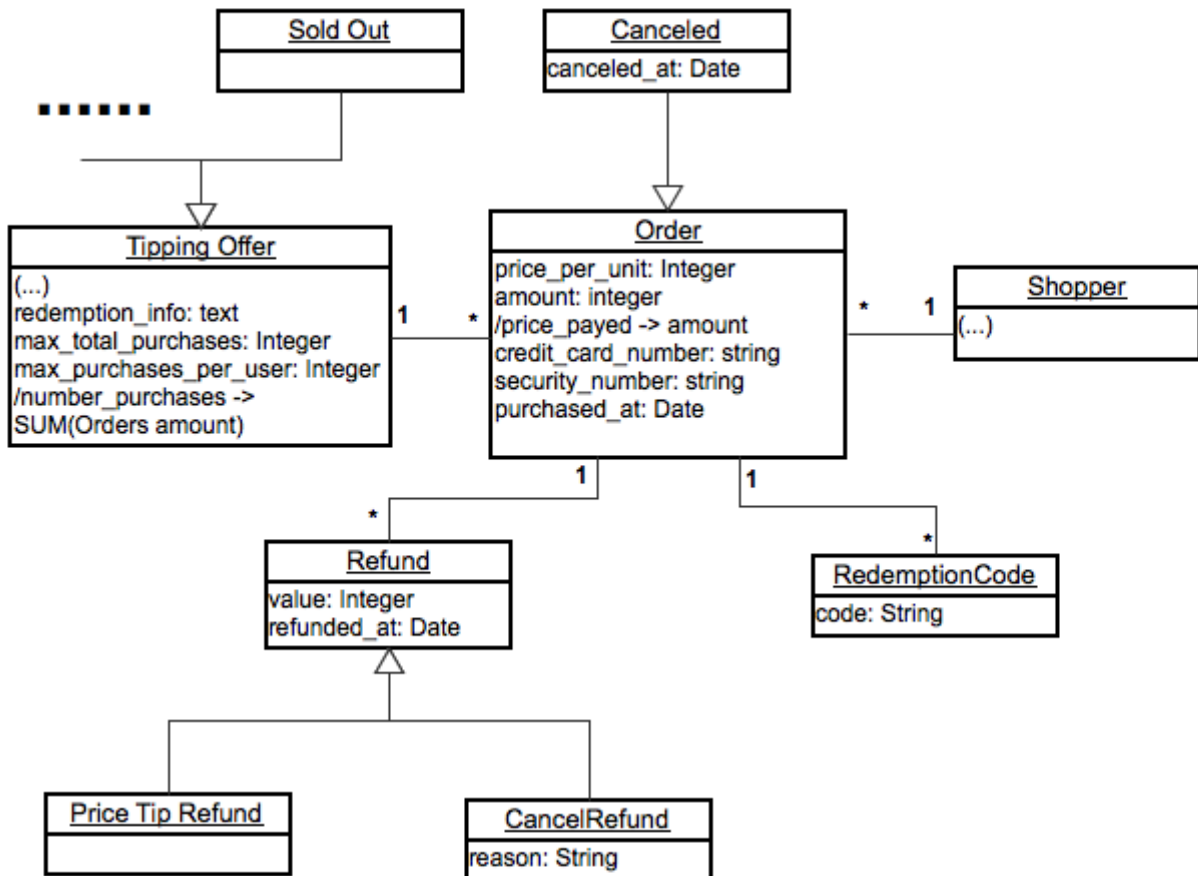
Use case:	Refund tipped price
Actors:	Administrator
Goal:	Refund the difference between the price paid in orders and the final price
Summary:	The administrator wants to refund the difference between the price an administrator paid for a tipping offer and the price level reached in the end.
Typical course of events:	1. The administrator requests to refund the tipped price of an order. 2. The system refunds the difference between the price paid per each offer and the price reached for that tipping offer when it finished.
Alternative course:	2. If the price paid was the same as the final price of the tipping offer the system stops the refund and notifies the administrator.

Cancel order

Use case:	Cancel order
Actors:	Administrator
Goal:	Cancel an existing order
Summary:	The administrator might want to cancel an order for several reasons: mistaken purchases, merchant cancellation, etc.
Typical course of events:	1. The administrator requests the system cancel an order. 2. The system refunds the price paid to the used credit card (taking into account possible previous tipping refunds) and marks the order as canceled. This cancellation does not affect the total number of purchases (otherwise the tipping offers could go back to a previous price which is not a desired behavior).
Alternative course:	2. If the offer has been already cancelled the system stops the cancellation and notifies the administrator.

4.1.5 Conceptual data model

The existing data model has to be modified and extended to support all the new data from the new requirements. Here it is shown only the new entities and modified ones to be able to differentiate the changes produced in the system. Here is a description of the changes performed in the data model as well as the data diagram that reflects them:



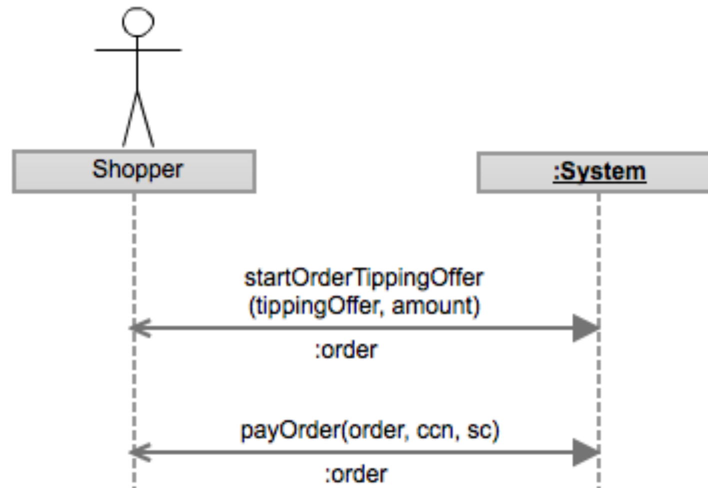
- **RedemptionCode:** A redemption code is a unique code that verifies to the merchant that the shopper had actually bought the offer. It is randomly generated when the order is made.

- **TippingOffer:** There is new information to be added to the tipping offers. The redemption information, the maximum total purchases and the maximum purchases per shopper. Also a new type of offer appears as the tipping offers:
    - **SoldOut:** A tipping offer becomes sold out when the maximum number of purchases has been reached.
- **Order:** The order saves the information of the ordering process. The price per unit paid at the moment of the order and the amount ordered. It also saves the date the purchase was made and the payment information as future possible refunds might need to be done.
    - **Canceled:** A cancelled order is produced for some abnormal situation with an order. The administrator can cancel the order. This cancellation does not affect the number of purchases made for a tipping offer.
- **Refund:** A refund is an entity that saves the action of returning some money to the shopper. The value of a refund can never be higher than the price of the order minus the existing refunds. This constraint will help ensure no mistake is made with the refunds and more money than the initially paid is returned. We want the date that refund was processed for tracking payment reasons. A purchase can be for different reasons:
    - **PriceTipRefund:** This is a normal refund for Tipping Offers. If I paid X price when I purchased the offer and after a while the price tipped (dropped because a new price level was reached), this money is returned to the shopper through a refund.
    - **CancelRefund:** This refund is done when an abnormal situation occurs. The merchant might run out of stock or the shopper might have ordered mistakenly a tipping offer. This refund saves the amount refunded to the shopper and the reason.

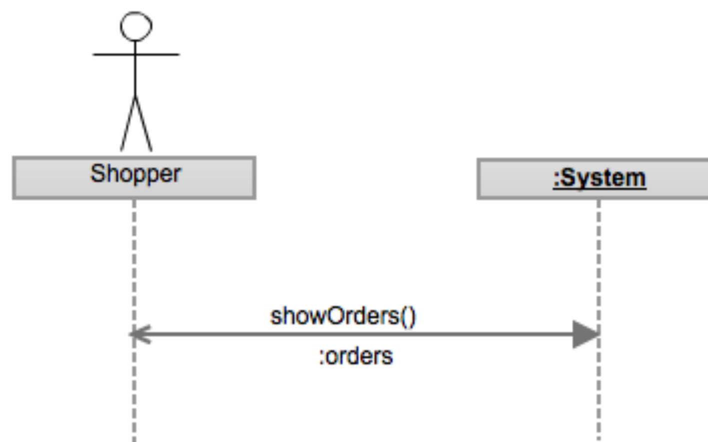
4.1.6 System Sequence Diagrams

The following diagrams represent the interactions between the user and the system in each one of the use cases previously analyzed in the second iteration.

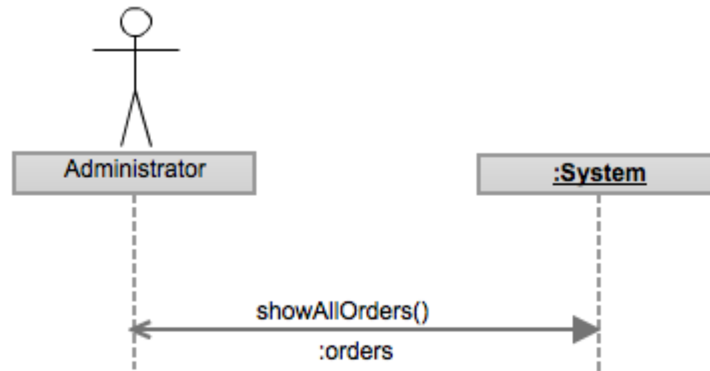
Order Tipping Offer



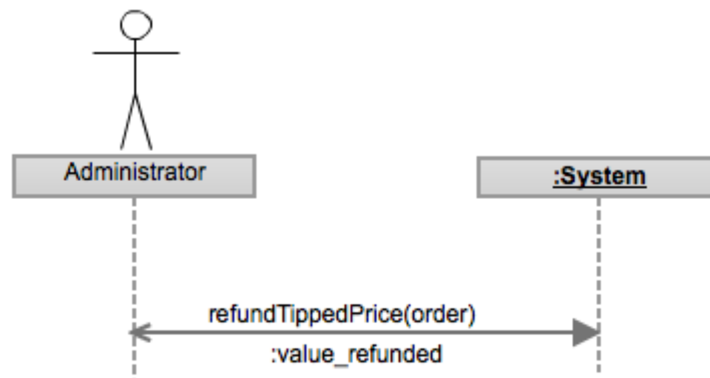
List Orders



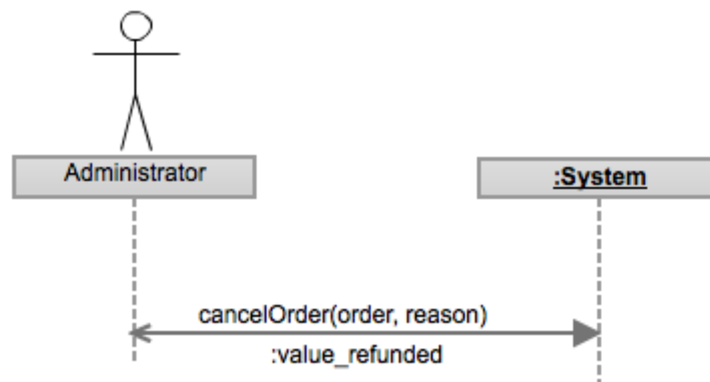
List All Orders



Refund Tipped Price



Cancel Order



4.1.7 Operation Contracts

Following the same procedure as it was done in the first iteration, here is the diagram of the system operation and the contract descriptions that belong to the previous system sequence diagrams.

SYSTEM
startOrderTippingOffer(tippingOffer, amount) :order **payOrder(order, ccn, sc) :order** **showOrders() :orders** **showAllOrders() :orders** **refundTippedPrice(order) :value_refunded** **cancelOrder(order, reason) :value_refunded**

Name:	**startOrderTippingOffer(tippingOffer, amount) :order**
Responsibilities:	prepare a new order of number 'amount' of 'tippingOffer'
Exceptions:	If the tipping offer is not active, the maximum total purchases or the maximum purchases per user for the shopper is exceeded show error
Preconditions:	tippingOffer is an active offer amount does not exceed the remaining purchases (max_purchases of the offer minus the existing number of purchases) the shopper does not exceed the maximum purchases per user (max_purchases_per_user minus the shopper's existing purchases is equal or bigger than 'amount')
Postcondicions:	'order' is an instance not saved 'order's price_per_unit equals to the current price of the tipping offer 'order's amount equals to 'amount'
Output:	order

Name:	**payOrder(order, ccn, sc) :order**
Responsibilities:	charge the payment of the order to the credit card, and save the information
Exceptions:	if there is a problem with the transaction (numbers invalid / credit card not chargeable / transfer error / etc.) show error
Preconditions:	'ccn' is a valid credit card number and the 'sc' is a security code valid for 'ccn' credit card. The credit card has enough money to pay the order 'order' price
Postcondicions:	'order' is saved in the system. 'order' attribute purchased_at is equal to the current date the offer associated to 'order' becomes SoldOut if the max_total_purchases has been reached a price equal to 'order' price has been charged to the 'ccn' credit card there are 'order' attribute amount RedemptionCodes instances with a unique number associated to 'order'
Output:	'order'

Name: **refundTippedPrice(order) :value_refunded**

Responsibilities: Refund the difference played for a tipping offer and the final price

Exceptions: If the tipping offer associated to 'order' is not of kind Expired show error
If the order did not have any overcharged amount show error (the tipping offer did not tip after the 'order' purchase was performed)

Preconditions: The tipping offer associated to 'order' is Expired
'order' does not have any PriceTipRefund associated
'order' is not Canceled
'order' has an overcharged amount (price paid different than final price) (the price_per_unit in 'order' is higher than the current price of the of tipping offer associated to the order_

Postcondicions: 'value_refunded' is equal to the difference between the price_per_unit in the order and the tipping offer current price multiplied by 'order' attribute amount
a new instance of PriceTipRefund associated to 'order' with value_refunded equal to 'value_refunded' exists.

Name: **showOrders() :orders**

Responsibilities: List all orders that belong to the current shopper

Exceptions: ---

Preconditions: ---

Postcondicions: 'o' is a list of the orders associated to the current shopper, each one with its redemption codes and refunds information

Output: 'o'

Name: **showAllOrders() :orders**

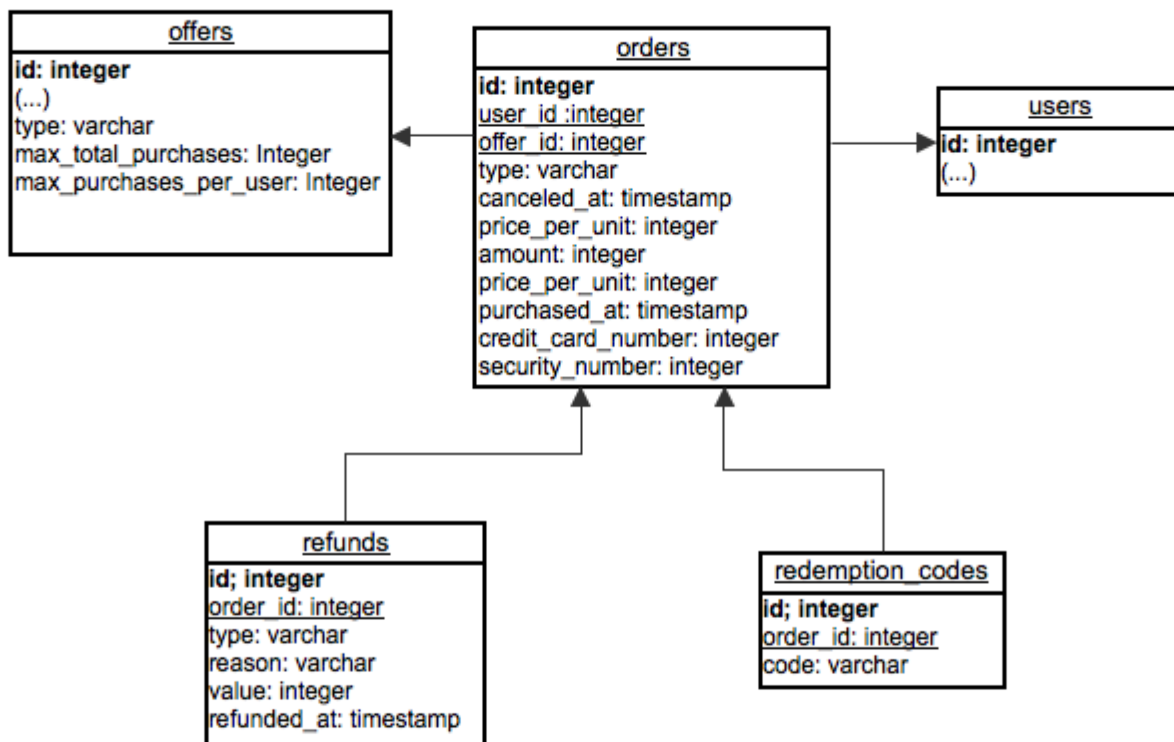
Responsibilities:	List all orders in the system
Exceptions:	---
Preconditions:	---
Postcondicions:	'o' is a list of all the orders in the system, each one with its redemption codes and refunds information
Output:	'o'

Name:	**cancelOrder(order, reason) :value_refunded**
Responsibilities:	Cancel an order and refund the corresponding money
Exceptions:	If 'order' is Canceled show error
Preconditions:	'order' is not Canceled
Postcondicions:	'value_refunded' is equal to the remaining price paid for the order (the price paid minus possible existing PriceTipRefund) It exists a instance CancelRefund associated to 'order' with reason equal to 'reason' 'order' does not have any redemption codes associated
Output:	'value_refunded'

4.2 Design

4.2.1 Normalized Database Diagram

The same procedure as in the first iteration has been followed to normalize the conceptual data model. The result is show here in this diagram:



5 Third Iteration - Subscriptions and Redemptions

This third iteration added is focused in adding a new feature to allow shoppers to subscribe to tipping offers and adds two new redemption strategies of the offers purchased.

The first feature of subscribing users to tipping offers appeared from the need to keep shoppers interested in tipping offers up to date. Some users reported that they would not buy the tipping offer until certain price level. By letting users get subscribed to tipping offers we would inform them about changes that occur, like a price tipped which could potentially persuade them of buying the offer.

Besides that, some users also wanted to receive a periodic newsletter to be informed about new offers. This is a very important feedback, users were actually asking to be advertised offers!

The other feature this iteration was focused in is letting tipping offers have two other redemption strategies. Some merchants requested to use their own pre-generated codes to redeem the products they offered. They used this methodology and it was a problem for them to have to deal with codes that we sent to them.

Some others also requested they did not want to use redemption codes and they would just give some information to the shopper on how to contact them to redeem the deal.

This feature did not actually add new functionality from the user perspective but tweaked the logic of some the existing use cases. The following section will follow up with the changes to the existing use cases as well as analyze the newly welcomed ones.

5.1 Specification

5.1.1 Functional Requirements

The shoppers might not want to buy right away a tipping offer until it reaches a certain price.

Let's say we sell "one user test at usertesting.com" with a current price $10 with 20 purchases made. Once enough purchases have been made the next prices are $8 with 30 purchases, $6 with 50 purchases and $5 with 100 purchases. A shopper might think he is willing to pay up to 8$. We do not want to lose this sale in case the tipping offer reaches 8$.

Shoppers can **subscribe to a tipping offer** in order to be able to receive an update when the price lowers.

On the other hand, some shoppers who really like the idea behind tipping offers, just want to **get updates whenever any tipping offers'** price lowers.

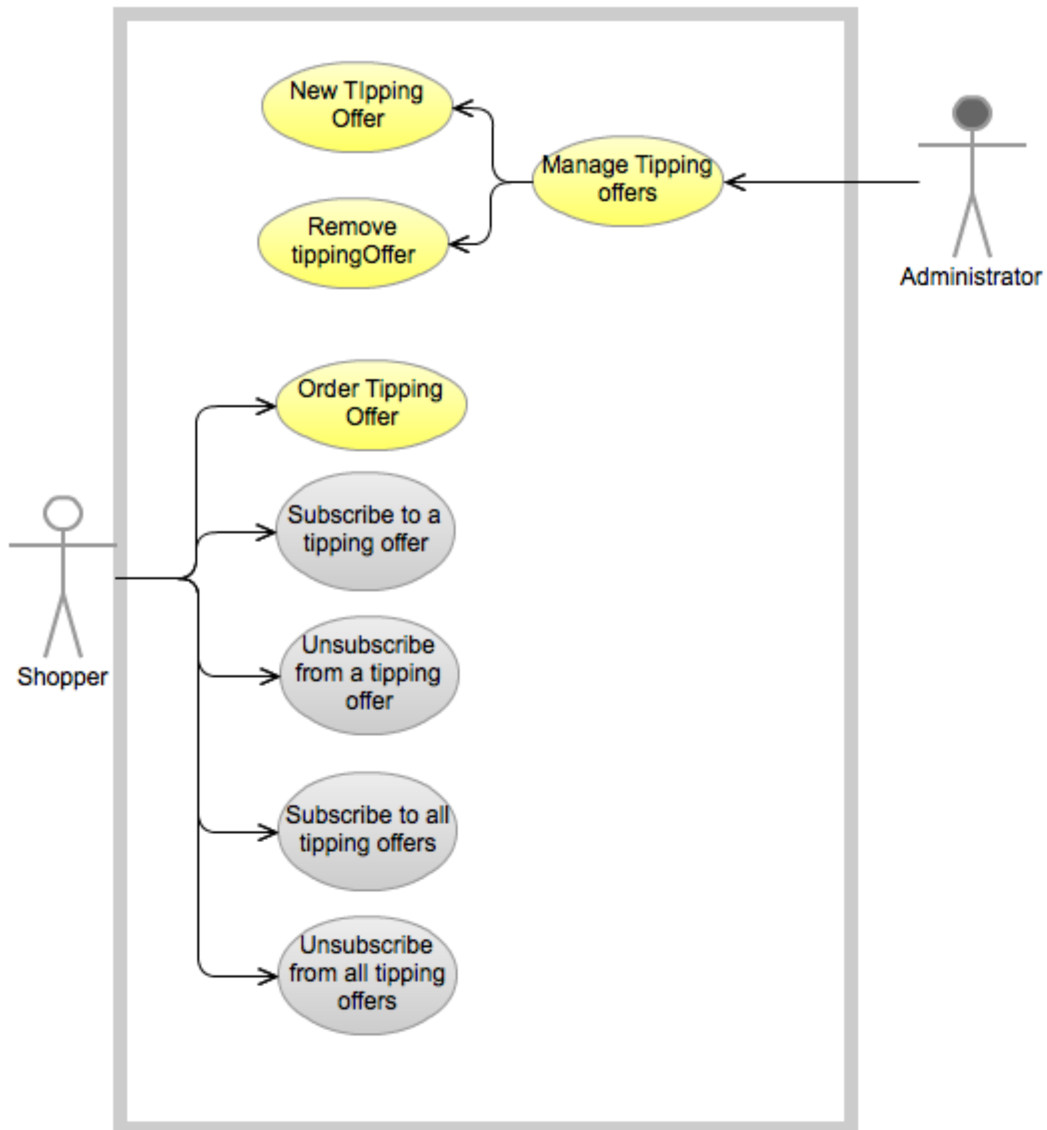
Of course in any of both subscription cases, we have to be able to **let the users unsubscribe** in case they get bothered by the updates, or just do not feel interested in the tipping offer anymore.

Some merchants have their own system of redemption codes. Sometimes it is not possible for them to use our own random generated redemption codes. They just can **supply us with a redemption code list**. In the case we are selling a tipping offer for this merchant, we will need to supply the shopper with redemption codes from the mentioned list instead of our own random generated code list.

Some merchants do not even want to deal with redemption codes. They simply trust the word of "I am coming from *Rewardli*" to give the discount to the shoppers. Therefore when we sell a tipping offer from one of those merchants we will simply **not have to deal with any kind redemption codes.**

5.1.2 Use Cases Diagram

The existing use cases affected by the new features have been highlighted in yellow. Their use case specification has been also updated accordingly in the next part as well as some of the contract operations (the system sequence diagrams have stayed the same though).



5.1.3 Use Cases Specification

These are the analyzed use cases that have been found analyzing the functional requirements of the third iteration.

New Tipping Offer (modified)

Use case:	New tipping offer
Actors:	Administrator
Goal:	Create a new tipping offer
Summary:	The administrator wants to create a new tipping offer and schedule it for a specific interval of time.
Typical course of events:	1. The use case starts when administrator requests to create a new tipping offer. 2. The system shows the different options to create a tipping offer to the administrator. 3. The administrator describes the tipping offer, selects the redemption method (and adds a list of redemption codes if it is a pre-generated list). 4. The administrator enters the different price levels and completes the creation. 4. The system records the information about the tipping offer and redemption strategy and confirms it to the administrator.
Alternative course:	4. The description or times given by the administrator are not valid. The system does record the tipping offer. It asks the administrator to try again.

Remove Tipping Offer (modified)

Use case:	Remove tipping offer
Actors:	Administrator
Goal:	Remove an existing tipping offer.
Summary:	The administrator removes an existing tipping offer.
Typical course of events:	1. The administrator requests to remove a specific offer of the system. 2. The system removes the offer, the unused redemption codes that it has and confirms the action to the administrator.
Alternative course:	---

Order Tipping Offer (modified)

Use case:	Order tipping offer
Actors:	Shopper
Goal:	Order and pay a tipping offer
Summary:	The shopper is interested in one tipping offer and wants to make a purchase of an amount of the tipping offer.
Typical course of events:	1. The shopper requests to order an amount of active tipping offer. 2 . The shopper proceeds with the payment with credit card. 3. The system saves the payment and shows the order confirmation to the shopper with the payment information and the redemption information. 4. The system verifies if the price of the offer has tipped. If it is the case, it decreases the current price of the offer and notifies all subscribed shoppers to the bought offer and all subscribed shoppers to all tipping offers. In case the maximum number of purchases is reached the offer is marked as sold out as well. 4. The shopper can bring the redemption code to the vendor to redeem the deal.
Alternative course:	1. If the offer is not active, has reached the maximum total of purchases, the shopper has reached his maximum number of purchases for that offer or there are no more redemption codes if the redemption method is a pre-generated list of codes, the system stops the purchases and notifies the shopper about the limit reached. 3. If the system can't succeed with the payment the purchases are stopped and the shopper is notified about the problem.

Subscribe to a tipping offer

Use case:	Subscribe to a tipping offer
Actors:	Shopper
Goal:	Subscribe the shopper to a tipping offer
Summary:	A shopper gets subscribed to a tipping offer to receive updates of it.
Typical course of events:	1. The shopper requests the system to get subscribed to a tipping offer. 2. The system subscribes the shopper to the tipping offer and will send updates to the shopper whenever they happen, particularly when the offer tips and few hours before it expires.

Unsubscribe from a tipping offer

Use case:	Unsubscribe from tipping offer
Actors:	Shopper
Goal:	Unsubscribe the shopper to al tipping offers changes
Summary:	A shopper gets unsubscribed to from receiving updates about a tipping offer in the system.
Typical course of events:	1. The shopper requests the system to get unsubscribed of a tipping offer. 2. The system unsubscribes the shopper from the tipping offer and will not send any updates about the specified tipping offer anymore.

Subscribe to all tipping offers

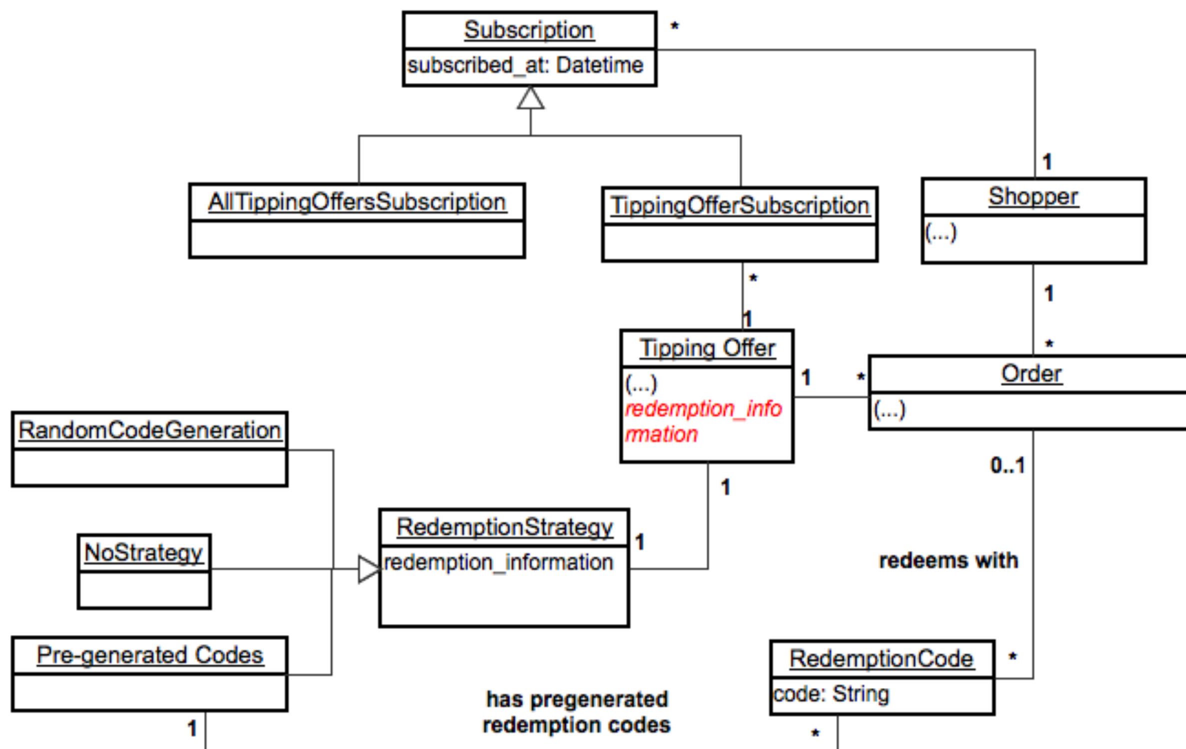
Use case:	Subscribe to all tipping offers
Actors:	Shopper
Goal:	Subscribe the shopper to al tipping offers changes
Summary:	The shopper gets subscribed to receive updates about any tipping offer in the system.
Typical course of events:	1. The shopper requests the system to get subscribed to all the tipping offers. 2. The system subscribes the shopper to all the tipping offers and will notify him of updates about any tipping offer.

Unsubscribe to all tipping offers

Use case:	Unsubscribe from all tipping offers
Actors:	Shopper
Goal:	Unsubscribe the shopper from all the tipping offers
Summary:	The shopper gets unsubscribed from updates of all tipping offer.
Typical course of events:	1. The shopper requests the system to get unsubscribed from all tipping offers subscription. 2. The system unsubscribes the shopper from all tipping offers and will not send any more updates about any tipping offer changes.

5.1.4 Conceptual Data Model

In this iteration, not only new entities have been added to the relation but some relations and information has been moved from. It looks like not much new information has been added to the system but the new entities and their relation do already represent the data needed to represent the new information. The final result is here shown, with an explanation about the new entities following it:



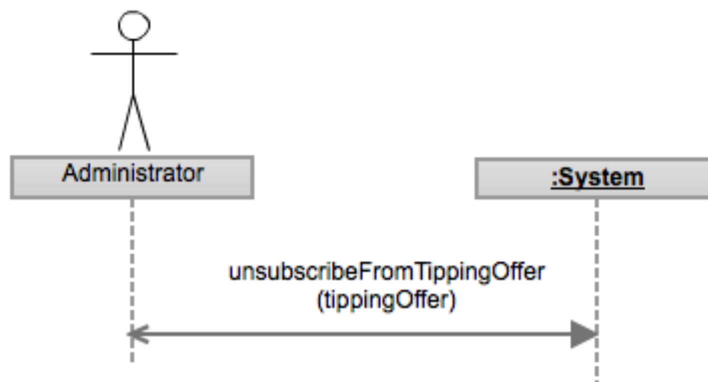
- **Subscription.** A subscription is an entity that reflects that a shopper has subscribed to something at a specific moment in time. Right now there are only two different types of subscriptions but more could easily be added in the future.
    - **AllTippingOffersSubscription.** This subscription shows that a user wants to receive updates whenever a tipping offer is going to expire or has tipped price.
    - **TippingOfferSubscription.** This subscription represents the user subscription to a specific tipping offer.
- **TippingOffer.** The tipping offers can now have a redemption strategy. Therefore the redemption information is more related to a redemption strategy than the offer itself and has been moved accordingly.
- **RedemptionStrategy.** The redemption strategy represents the way the redemption of the tipping is done with the merchant holds all the information needed for the redemption (the pre-generated redemption codes in case they are needed for example).
    - **NoStrategy.** This is strategy does not manage any redemption codes, it only shows the information that will have been provided by the merchant.
    - **RandomCodeGeneration.** This strategy creates a random-generated RedemptionCode each time a new order is done for a tipping offer.
    - **PreGeneratedCodes.** This strategy has a list of codes that can be used for redemption. Each time an order is created the corresponding number of pre-generated codes is moved to the order.
- **RedemptionCode.** A redemption code can now be associated with a PregeneratedCodeStrategy. If this is the case that means the RedemptionCode has not been used by any order. That is, a RedemptionCode cannot be related at the same time to a PregeneratedCodeStrategy and an Order.

5.1.5 System Sequence Diagrams

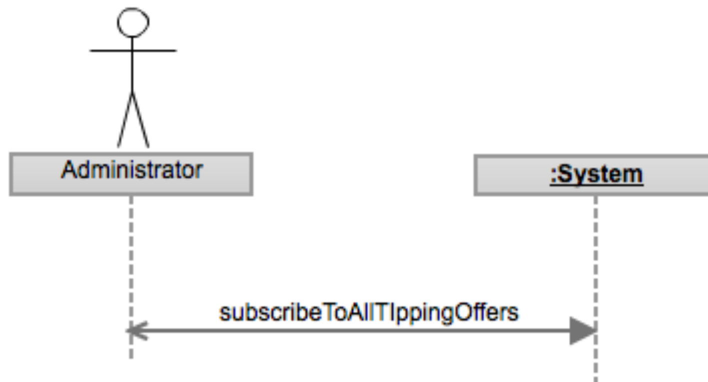
The sequence diagrams for the uses cases 'New Tipping Offer', 'Remove Tipping Offer' and 'Order Tipping Offer' remain as in the second iteration. For the 'New Tipping Offer' the only clarification to make is that the (information*) that the administrator gives to the system includes as well the redemption strategy and in case it is a pre-generated list, the redemption codes.

Subscribe to tipping offer

Unsubscribe from tipping offer



Subscribe to all tipping offers



Unsubscribe from all tipping offers

5.1.6 Operation Contracts

The operation contracts of the new use cases and the ones that have changed are here shown and specified.

SYSTEM
(modified) **updateTippingOffer(tippingOffer, information*)**
(modified) **removeTippingOffer(offer) :confirmation**
(modified) **payOrder(order, ccn, sc) :order**
subscribeToTippingOffer(tippingOffer)
unsubscribeFromTippingOffer(tippingOffer)
subscribeToAllTippingOffers()
unsubscribeFromAllTippingOffers()

Name:	**updateTippingOffer(tippingOffer, information*)**
Responsibilities:	save information to a tipping offer
Exceptions:	if the title is empty or the redemption method is unknown show error
Preconditions:	information is a set of information of the offer (title, description, redemption information, redemption method...). The title and the redemption method are present. If redemption method is “PregeneratedCodes” then a list of codes is also supplied.
Postcondicions:	tippingOffer has the attributes in information* RedemptionCode exist with each of the redemption codes that were supplied (if any)

Name:	**removeTippingOffer(offer)**
Responsibilities:	remove the tipping offer record
Exceptions:	if offer does not exist show error
Preconditions:	offer exists
Postconditions:	offer does not exist anymore in the system offer does not have any redemption codes

Name: **payOrder(order, ccn, sc) :order**

Responsibilities: charge the payment of the order to the credit card, and save the credit card information

Exceptions: if there is a problem with the transaction (numbers invalid / credit card not chargeable / transfer error / etc.) show error

Preconditions: 'ccn' is a valid credit card number and the 'sc' is a security code valid for 'ccn' credit card. The credit card has enough money to pay the order 'order' price

Postcondicions: 'order' is saved in the system.
'order' attribute purchased_at is equal to the current date.
the offer associated to 'order' becomes SoldOut if the max_total_purchases has been reached
a price equal to 'order' price has been charged to the 'ccn' credit card
if the redemption strategy of the offer is "RandomCodeGeneration" there are 'order's attribute amount RedemptionCodes instances associated to 'order'
if the redemption strategy of the offer is "PregeneratedCodes", 'order's attribute amount RedemptionCodes that belong to offer and had no order are associated to 'order'
if the price of tippingOffer associated to 'order' has tipped, all the users that have a TippingOfferSubscription with tippingOffer and all the users that have a AllTippingOfferSubscription have been notified of the price change

Output: 'order'

Name: **subscribeToTippingOffer(tippingOffer)**

Responsibilities: subscribe the current shopper to receive updates about tippingOffer

Exceptions: if the current shopper is already subscribed to tippingOffer show error

Preconditions: it does not exist any TippingOfferSubscription with tippingOffer and the current shopper

Postconditions: ‘s’ is a new instance of TippingOfferSubscription
‘s’ is associated with the current shopper
‘s’ is associated with tippingOffer
‘s’ subscribed_at has the present date and time.

Name: **unsubscribeFromTippingOffer(tippingOffer)**

Responsibilities: unsubscribe the current shopper from receiving updates about tippingOffer

Exceptions: if the current shopper has no TippingOfferSubscription associated with tippingOffer show error

Preconditions: the current shopper has a TippingOfferSubscription associated with tippingOffer

Postconditions: there is no TippingOfferSubscription associated to the current shopper and tippingOffer

Name: **subscribeToAlTippingOffer**

Responsibilities: subscribe the current shopper to receive updates about any tipping offer

Exceptions: if the current shopper already has a AllTippingOfferSubscription show error

Preconditions: the current shopper does not have any AllTippingOfferSubscription associated

Postconditions: 's' is a new instance of AllTippingOfferSubscription
's' is associated with the current shopper
's' subscribed_at has the present date and time.

Name: **unsubscribeFromAllTippingOffer**

Responsibilities: unsubscribe the current shopper from receiving updates about any tipping offer

Exceptions: if the current shopper does not have an AllTippingOffersSubscription associated show error

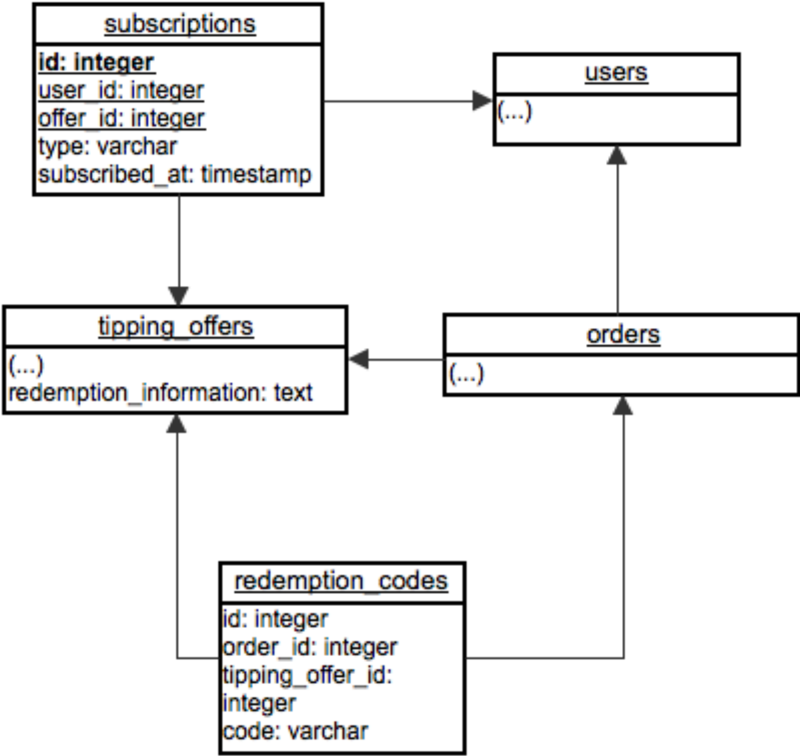
Preconditions: the current shopper has an AllTippingOffersSubscription associated

Postconditions: it does not exists any AllTippingOffersSubscription associated to the current shopper

5.2 Design

5.2.1 Normalized Database Diagram

The same procedure as in the first iteration has been followed to normalize the conceptual data model. One specific un-abstraction has been designed to simplify the database and implementation process of redemption strategies. As all the redemption strategies could share the same connections and information in terms redemption information and association with redemption codes (0 codes for NoStrategy and RandomCodeGeneration and 0.* for PregeneratedList) the RedemptionStrategy class has been disregarded to have its own table and entity. Consequently the redemption information has been saved as it was done previously in the offers table and the redemption codes for pre-generated code lists have been associated with the tipping_offers. The result is in the following diagram:



6 Implementation

Many different technologies have been used during the implementation of this project. The most important criteria taken into account to choose the technologies used in this project have been: the community supporting the technology, the current team knowledge versus the learning curve, the value versus the complexity brought to the project.

As it was mentioned in the scope of the project, Jean the CTO, chose the architecture stack, the Chief Technology Officer of the company. However, it is very interesting to analyze the new and old elections and their resulting benefits and drawbacks found during the implementation process.

The following part of the memory will present this technologies used in the implementation of the project, where and how they have been used and the most important part, the reasons behind to select them.

6.1 Backend

6.1.1 Ruby

Ruby is a dynamic and open-source object oriented programming language. Ruby offers an elegant syntax oriented to productivity. The syntax that the language offers is probably more human-readable that most of the other language.

This productivity is not only achieved by the amount of time that takes a product or feature to get done but also the amount of code. As a general rule of thumb, the less amount of code, the easier it is to maintain.

Behind this simplicity Ruby is a language with some very powerful features. The use of anonymous functions and the use of blocks are some of these interesting and very useful features of the language. It is object-oriented and each one of its basic types is an object, including booleans and integers.

Ruby has also an important amount of libraries and it can also be integrated with other language libraries easily.

The team had previous experience working with Ruby on Rails and the *Rewardli* project was mounted on a stack with Ruby and Ruby on Rails framework. It was totally worth integrating this project into the existing stack and application and there was no reason for changing it as the previous experience with the existing project was delightful.

6.1.2

6.1.3 Ruby on Rails

Ruby on Rails is a well-known full stack open-source web framework. The main features of Ruby on Rails stack offers are:

- **ActiveRecord**: Object-relational mapping system for models that can be set up with different databases.
- **ActiveResource**: Web services package.
- **ActionPack**: Base for controllers and views of MVC.
- **ActiveSupport**: A general utility library.
- **ActionMailer**: Email framework.

One of the most interesting things is that it is developer oriented, quoting a caption of the homepage: "*Optimized for programmer happiness and sustainability productivity*". This means that not only the product development has been the main focus when Ruby on Rails was first created but the developer happiness itself.

The first Ruby on Rails principle is "*Convention over configuration*". Configuration set up is avoided as much as possible if a reasonable convention can be taking. The clearest example for this is naming. The table that maps a model in rails is always named as the plural of the model name.

The second most important Ruby on Rails principle is "*Don't repeat yourself*". Code redundancy is Rails enemy and a good toolset made of helpers and partial views among others is used to try to achieve the 0-redundancy.

It has a very active community supporting and improving the framework in a day-to-day basis.

The existing application of *Rewardli* was implemented in Ruby on Rails. Because of that, the advantages of integrating this project into it and the previous of experience of the team made Ruby on Rails and trustable and reliable choice.

6.1.4 Hosting Server

The hosting server service is one of the most important decision when designing a web architecture stack Hosting servers rely on different technologies, and sometimes they do not allow the user to choose all the features he would want to have.

Heroku is a hosting service in the cloud that eases the deployment of the new code versions and all the tasks related to the administration. Due to the size of the team, this is very important because it lets the team put the effort in the most important thing: the product.

The main advantages of hosting the application with Heroku are the scalability and efficiency (the latter in terms of usability). Heroku makes it very easy manage the resources of the server. If the traffic of the site grew from 100 visits to 10.000 a month, we could adjust the number of running instances in the server in a matter of minutes with just one command.

6.1.5 PostgresSQL

The decision of which SQL database would be used in the application was mainly driven by the hosting service. Heroku only supported PostgresSQL databases so there was not much margin of election.

Despite that PostgresSQL is known for being a flexible and scalable database, so it did actually fit perfectly in the stack

6.1.6 Rspec

The unit tests are implemented with RSpec. The RSpec tool is one the most used tools for unit testing in ruby. It offers a nice organization and provides with the necessary tools to write simple isolated unit tests.

6.2 Frontend

6.2.1 HTML with HAML

HTML is the markup language that structures the content in the world wide web. This was obviously the language used in the views of the application that are sent to the web browsers.

But in the development environment, the implementation was done with HAML. HAML is also a markup language but it compiles into HTML. The four basic principles used when HAML was created were: 'Markup has to be beautiful', 'Markup should be DRY', 'Markup should be well-indented' and 'HTML structure should be clear'. It has a more straightforward and beautiful syntax, which is always a good thing for the code readability and the comfort of developers of the app. Here is an example of how a piece of code looks in HTML and HAML

Sample code in HTML

```
1  <html>
2      <head>
3        <meta name="keywords content="keywords"/>
4      </head>
5      <body>
6          <div id="main-container">
7               <h1>Sample header</h1>
8               <p>Lorem ipsum.....</p>
9          </div>
10     </body>
11 </html>
```

Same sample code in HAML

```
1 %html
2 %head
3     %meta{:name => "kewords",
4           :content => "keywords"}
5     %body
6         #main-container
7         %h1 Sample header
8         %p Lorem ipsum....
```

6.2.2 CSS with SASS

CSS is a stylesheet language that provides HTML markup language visual styles: backgrounds, shapes, sizes and text color among many others. It provides a layer of style separated from the content and the markup language itself. CSS is used in the world wide web and it is widely supported through the browsers (with some specific differences that will not be presented in this document).

But CSS lacks many basic programming paradigms. There is no way in CSS to define constants (for example a color we would use throughout the site) or anything similar. In a medium sized project it becomes a tedious language to work with.

SASS is an extension of CSS that compiles into CSS. This way we can use a richer language that integrates features like declaring constants, mixings and other paradigms. Furthermore, there is no compatibility issue as SASS compiles into CSS that is the code that will be sent to the browsers.

Besides that, SASS has a nicer markup. It has similar features to HAML, it uses indentation to provide closure tags among other conventions.

Sample styles in CSS

```
1 a {
2   color: #00FF00
3 }
4 a:hover, a:visited {
5   color: #0000FF
6 }
7 a:active {
8   color: #00FF00
9 }
```

Same sample styles in SASS

```
1 $green = #00FF00
2 $blue  = #0000FF
3 a
4   color: $green;
5   &:hover, &:visited
6     color: $blue
7   &:active
8     color: $green
```

6.2.3 JavaScript

JavaScript is the dynamic prototype-base standard language for the client side in the world wide web. JavaScript can interact with the document object model that represents the HTML information given from the website.

JavaScript interacts with the HML using the Document Object Model (DOM). The DOM is a convention representation of the HTML data tree.

This project has used JavaScript for the client side views, to provide a richer and more pleasant user experience. Some of those features that were implemented in the client side are input validation, Ajax interactions, field autocompleting and visual effects.

6.2.4 jQuery

jQuery is a free open-source JavaScript library that simplifies client-side scripting. It helps to speed up the development time of the features previously mentioned. jQuery also guarantees cross-browser compatibility, which is an issue to take into account when using JavaScript libraries.

The main advantage of using jQuery how it makes it much more easy to navigate around the Document Object Model from HTML.

It holds big repository of plugins that can be installed and ready to use in matter of minutes. From auto-completers, dynamic tables, animation effects to Ajax libraries, slideshow galleries, timers and rich document text edition. Endless options which one can also personalize to its needs depending on the license of use.

6.3 Testing Technologies

The unit tests are implemented for each Controller, Model and class in general. They give a general confidence of the reliability of the implementation and even more important, help a lot in the process of continuous changing the product by alerting of neglected behaviors.

The integration tests were created by writing a set of instructions that represent the flow a user would follow to perform an important task (for example order a tipping offer). These integration tests ensure that the general behavior and most important features worked properly.

Selenium web driver and RSpec together with Capybara library is the toolset that has been used to write integration tests. Selenium is a web browser automation tool. It provides an interface from which you can give instructions to perform with a browser (click links, fill forms, etc.). Capybara is a library that interacts with Selenium. It integrates perfectly with RSpec.

```
1  describe "create and show a tipping offer", :js => true do
2    before :all do
3      @offer = Factory(:tipping_offer_with_items)
4    end
5
6    it "should create and show a tipping offer" do
7      click_link "Sign up / Log in"
8      fill_in "Email",    "polmiro@gmail.com"
9      fill_in "Password", "test_password"
10     click_link "Log in"
11
12     visit new_tipping_offer_path
13     fill_in "Id", 1
14     fill_in "Title", "Adobe Photoshop CS6"
15     fill_in (...)
16     click_button "Create Tipping Offer"
17
18     visit tipping_offer_path(:id => 1)
19
20     page.should have_content?("Adobe Photoshop CS6")
21   end
22 end
23
```

7 Planning, monitoring and budget

This section explains the planning, monitoring and the budget of the whole project. On one hand it will show how the different tasks have been organized, and in the other it will describe the costs of the hardware, software and human resources.

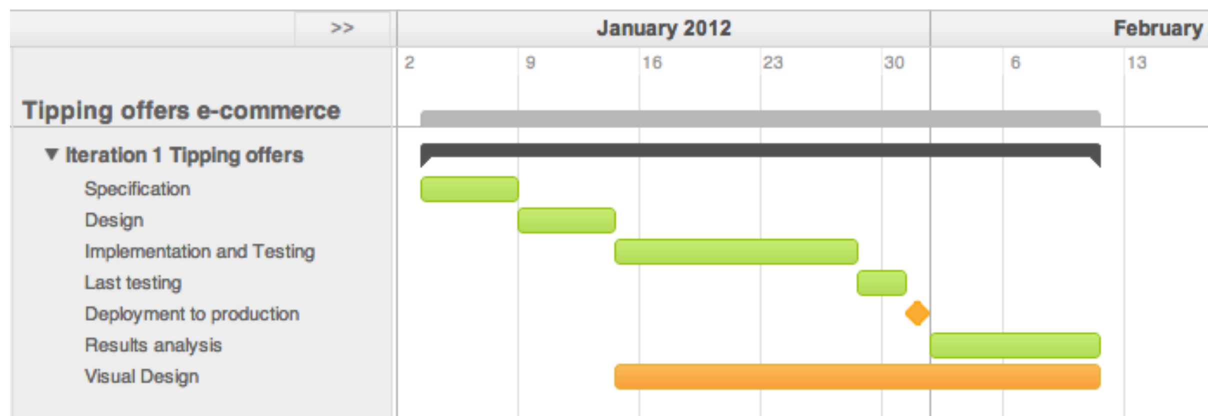
7.1 Planning description

The following *gantt* diagram represents how the time has been organized and spent during the different iterations. For visualization purposes it has been split in three images (one per iteration).

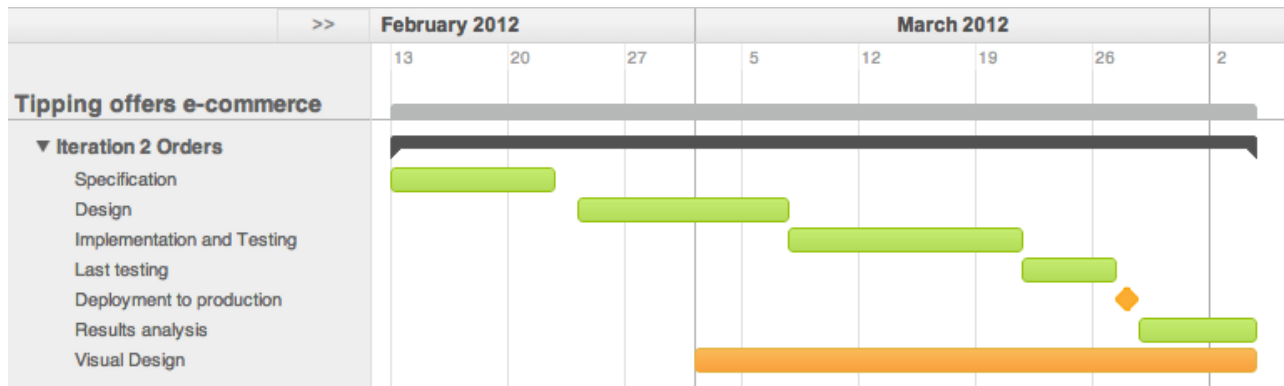
As it has been described throughout this memory, the project is organized in different iterations. All of them follow the same pattern, the common development phases. The implementation and testing are done incrementally together with unit testing, and integration tests when a flow is finished. The visual design was developed and improved once the system was specified and designed by our designer.

The last phase showed in each iteration called "Result analysis" represents the time where the team analyzed the results of the completed iteration and started to prioritize the stories for the next iteration.

7.1.1 Iteration 1 – Tipping offers

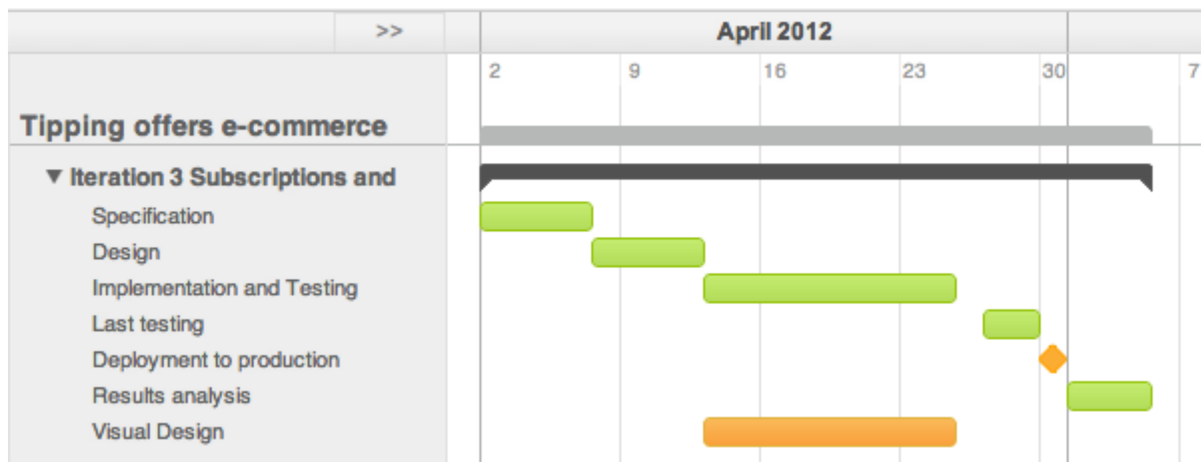


7.1.2 Iteration 2 - Orders



This second iteration was the longest one because it involved a payment flow. This feature was designed in a very low-level detail to provide a very high robustness against unexpected behaviors. Uncompleted payments or double charges where a total unacceptable.

7.1.3 Iteration 3 – Subscriptions and redemption strategies



The third iteration is the shortest one and was finished in less than a month. The use cases where simple and the development perfectly smooth. The visual design was also minimal and could reuse previously created styles.

7.2 Budget study

The following section analyzes all the material used during the development of this project and calculates its approximate budget. The rates used are exact in terms of products and services, and market approximated in the United States for the human resources.

7.2.1 Hardware / Hosting

To develop this project several machines have been used for development purposes. Specifically 4 laptops for each one of the team individuals. Also two servers hosted in the cloud at *Heroku*, one for production and one for staging environments. This is the detailed budget:

Equipment/Service	*Description*	*Price*	*Recurring*
Macbook Pro 13”	Development machine for each one of the team members	4 x 1100$= **$4,400**	-
Heroku Production Hosting	4 dynos server, with Ronin Cache Database Plan, Sendgrid Add-on and Hosting SSL	-	$323 + $200 + $80 + 20$ = $623 / month
Heroku Staging Hosting	1 dyno server, with Ronin Cache Database Plan	-	$35 + $200 = $235 / month
		$4400	**$858 / month**

The total cost after 4 months of development in Hardware and Hosting is **$7832**
The recurring cost to maintain the hosting after this time is **$858 / month**

7.2.2 Software

Different software has been used as well, to support the development and team-working. This is the detailed list:

Software	Description	Price	Recurring
Microsoft Mac Office 2011	*Basic tools (license)*	4 x 200 $ = $800	-
Balsamiq Desktop	*Web mocking software (license)*	1 x $80 = $80	-
Railsonfire	*Continuous Integration (beta license)*	-	-
Campfire	*Communication tool*	-	$12 / month
AgileZen	*Task-board tool for*	-	$29 / month
Ruby	*Language&Framework*	-	-
Ruby on Rails	*Framework*	-	-
PostgreSQL	*DBMS*	-	-
RSpec/Capybara/Other	*Testing Tools*	-	-
mVim / aquaMacs	*Text Editor*	-	-
Adobe Photoshop CS5	*Image editor Software*	$600	-
TOTAL		**$1480**	**$41 / month**

The total cost after 4 months of development in Sofware is **$1644**

The recurring cost to maintain the services for development after that is **$41 / month**

7.2.3 Human Resources

To study the budget for human resources we will estimate the number of hours for each one of the roles in the team and use an approximate market salary in the United States. The roles chosen could have been different: more specific for example. But they are the ones that describe best the Rewardli team.

The different roles that we will take into account are:

- **Product Manager**
- **Full-Stack Engineer**
- **Web designer**

Taking into account the project planning and the amount of hours spent per each task at a rate of 9 hours per day, and the following approximate price table:

Role	***$ / hour***	***Hours***	***Total***
Product manager	50	200h	$10,000
Full-Stack Engineer	45	800h	$36,000
Web designer	35	500h	$17,500
	TOTAL	**1500h**	**$63,000**

7.2.4 Total budget

Having analyzed the hardware, software and human resources, the last step is to calculate the total budget of the project during these four months and the recurring price form maintaining the product, the latter in terms of software and hardware services.

Type	Budget
Hardware	$7,832
Software	$1,644
Human Resources	$63,000
TOTAL	**$72,476**

The recurring cost after the development of this project for the server maintenance and development services is **899$ / month**.

8 Conclusions

8.1 Achieved Goals

At the beginning of this paper, there were three specific goals that this project focused on.

The first goal, to give access to wholesale offers for small and medium businesses, has been accomplished successfully. After the implementation of this project, many tipping offers were sold which resulted in buyers paying very low prices. These buyers would have never been able to obtain these prices before this platform and concept existed.

Secondly, the user base of the platform was expected to grow substantially. This would be achieved by the potential sharing factor of tipping offers. When a successful tipping offer was active, the user growth peaked. It did not reach the growth expected but it definitely improved.

Finally, we wanted to solve a problem that many merchants have. We wanted to provide a quick release for their accumulated stock. The general users' profile in *Rewardli* in the present day is basically related to the Startup environment and the software development community. Because of that, most of the tipping offers published in the platform were based on software or services that did not have a material stock. Therefore we never had the chance to face this need.

One of the main purposes behind this paper was to provide an insight of the development process of the product, from the initial stage and design, to the different versions accomplished. I can finally say this paper has been able to illustrate this desire.

Despite the goals seeming slightly ambitious, I am very satisfied that most of them have been successfully achieved.

8.2 Future Extensions

This project is just the beginning of the tipping offers concept. The product will probably be applied to new markets, evolve its concept or even be the base for new ideas.

Some new features can be added to automate the process of creating the tipping offers. Instead of being the administrator's responsibility to agree and register the tipping offers, the flow could be open to merchants. The merchants would sign up to the platform and post their own tipping offers. In this case, the site would actually behave more as a marketplace where small and medium business would meet together with vendors.

On the other hand, one variation worth mentioning that the company has been thinking about, is to apply the tipping offers concept to subscription products. Not only we would sell products with a one-time price, but products with recurring costs as well.

The concept changes a bit but it is very similar to the tipping offers. Using a monthly subscription product, companies can buy same product/service for the same price each month. At the beginning of the month, the price can go down if there are enough subscribers to the offer. This strategy can perfectly group businesses and leverage their buying power, just as it happens with the tipping offers presented in this project.

Finally, I would like to point out that the most important thing in a startup is to prioritize. A startup has very limited resources so it is crucial to put the focus in specific goals and not try to achieve everything at once. As we have seen, there are many different paths and possibilities to extend the product. The next step would be to analyze those and decide if any of them is worth of receiving all the attention of the company.

8.3 Personal Evaluation

Writing this final personal evaluation I realize that the process of developing this project has been long but definitely worth it. Although I initially thought that the final degree project would only be a heavy workload, it has helped proved to me the knowledge and skills I have acquired during these five years at university.

One of the hardest things that I had been struggling for a while was deciding how to organize this project. At the beginning, I got carried away by the idea of writing the paper with the traditional structure. Due to the habit it was very hard for me to imagine it in a different way. However, I could realize soon enough to redirect it into the correct direction. There is one thing that I would probably change if I could go back. Basically the use case methodology in this paper, was not actually used during the development of the project. Instead the team used user-stories. Despite the fact that they both represent the same thing, I would have ideally followed the exact process used in the company.

I would like to emphasize the methodology of the company as one of the most rewarding and learning experiences throughout the development of this project. Getting exposure to the startup environment, best practices, latest technologies, working side-to-side with an awesome team, I am not able to think of a better way to consolidate the education received in the Barcelona School of Informatics.

For this, and everything that is to come, I would like to thank everybody that has given me the smallest push to be where I am right now: *Rewardli* for welcoming me in this great company, *Stepone* for investing in me with this great chance, Barcelona School of Informatics for letting me develop this project from the distance and the rest of you for your help and support.

9 Bibliography

9.1 Books

Dolors Costal, M. Ribera Sancho, Ernest Teniente.
Enginyeria del Software: *Especificació de sistemes orientats a objectes amb la notació UML.*
Edicions UPC, 2003. ISBN 84-8301-727-X.

Eric Ries
The lean startup: *How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*
Edition 2011

Jason Fried, David Heinemeier Hansson
Rework
Edition 2009

Jason Fried, David Heinemeier Hansson, Matthew Linderman
Getting Real: *The smarter, faster, easier way to build a successful web application*.
Free online version: http://gettingreal.37signals.com/

Paolo Perrotta
Metaprogramming Ruby: *Program Like the Ruby Pros*
Edition 2010

9.2 References

Ruby Libraries Documetation http://ruby-doc.org/

Ruby on Rails http://rubyonrails.org/documentation

Railsonfire Blog http://blog.railsonfire.com/

Heroku http://www.heroku.com

10 Glossary

agileZen: a simple, flexible, and cost-effective web-based software for project management built on ideas from agile, lean, and kanban methodologies

Campfire: a simple web-based real-time group chat tool for business

CSS (Cascading Style Sheets) is a style sheet language used for describing the presentation semantics (the look and formatting) of a document written in a markup language

Git: a distributed revision control and source code management system with an emphasis on speed

Git Flow: Git extensions to provide high-level repository operations for Vincent Driessen's branching model.

Google Analytics: is a free service offered by Google that generates detailed statistics about the visitors to a website

DOM (Document Object Model) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents

HAML (HTML Abstraction Markup Language): is a lightweight markup language that is used to describe the XHTML of any web document without the use of traditional inline coding

Heroku: is a cloud platform as a service (PaaS) supporting several programming languages

HTML (HyperText Markup Language): is the main markup language for displaying web pages and other information that can be displayed in a web browser

JavaScript: is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions

jQuery: a cross-browser JavaScript library designed to simplify the client-side scripting of HTML

MVC (Model-View-Controller): The **model–view–controller** framework separates the representation of information in a computer program from the user's interaction with it.

PostgresSQL: is an object-relational database management system (ORDBMS) available for many platforms

Railsonfire: a continuous deployment service for applications

Rspec: is a behavior driven development (BDD) framework for the Ruby programming language, inspired by JBehave

Ruby on Rails: is an open source full-stack web application framework for the Ruby programming language

SASS (Syntactically Awesome Stylesheets) is a stylesheet language initially designed by Hampton Catlin and developed by Nathan Weizenbaum

SCRUM: an agile software development method for project management

SEO (Search Engine Optimization): s the process of improving the visibility of a website or aweb page in search engines' "natural," or un-paid ("organic" or "algorithmic"), search results

SQL: is a special-purpose programming language designed for managing data in relational database management systems (RDBMS).

Statowl: It is a web search engine to search about information and statistics about the World Wide Web.

SVN (Apache Subversion): is a software versioning and revision control system distributed under an open source license

W3C (World Wide Web Consortium): is the main international standards organization for the World Wide Web (abbreviated WWW or W3)

11 Annex

11.1 Demo

The best way to see the results is definitely visiting directly to browse the rewardli website at *http://www.rewardli.com*. Sign up is free and easy so feel welcome to try it out! Some features are specific to groups and will not be accessible though and the administrator will obviously not be accessible.

However, it is of my interest to show some of the visual results in this paper. Here in this section some screenshots of some functionalities developed throughout this project will be described to provide a view of the results.

11.1.1 Tipping offer page

This is an example of a real tipping offer sold in December 2012. In the top we can see the price scales with the current price highlighted. The rest is all information about the offer and a big call to action to make an order .

rewardli

Pol Omella
My Account | Logout

Deals > Marketing > UserTesting.com

Status: active Edit Delete

The fastest way to find out why users leave your site or mobile app
5 user tests from UserTesting.com - * DEAL EXTENDED AND PRICE LOWERED *

Tipping Point beta
Price decreases the more people buy notify me

1 - 4	5 - 9	10 - 19	Current	50 - 200
~~$140~~	~~$120~~	~~$100~~	$75	$50

Buy Now $75

Value	Discount	You Save
$195	62%	$120

30 purchased.

Tweet Send Like

Common Questions
How does the price work?
What if the price comes down after I buy?
When can I redeem my purchase?
Can I buy now and redeem later?
Is there a guarantee?

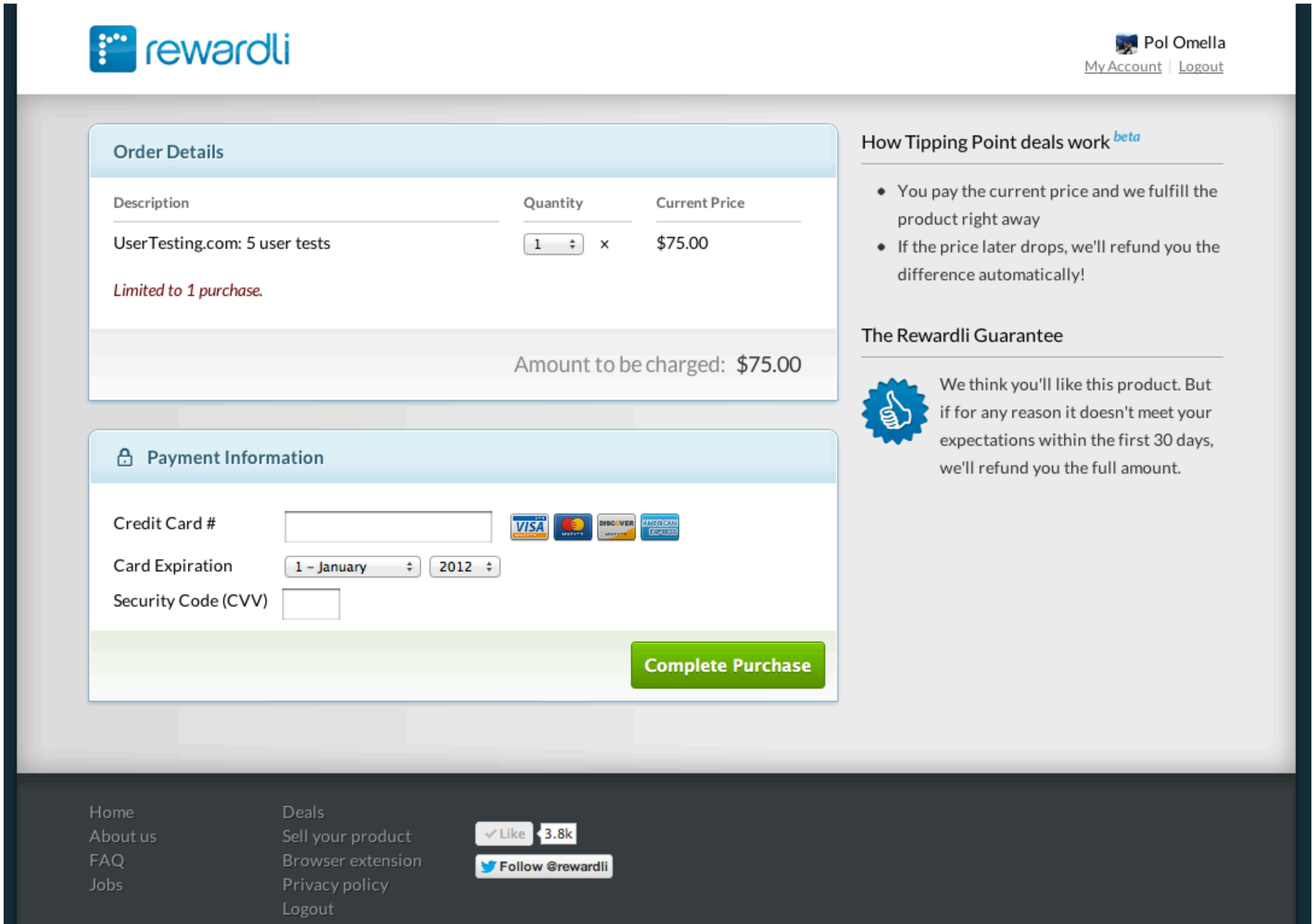
UserTesting.com
On-Demand Usability Testing
00:00 / 02:45 HQ

What you get
- 5 user tests from individuals who meet your targeting criteria
- Videos of users interacting with your site with a recording of their voice (~ 15 min. each)
- Written feedback and responses to your questions

Terms
- Credits valid for two years from date of purchase. Use them all at once or separately.
- One purchase per user
- Valid for new and existing customers

11.1.2 Payment page

The next page is the actual order process. It shows a simple summary of the order information and customization of the quantity that the shopper wants to purchase. After that the billing information to proceed with the payment.



11.1.3 Order administrator page

This is the administrator of the platform. The following page is the page where the administrator can manage and track all the orders performed in the system. We can notice the different options to proceed with possible refunds or cancelations.

rewardli   Logged in as **polmiro@gmail.com** | Logout

Activity | Offers | Content | Emails | Payments | Users | Metrics

All activities | Clicks | Visits | Invites |

Cashback Purchases | **Orders** | Offer Purchases |

Genenéral subs. | Group subs. | Offer subs. | Gifts Invitations | Reviews | Questions |

Orders

Date range:

Go

Merchant	User	Qty	Sale	Overcharged	Purchased at	Refunds	
Grasshopper	[illegible]	1	[illegible]	$0.00	[illegible] 12:22	$0.00 (0)	view Cancel
Preferred Return	[illegible]	1	[illegible]	$0.00	[illegible] 12:11	$0.00 (0)	view Cancel
inDinero	[illegible]	1	[illegible]	$0.00	[illegible] 15:48	$0.00 (0)	view Cancel
AwayFind	[illegible]	1	[illegible]	$0.00	[illegible] 17:16	$0.00 (0)	view Cancel

Add an order

11.1.4 Administrator new tipping offer page

Continuing with the administration part, this is the form page to create a tipping offer. Notice that the tipping offer is a very complex entity (and nested entities). The shown screenshot does not fit the whole form. It actually continues with some other information (redemption methods, prices, etc).

rewardli
Logged in as **polmiro@gmail.com** | Logout
Activity | Offers | Content | Emails | Payments | Users | Metrics
Hot offer | Cashback offers | Tipping offers | Custom offers

New Tipping Offer

Merchant 99designs create a new merchant

Title *Description of the offer used at a lot of places.*

Headline *Headline used on the offer page.*

Sub headline *Sub Headline used on the offer page just under the headline.*

Summary *2 sentences description of the offer.*

Description *Detailed description of the offer.*

Paragraph

Path: p