TAMPERE UNIVERSITY OF TECHNOLOGY

ALEJANDRO YUSTE MASCARÓ

SIMULATING OPEN SOURCE SOFTWARE COMMUNITIES THROUGH COLLECTIVE GAMES

Masters of Science Thesis

Examiners: Associate Professor Imed Hammouda and Msc. Terhi Kilamo
Examiners and topic approved in the Faculty of Computing and Electrical Engineering Council, meeting on 03/04/2013

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
YUSTE MASCARÓ, ALEJANDRO: Simulating Open Source Software Communities Through Collective Games
Master of Science Thesis, 56 pages
February 2013
Major: Software Systems
Examiner: Associate Professor Imed Hammouda
Examiner: Teaching Associate Terhi Kilamo
Keywords: Open Source Software, Open Source Software Communities, Simulating, Collective Games

According to the Open Source Initiative, Open Source Software (OSS) can be defined by ten criteria. The most important and relevant ones are the free redistribution of the software, the inclusion of the source code and the authorization to modify and redistribute the work. OSS products are a vital part of how we understand Internet. But, for most people, it is still complicated to understand what is an Open Source Software Community. In this thesis, we have analysed how these OSS communities work, how they are structured and how they get the results that made them popular. Furthermore, a tool that simulates many of the features of OSS communities has been implemented. This platform permits a user to feel how is joining one of these communities and working with other community members to solve a complex problem through collaboration.

This thesis has allowed us to remark the importance of collective games in simulating the dynamics of OSS communities. These communities are formed by members who have to come together to develop a product. Thus, the notion of collaboration is essential; as in the collaborative games where the players have to cooperate to reach a solution. This project also helps us illustrate the collective games approach through the Sudoku game, which is the game chosen to develop the simulation platform. To perform it, we have used intelligent agents which roles are to act like members of a real community. The result is that a human user can join it and play in different roles to understand the operation of OSS communities.

PREFACE

This Master of Science Thesis has been undertaken at Tampere University of Technology (TUT) at the Faculty of Computing and Electrical Engineering from October 2012 to March 2013.

Once finished, the platform produced is available on-line for all those people who want to check it or even collaborate to improve it. The web page of the project is the following:

http://alejandroyuste.github.com/MudokuWithAgents/

I would like to thank Imed Hammouda for assigning me this thesis giving me the opportunity to come to Tampere and enjoy these ten months abroad. Also, I would like to thank all people from TUT Open Source Research Group for the good environment of work I had and all the things they taught me, I am sure they will be very useful in the future.

Alejandro Yuste Mascaró
Tampere, March 22, 2013

CONTENTS

NOMENCLATURE AND ABBREVIATIONS

ABM	Agent-Based Model
CSS	Close Source Software
FSF	Free Software Foundation
OS	Open Source
OSC	Open Source Community
OSS	Open Source Software

1. INTRODUCTION

During the last decades, technology has increasingly become one of the most important parts of our lives. It has changed habits human beings have had for centuries and nowadays we are totally dependent on this progress. One of the main reason for this tendency is Internet. The network of networks has altered the way we perceive and understand the world [1]. For example, people feel today closer than before, even if they live in different countries. Also, we can know every sort of news only a few seconds after they happened. It would not have peen possible 25 years ago.

How did we arrive to this point? Who are responsible of this situation? Internet has been created to connect computers and, accordingly, people. It was thought to share information and data by some users/developers who felt the necessity to solve a lack of solutions. And this is still the key of the Open Source Communities, to find an answer to any kind of questions through Internet.

Contrary to what may seem, the way we understand Internet is thank to the work of this type of Communities. Almost 65% of Internet servers are working with a Linux kernel [2] and about 60% of web servers are Apache [3]. Also, the Linux kernel is present in fridges, washing machines, smart phones and multiples devices we use daily. We can find Open Source Software (OSS) communities products everywhere, but how these communities really work? Who are their members? How are they organized?

1.1 Motivation

One of the main problems potential members might have regarding OSS communities is just before joining them. Most of new users are scared to share their work thinking it is not good enough for the project or it could contain bugs. This fact creates a barrier for those people who really want to join them but do not know how they are organized and what is the software development process in this kind of communities. Additionally, the establishment of the relationship with the community may be tough, people do not usually like to share their thoughts and work through the Internet.

Potential new members could also have many communication problems. As they might not understand the different roles present in these communities, they could ask for their issues to wrong members or they could try to do a work they are not ready for. Besides, it is not an easy work to learn about a concrete community. Normally, they are formed for many users who are working in different parts of a project and discussing about many different facts about it at the same time. All these information could be exasperating at the beginning and it is for this reason they need a platform to help them knowing where to start on an OSS community.

1.2 Objectives

In this master of science, we will try to understand how OSS Communities are structured, which motivations they have and how they work. Then, the issue will be treated as a Serious Game [4]. This is a simulation of a real-world event or process designed for the purpose of solving a problem. This kind of games always has an educational purpose. A user is playing to reach an objective but what he/she is really doing is solving a complex problem.

In this way, this last concept will be applied over collectives games. These games are collaborative games where many players have to share their knowledge to reach a goal. Thus, we will study the significance of collective games in simulating the dynamics of Open Source Software Communities. Concretely, we will be trying to illustrate the collective games approach through the Sudoku Game.

Before doing this thesis, there were questions about whether it was possible or not to simulate an Open Source Community (OSC). Thus, one of the objective was to succeed in this performance. After a few months, we achieve the creation and the development of a simulation tool. It imitates the different roles present in an OSS community to help the users know the different functions that are part of it.

1.3 Structure of Thesis

The thesis is divided in seven different chapters. After the introduction, in Section 2, a brief review about the history of Free Software can be found, as well as the origin of the term Open Source Software and an analysis of Open Source Communities in order to know what is going to be simulated. In Section 3 is explained how we have simulated our OSS community and how the simulation works applied to the concrete Sudoku case. Then, in Section 4, the details of the previous tool we have used to develop our own are precised, focusing in the parts that have been more important in our tool. Section 5 details all the features of our own platform, with all the changes we have done to the previous tool and the new characteristics that have been developed. In Section 6 can be read the analyses of the results and the goals as well as the next investigation steps and the limitations of our own tool. Finally, in Section 7 can be seen the validation of the tool in addition to the conclusions we have obtained from the whole process.

2. BACKGROUND

From the first pieces of software until the digital age, about 1950-1960, computer programs used to be developed by single developers. The process consisted in punching little rectangular holes in (decks of) cards that were read by computing machines without operating systems [8]. After this, almost all software was produced by computer science academics and corporate researchers working in collaboration. It is not until the invention of the integrated circuit that the software developing companies began to hide the code to sell the executables under restrictive licences. Then, this business model became the general rule.

2.1 Free Software

The story begins in 1971 when a physics student called Richard Stallman enrolled as a graduate student at MIT. In 1980, he and some other hackers at the AI Lab were refused access to the source code for the software of a newly installed laser printer. Stallman had modified the software for the Lab's previous printer, so it messaged a user when the person's job was printed, and would message all logged-in users waiting for print jobs if the printer was jammed. Not being able to add these features to the new printer was a major inconvenience, as the printer was on a different floor from most of the users [9]. This experience convinced Stallman of people's need to be free to modify the software they use and laying the foundation of the Free Software Foundation.

2.1.1 Free Software Foundation

The movement was born on September 1983, when Richard Stallman announced the GNU Project. This was a mass collaboration project whose aim is to give computer users freedom and control in their use of their computers and computing devices, by collaboratively developing and providing software that is based on the freedom rights. Two years later, in October 1985, he created the Free Software Foundation, initially to raise funds to help develop GNU. All these years the institution has been working to remove copy, distribution and modification restrictions for all the software developed. In February 1986, FSF published the definition below for free software [10].

"The word 'free' in our name does not refer to price; it refers to freedom. First, the freedom to copy a program and redistribute it to your neighbours, so that they can use it as well as you. Second, the freedom to change a program, so that you can control it instead of it controlling you; for this, the source code must be made available to you."

The foundation has never lost the initial idea of developing the GNU project. However, nowadays the institution is also working in other fields as the GNU General Public License (GPL) and its derivatives, a widely used license for free software projects. It has a FSF's publishing department called GNU Press which is responsible for publishing affordable books on computer science using freely distributable licenses. The Free Software Directory is a listing of software packages that have been verified as free software. FSF also keeps maintaining the Free Software Definition and many documents that define the free software movement. It provides the project hosting for software development projects on their Savannah website. It promotes many political campaigns against what it perceives as dangers to software freedom, including software patents, digital rights management and user interface copyright. It presents the awards: "Award for the Advancement of Free Software" and "Free Software Award for Projects of Social Benefit" for the few people or groups of people who deserve it the most every year.

According to the definition of FSF, Free Software is every one that follows next four freedoms [11]:

Freedom 0: The freedom to run the program for any purpose.
Freedom 1: The freedom to change how the program works to make it do what you wish.
Freedom 2: The freedom to redistribute copies so you can help your neighbour.
Freedom 3: The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits.

2.1.2 Open Source Software

The Free Software represents all the philosophy of the movement, but it is not valid as a business model because of the use of the word "free". For this reason, in 1998, Bruce Perens and Eric S. Raymond created the Open Source Initiative. They advocated that the term Free Software should be replaced by Open Source Software [12] as an expression which is less ambiguous and more comfortable for the corporate world. To understand the differences between the two organizations the FSF published [13]:

"The term "Open Source" software is used by some people to mean more or less the same category as free software. It is not exactly the same class of software: they accept some licenses that we consider too restrictive, and there are free software licenses they have not accepted. However, the differences in extension of the category are small: nearly all free software is open source, and nearly all open source software is free."

Thus, we can say both types of software are more or less the same, differentiating only in the legal part. We can observe in Figure 2.1 how all software are classified and which one can be downloaded for free.

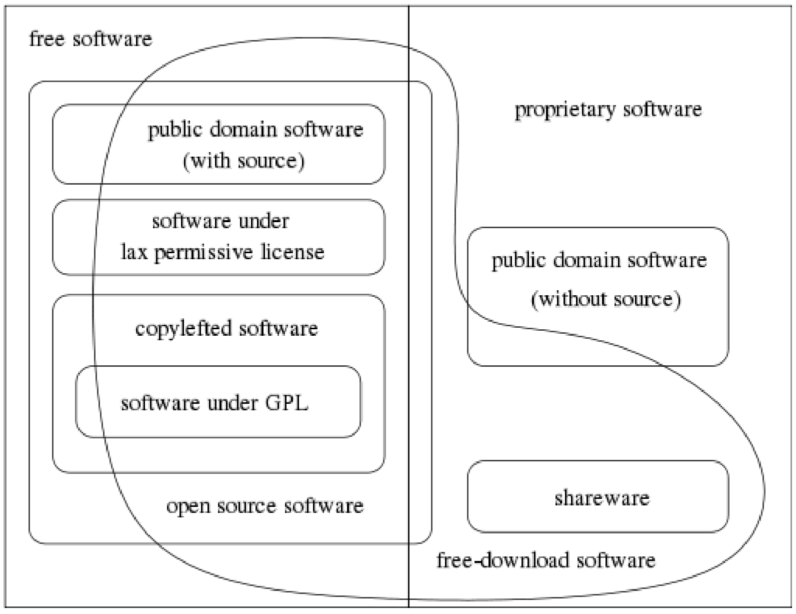


Figure 2.1: Classification of Software

2.2 Open Source Communities

A community is a group of people with a shared interest, purpose, or goal, who get to know each other better over time [14]. The final goal or shared interest of an Open Source Community is to produce quality "free" software in a very different way that the privative software. Traditionally, in companies dedicated to develop software we find a restricted work team who create the software and only release the compiled version. With this strategy, they permit that nobody really knows how the program runs and that only them can fix the bugs, maintain and improve the software.

The difference stands in that in open source software the code is very often developed in a public collaborative manner. This means that the members of a community cooperate to create a product or improve an existing one. It could work as follows: a single user or a group needs a new characteristic for any reason, so he/she/they decide to create a product or improve one existing, then a project is published on Internet and any person interested can join them to do any task related. Finally, the code is released with the compiled version. Anyone can check, modify and then release it again in the same way.

2.2.1 Members

But, which kind of people join these communities? Studies show that about 98,5% of them are men [15], the age is from 16 to 25 year old, 60% of them are in relationship and 83% of them are related with the IT sector [16].

Based on these studies we can say that these people are students or workers in any IT sector and that they dedicate their free time to develop projects which motivate them. High chances are that they have a certain level of education (most of them are bachelor's students) and that they enjoy exploring the limits of what is possible, in a spirit of playful cleverness [17]. Members join communities for two principal reasons: it is the quickest way to achieve a goal and they receive recognition from other users. The entire organization of these communities is based on the last principles.

Time is very important for the members, as they normally do not receive any salary for their job in the communities. Thus, there are published protocols and standards to waste as little time as possible in the different steps of developing. For example, when a developer has an issue, before asking to the community, he must read the manuals, look for similar problems in the own community, search for it in every web page and if the person still has any doubt, then he finally can ask to the correct people in the community. The prestige refers to the fact that every programmer likes to show his skills and do a good job to help the others. Because of this, the communities are structured as a meritocracy; the person who is working the best has more prestige and is more acknowledged in the community. So, this person would work in more responsible positions and his opinions would be more listened when the community has to make a decision.

2.2.2 Characteristics

Communities have changed some traditional work concepts; they have developed some tools to modify the way they work compared with proprietary software development. Open source communities work in a really different way than traditional development software models [19].

The attribution concept is substituted for recognition. When a programmer provides a part of software, it belongs to the community, but the rest of the members recognize that every part has been created for a particular member. The final product is formed from a lot of contributions by each developer and every Internet user can check any part and modify it, improve it, etc to make the final product even better. This suggests that the final product is the result of a collective intelligence. The sum of each small created part is setting a collective knowledge that is growing with every new contribution.

Replacing the synchrony by asynchrony. As they work through Internet, the members do not need to work in the same place or at the same time. It implies that there should not be an exact dateline to make decisions or finish a concrete work, and that all members can submit a part when they want and if a problem is found, then it can be studied.

Replace the opacity to transparency. At this point, we find two types of transparency:

- Organizational and operational: All the levels of the community have the same information which reduces wasted time and makes every people feels of equal importance.
- Systemically: In the proprietary software, the security is based on the information hidden. As nobody can check the source code, it is supposed that nobody knows how it runs. However, it might not have a security layer. Working with inverse engineering could help figure out what is the code. In contrast, in open source software, if everyone can check the source code, the code is designed to be safe.

Change exclusivity for inclusivity. The control and testing of the software is made by all the members and users of the software, not only by a limited work team where only a few of them can check the code. This is very important because the work can be done in a short time and as more people are involved, more bugs are fixed and better will be the result. "Given enough eyeballs, all bugs are shallow" [18].

Move from an autocracy to a meritocracy. More prestigious is a member, more weight this person has in the important decisions of the community. The significance of this is that the prestige becomes a dynamic concept. If an important member takes a wrong decision, next time his/her opinion is less taken into account. The aim of this system is to find the efficiency through good work. For this reason, the power is not always in the same hands. If a member is working well and the community runs well, this member receives prestige and more responsibility. This person does not give order to the others but on the contrary, this member must obey to the rest of the community . This means that if this person does not follow what the community feels is necessary, the community will not trust him/her anymore and they will follow another prestigious member.

The community has the conscience they are working for a product under construction, never a final result. Although, they often have to release stable versions for the users. They handle a version system; this means they do not try to control completely the software, they coexist with chaos. This is due to the fact that it is impossible to control every detail, so they try to make the whole product works.

2.2.3 Roles

As we can see in Figure 2.2, the communities are formed by different types of roles with an onion model which is necessary for a healthy software production.

In the center of the onion we find the project Leader. This person is often the one who has initiated the project. The main role is to be responsible for the vision and the overall direction of the project. In the next layer there are the Core Members. They are responsible for guiding and coordinating the development of the OSS project. These members have been involved with the project for a relative long time and have made significant contributions to the development and evolution of the system. In those OSS projects that have evolved into their second generation, a single Project Leader no longer exists and the Core Members form a council to take the responsibility of guiding the development.

The next layer is for the active developers, who regularly contribute to new features and fix bugs; they are one of the major development forces of OSS systems. Very similar to the last ones are the peripheral Developers. This is the people who occasionally contribute to new functionality or features for the existing system. Their contribution are irregular, and the period of involvement is short and sporadic.

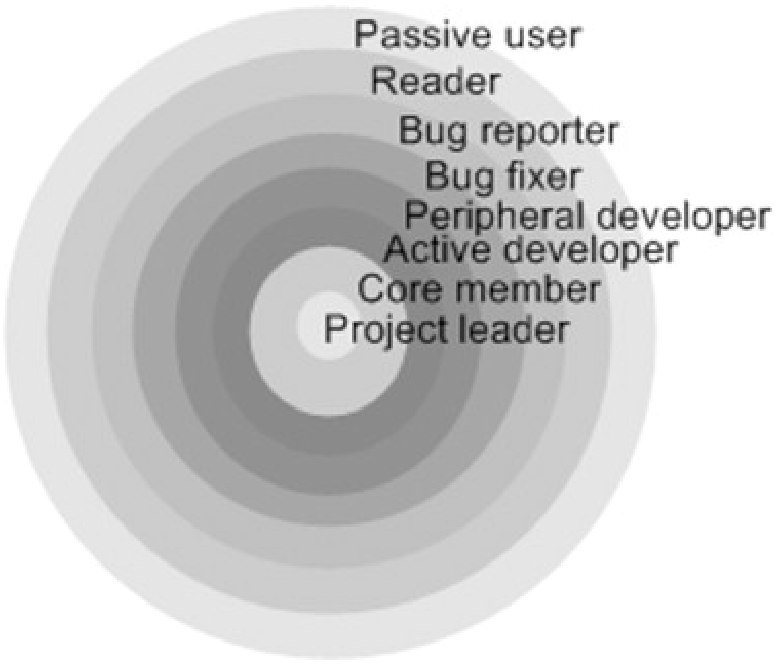


Figure 2.2: Onion model of the roles of the community

Then, there are the bug fixers. These members fix bugs that either they discover by themselves or are reported by other members. Bug Fixers have to read and understand a small portion of the source code of the system where the bug occurs. The people who discover and report bugs are the bug reporters; they do not fix the bugs themselves, and they may not read source code either. They assume the same role as testers of the traditional software development model.

In the borders of the onion we find the least influential roles. The readers are active users of the system; they not only use the system, but also try to understand how the system works by reading the source code. And finally there are the passive users. These members just use the system in the same way as most of us use commercially available Closed Source Software (CSS).

As OSS communities work as a meritocracy, the different members are promoted or downgraded depending on their work accomplished in the community. A member might also belong to different roles. For example, a contributor could also be a bug reporter.

2.2.4 Decision Making

The necessity of making decisions is clear, the projects have to be governed in some directions and there are always problems to solve. However, not all the communities are working in the same way. Usually, this point depends on the size of the community:

Small Communities. These communities are usually governed by a single leader. This person, who often is the first owner of the project, is in charge of taking all the decisions related to the community: which contributions are committed, what is the direction of the project, how will be the user interface, etc.

Medium Communities. These communities do not present only one single leader, but single leaders for all the different parts that form the complete community. We can say the community is formed by little self-managed communities working on common objectives.

Large Communities When it is not possible than a single person controls all the facts of a community, they are governed by a group of developers, usually the core developers. To make decisions, these communities use a consensus where an unanimous vote is not required to reach a final resolution. However, the people who disagree usually accept the conclusion because trying another solution on their own would take too much time. Nevertheless, when there are very different solutions and many people supporting each of them, we can find the following scenarios:

- Passive denial: It is the case when there are some people who do not accept a solution, but they are still included in the project. They would not participate in the implementation of that part, but they would still enjoy the final product.
- Implementation of two different solutions: this situation could happen when in a concrete moment is decided to implement two different solutions, and finally choose only one of them. It doubles the time of implementation compared to only one solution,so for this reason, it should be avoided.
- Forking: It means the creation of two different projects based on the original one. This suggests that the two projects will never be again part of the same, resulting in new alternative projects. This must also be avoided. When the efficiency of the whole process is thought about, the forking is a very bad solution. It should be always the last solution for a problem.

The concept of a community is dynamic and there is not only one pattern to describe them. We have tried to generalize the concept and explain the most commons facts that we can observe. However, some communities could be very similar to the teams in the corporate world and others a complete mess where nobody knows exactly which is his/her work.

2.3 Problem

The utility and the excellent results of OSS communities has been proven. Yet, many users are not willing to be part of the experience. The reasons are numerous. Among them, we could find the fear of sharing their knowledge, the fear of not being good enough and the ignorance of the operation of these communities.

There is currently no studies or no interest for that matter. The users have to learn by doing. When they first join a community, they will certainly make many mistakes, but this is the only way to be trained so far. This is why we have decided to create a platform that will help the potential new members understand the performance of the OSS communities.

Another issue that might occur is the way the information are exchanged. The core of a project of an OSS community is the communication among the members. Usually, this is done through mail lists or forums. The problem of this method is the users are not familiar with the idea to share every piece of information with one another. The project we have conducted aims at developing trust in the OSS communities by creating a platform without emailing nor forums. The information needed can only be seen on the platform at any moment.

3. THE COLLECTIVES GAMES APPROACH

As it has been explained before, the simulation has been done to solve a Sudoku problem. But the objective is not to solve a Sudoku, it is to observe how an OSC works. Thus, in this chapter is explained the Sudoku problem and the characteristics and limitations for this kind of simulation. Also, it is described how the simulation has been done and the necessary facts that make it works.

3.1 Collectives Games

A collective game is a play where a single participant is not competing against other users, but on the contrary, all of them have to cooperate to reach the goal of the game [7]. Cooperative means that there is scope for working together, but not all are rewarded/punished equally for doing so and the aim of the game is still to be the best. Collaborative games only reward collaboration, and all players gain or suffer equally. Winning one of this games is possible with good luck and careful resource management but is more likely through good collaboration. Specifically, active communication amongst the players and timely sacrifices for the good of the group are the keys.

A collective game follows the next rules:

- Hardness: The game must be combinatorial in nature and thus requiring search. Technically speaking, these games must be NP-Hard.
- Adjustable hardness levels: We can easily and in a progressive manner adjust the hardness of the instances of these games.
- Collaborative solution building: The solution to these games could be constructed in a collaborative manner but may require conflict resolution.
- Decisions as discrete choices: The solution to such games can be achieved by making decisions among a number of possible choices.

In the same way, by analysing the game when it is played to reach a goal, each member should follow the next four lessons:

Lesson 1. To highlight problems of competitiveness, a collaborative game should

introduce a tension between perceived individual utility and team utility.
Lesson 2. To further highlight problems of competitiveness, individual players should be allowed to make decisions and take actions without the consent of the team.
Lesson 3. Players must be able to trace payoffs back to their decisions.
Lesson 4. To encourage team members to make selfless decisions, a collaborative game should bestow different abilities or responsibilities upon the players.

3.2 Simulating an OSS Community

There are many features to simulate an OSS Community. For example, you could change the behaviour of the different roles or you could modify the information system. We have tried to simulate the whole behaviour of the community, synchronizing the different roles. The work that has already been done and that is closer to what we have done is an agent-based simulation. It has succeeded to do a prototype of the development movement in OSC. They map one of the biggest OSS communities in SourceForge as a network and then, they simulate the evolution of the network calibrating it with empirical data [23].

The simulation has been implemented using an Agent-Based Model. This model, very used in all types of simulations, permits to model a system as a collection of autonomous decision-making entities called agents. Each agent individually assesses his/her situation and makes decisions on the basis of a set of rules. These agents may execute various behaviours appropriate for the system they represent [6].

In this way, our simulation is based in a system of agents and the relationships between them. Specifically, we have implemented the behaviour of some of the different roles we can find in an OSC. Thus, we have obtained a complex system which never behaves in the same way. Depending on the morphology of the agents present in the community we get good or bad results where a good result is a correct solution of the game in a short time.

The game starts when the server is launched. At the beginning the server initializes a random game with 40 values which will be untouchable the whole game. Then, any number of agents or users can be connected and begin to interact among themselves to solve the problem.

3.2.1 Sudoku Case

The Sudoku game has been chosen as a concrete problem to apply to our simulation of an OSS community. This game follows all the rules mentioned above, but also it permits to see the game as an abstract system which is composed of three different subsystems, on the grid: Rows, Columns and Squares. This means that to achieve a correct solution the three subsystems have to be correct and all the members in each subsystem have to collaborate to get a final good result.

The predecessor of Suduku appears for the first time on July 6, 1895, in the newspaper *Le Siècle's rival, La France* and it was almost a modern Sudoku. We could find inside the journal the 9x9 square puzzle in which each row, column and broken diagonals contained numbers from 1 to 9, but did not mark the sub-squares. Although they are unmarked, each 3x3 sub-square does indeed comprise the numbers from 1 to 9 and the additional constraint on the broken diagonals leads to only one solution [21].

The modern Sudoku was designed by Howard Garns, a 74-year-old retired architect and freelance puzzle constructor from Connersville, Indiana, and it was first published in 1979 by Dell Magazines as Number Place. The current name appeared when it was introduced in Japan by Nikoli in the paper Monthly Nikolist in April 1984.

Sudoku is a puzzle game whose main objective is to fill up a grid with natural numbers, normally from 1 to 9, respecting the following rules:

- Rows: Each number in each row has to be different.
- Columns: Each number in each column has to be different.
- Squares: Each number in each square has to be different, where a square is a a sub-part of the grid. Traditionally, we find 3x3 squares located orderly.

The advantage of this game is that it can be seen as an abstract system formed by three different abstracts subsystems, one for every rule that determines the game. In this way, the game is finished only when all the subsystems are correct. In our simulation, we have used a 16x16 grid with 1-16 possible values. In Figure 3.1 we can observe how the grid we have developed looks like. In it, the squares are marked by thicker stripes and every time a cell is selected the scope of this value is coloured differently, doing it more user-friendly.

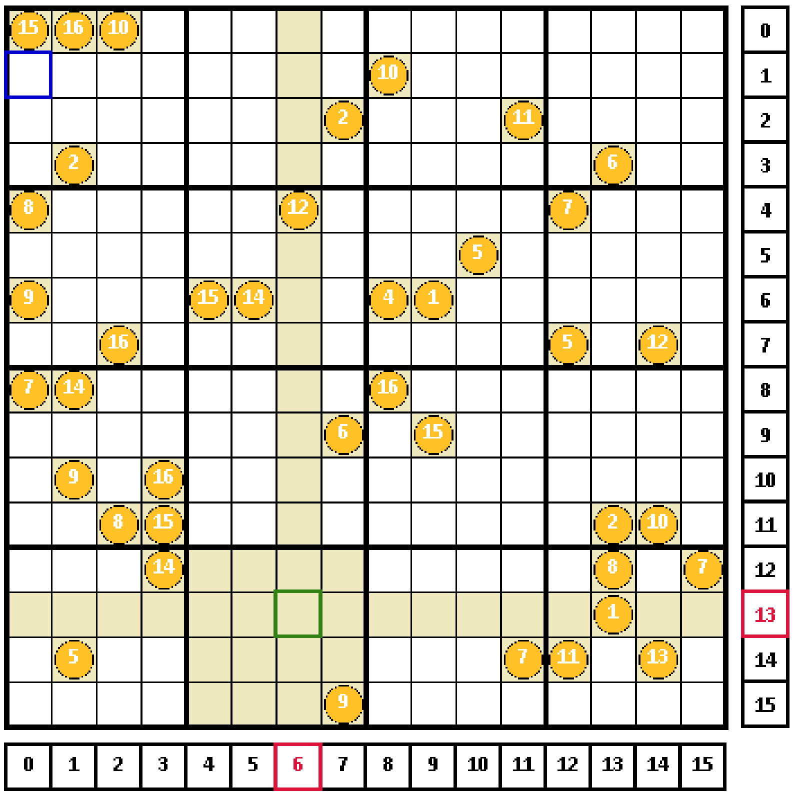


Figure 3.1: Grid with a Sudoku Problem initialized

3.2.2 Roles Simulated

The agents represent one of the most important parts of the game. The simulation can be done with real users, but a community of agents launched allows the simulation to be totally observable and more conclusions can be extracted. In the following list can be found all the roles simulated and how they work.

The *contributor* agents only participate in the community adding values to the grid. As each one of them belongs to one of the subsystems there are three types of contributors. In this way, a row contributor only adds values that are correct by rows, with a 5% possibilities of error. This means that when an agent fills a cell, the value is probably correct in one of the subsystems, but it may be wrong in the others.

There are also three types of *Bug Reporters* depending on the subsystem. Their work is to check if the values contributed by the previous role, that do not belong to the subsystem, are correct or not. For example, a column bug reporter checks the values contributed by row and by square contributors, assuming that the contributions by columns are correct. When one of these agents detects that a value is not correct for his/her subsystem he/she sends a report bug to the server and this position of the grid will be available to be filled again, marking that a bug has been detected.

A *Tester* works in a very similar way than a Bug Reporter, but only on correct values. There are also three types of them and they only participate in the community when a value of another subsystem is correct for their own subsystem. Then, they ask for committing this value assuming it is correct at least in two subsystems.

A *Committer* agent only participates when a voting session is active. Its work consists in emitting a vote to decide if a value will be committed or not. Each agent has a 15% of possibilities to vote contrary to its interest. In any case, one incorrect vote is not going to change the final result of the voting, as the values voted should be correct in two of the three subsystems. The correct values will probably be committed.

Unlike the other groups, there is only one type of *Project Leader*. This member knows all the rules of the game and he/she unifies the three subsystems. He/she collaborates in the community accepting or rejecting the committed values and deciding if they are suitable for the final solution or not.

In Figure 3.2 we can see an example of how the agents work differentially on a Sudoku Grid.

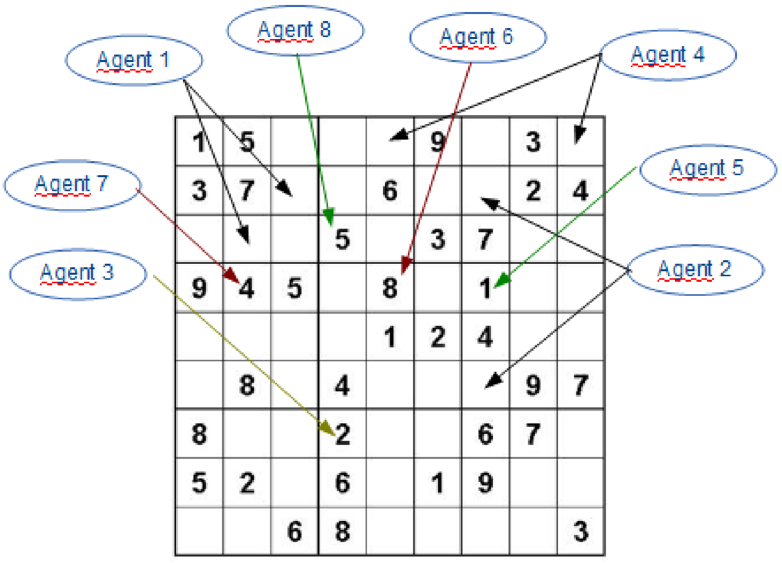


Figure 3.2: Example where different Agents work in the same Sudoku. Black lines represent the contributors, red lines the testers, green lines de bug reporters and brown lines the leader

3.2.3 Simulating an OSS Communities with the Sudoku Game

To simulate an OSS community on a Sudoku game the features of the community has to change to adapt to the possibilities the game offers. Thus, we have modified the most important characteristics of the communities.

Contributions. Whenever a member of the community makes a contribution, there will be a possible number added in any position of the grid. This number might not be correct, for this reason, other members will analyse it and decide if it has to be accepted or if it is a bug.

Bugs. When an agent or user detects a number which might not be correct in the position it has been contributed, it will be said that a bug has been detected. Once reported, the position in the grid will be free again to be filled with another number by another member.

Testing. A value is been tested correctly when a member of the community asks for committing it. It would mean the value has been tested by a different member and he/she has estimated that the value is correct in that position, although the number could still be incorrect.

Committing. When a value is asked for committing there is a voting session with all the committers members who decide if a concrete number is committed or not. As the agent committers only know about one of the subsystems of the game, they could vote incorrectly. When a value is committed, it means than it will be presented to the project leader to know if it is finally accepted or not in the final solution.

Accepting or Rejecting committed Values. The project leader is the only member that can do this work. He/she/it knows all the rules of the game and, for this reason, he/she/it can estimate if a value is correct or not in a certain position. Knowing this, this member will accept or reject the possible values in the final.

As every member of the community has the same information about the game, all of them have to have access to the grid in any moment in order to check the current value and state of a determined cell. For this reason it is very important that the grid remain the same for all the members and that every cell has only one determined static value and state at each moment.

In this manner, at the beginning of the game the grid presents only cells with the states *Waiting Value* or *Initialized by Server*. Then, an agent or a user can connect to the server which assigns him/her a unique ID. In this way, all the members are differentiated and it is always known who participate in each part. Once a member receive his/her ID, he/she can start to collaborate in the community.

The simulation begins when a member contributes to the first value to the grid. Then, the rest of the members starts to work over the contributions made, checking if the values are bugs or correct and communicating the resolutions with the server which broadcast a response to the other members. When a member asks for a voting session, the game is stopped for the agents not to alter the result of the voting. Each voting session lasts between 10 and 15 seconds and it is not necessary that all the committer members participate on it. The cell states we could find at this moment are *Contributed*, *Bug Reported*, *Committed* and the ones commented above. All of them are differentiated by the users or the agents that have changed the state. Once there are committed values, the Project Leader starts to accept or reject them and the simulation is completed, adding the cell state values *Accepted* and *Rejected*.

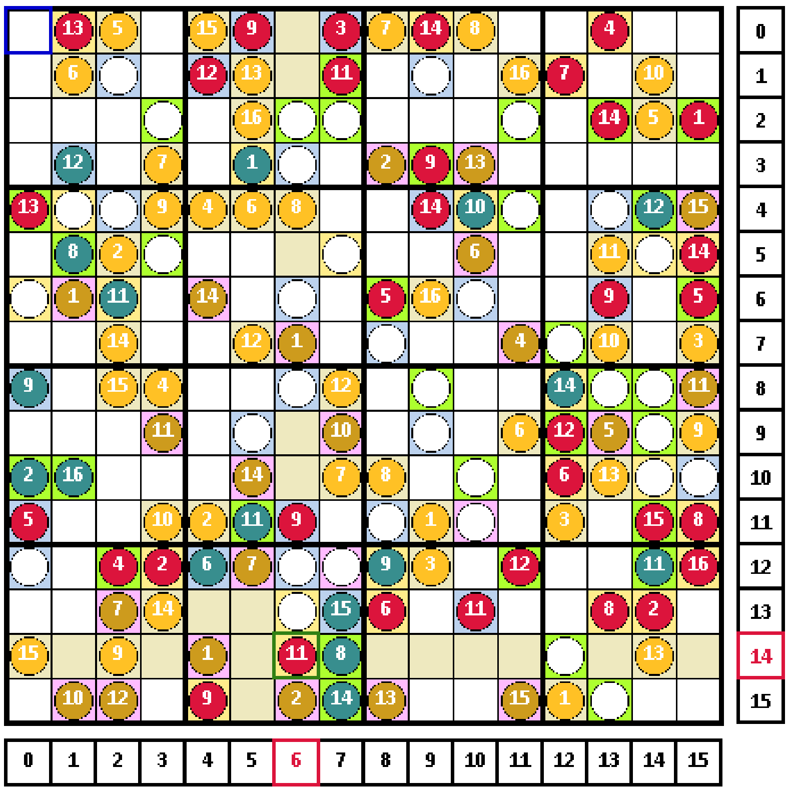


Figure 3.3: Example of a early-mid game situation

The early-mid game is quite fast, with many contributions and bugs reported. In Figure 3.3 we can see an example of a half-developed game. Later, there are more voting sessions, so the game is stopped frequently. In the last game, the situation is even slower, most of the cells are accepted values and in the free cells most of the values are not valid, so there are many contributions that are reported quickly. When there are only a few free cells, it is possible to get a deadlock situation. To solve this, the project leader rejects some accepted values in the most problematic positions. Thus, the contributors have more options to add new values and to solve the situation. The game finishes when all the state cells are *Initialized by Server* or *Accepted*.

4. MUDOKU PLATFORM

The tool we have developed is based on another software created at the Izmir University of Economics, Turkey. Mudoku is a server-client model software that permits to solve a Sudoku problem through the collaboration of a group of people. For this, different users can connect to the server and try to solve a 16x16 Sudoku problem.

In this chapter, this tool will be analysed, focusing in the parts that have been more important in the development of our own tool.

4.1 Overview of the Tool

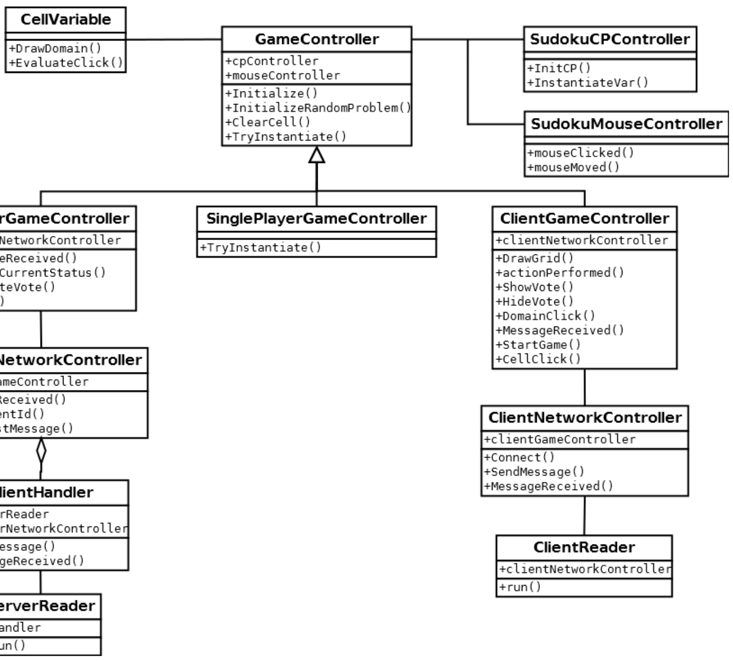


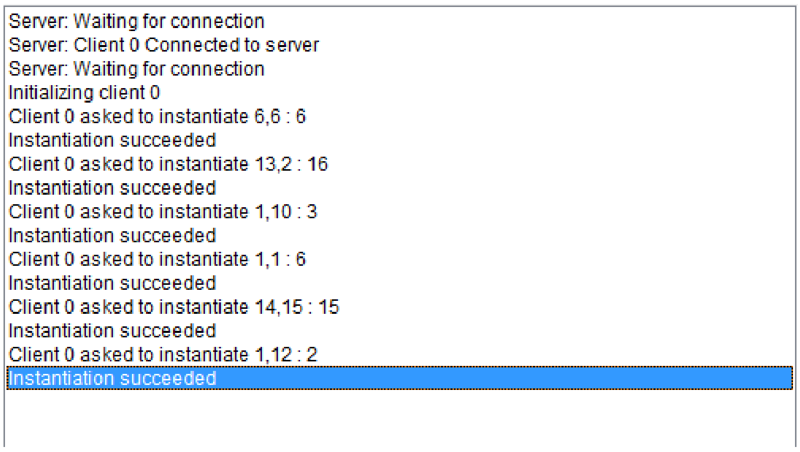
Figure 4.1: UML Diagram of Mudoku

The tool was developed using the Java^{TM} programming language with the software development environment Eclipse. The user interface has been created using Java Applets. This is an applet delivered to users in the form of Java bytecode. This is very interesting, because it can be easily added in a website and it can be run by the Java Virtual Machine in a separate process from the web browser, increasing the speed of execution. In this way, there are two different frameworks or applets defined in the tool, one for the server and the other for the clients. In Figure 4.1 we can see the UML diagram of the tool.

Mudoku uses an external OS library called CHOCO [22]. This is a Java library for constraint satisfaction problems and constraint programming built on a event-based propagation mechanism with backtrackable structures. Thus, in the tool are defined the constraints the library is applying to the Sudoku problem. This fact ensures that all the values added in the grid are always correct and it enforces all the rules of the game in each situation. There are two different ways of playing the game:

Single Player. In this mode, only one user is able to try to solve the Sudoku. The server must be running in the same computer than the one the user is playing on and there are not voting sessions or any collaborative facts.

Collaborative Manner. To play in this way the server could be running anywhere, even in people's own computer, and the clients have to know the IP and the application port of the host where the server is being executed. Once the server is launched, it will initialize a random game and it will wait for the clients. The interface is very simple, it only counts with a console line where all the actions produced are shown. An example of that is visible in Figure 4.2. When a client applet is launched the



```
Server: Waiting for connection
Server: Client 0 Connected to server
Server: Waiting for connection
Initializing client 0
Client 0 asked to instantiate 6,6 : 6
Instantiation succeeded
Client 0 asked to instantiate 13,2 : 16
Instantiation succeeded
Client 0 asked to instantiate 1,10 : 3
Instantiation succeeded
Client 0 asked to instantiate 1,1 : 6
Instantiation succeeded
Client 0 asked to instantiate 14,15 : 15
Instantiation succeeded
Client 0 asked to instantiate 1,12 : 2
Instantiation succeeded
```

Figure 4.2: Example of the console of Mudoku's Server Framework.

first step a user have to do is choose the IP and the port of the server. In Figure 4.3 we can see how this is chosen with the default case, when the server is running in the own computer. When there are one or more users connected, each one of them with their own applet, they can start to add numbers to the grid. To do this, the users can select any empty cell and below the grid will appear the possible values to add, without possibility of error. We find three different colors for the numbers

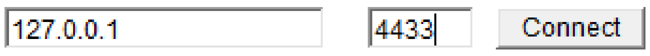
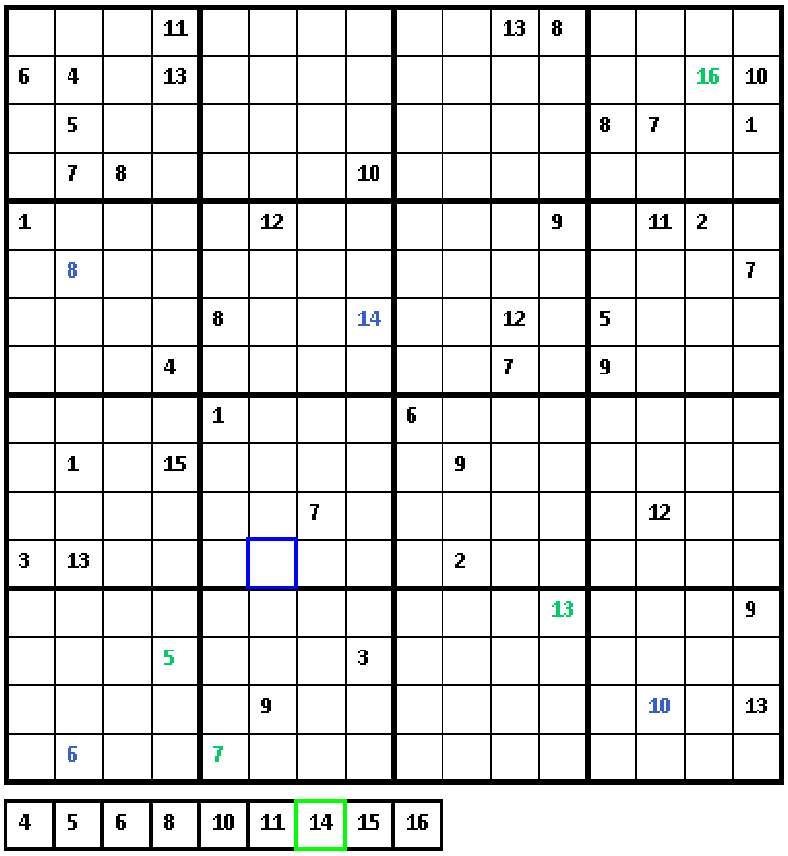


Figure 4.3: Framework to connect a Client to the Server.

in the grid: black for the values initialized by the server, blue for the values added by the own user and green for the ones added by other users. An example of this is shown in Figure 4.4.



			11							13	8				
6	4		13											16	10
	5											8	7		1
	7	8					10								
1					12						9		11	2	
	8														7
				8			14			12		5			
			4							7		9			
				1				6							
	1		15						9						
						7							12		
3	13								2						
											13				9
			5				3								
					9								10		13
	6			7											

4	5	6	8	10	11	14	15	16

Figure 4.4: Example of a grid where an user will add a new value.

When an user detects that a value could be incorrect it can start a voting to decide if that value should be removed from the grid. In this case, the user can select the conflictive value and below the grid will appear a new button to start a voting session. An user can ask to remove any value of the grid, even if this one has been instantiated by the server. An example of this can be seen in Figure 4.5.

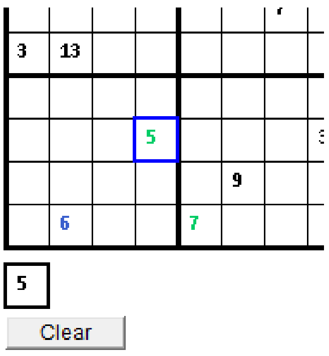


Figure 4.5: Button *Clear* to start a Voting session.

During the 15 seconds voting time all the users can vote to remove or keep the value with two buttons that will appear in the right-top corner. During a vote, it is not possible to ask to clean other values. An example of this is shown in Figure 4.6.

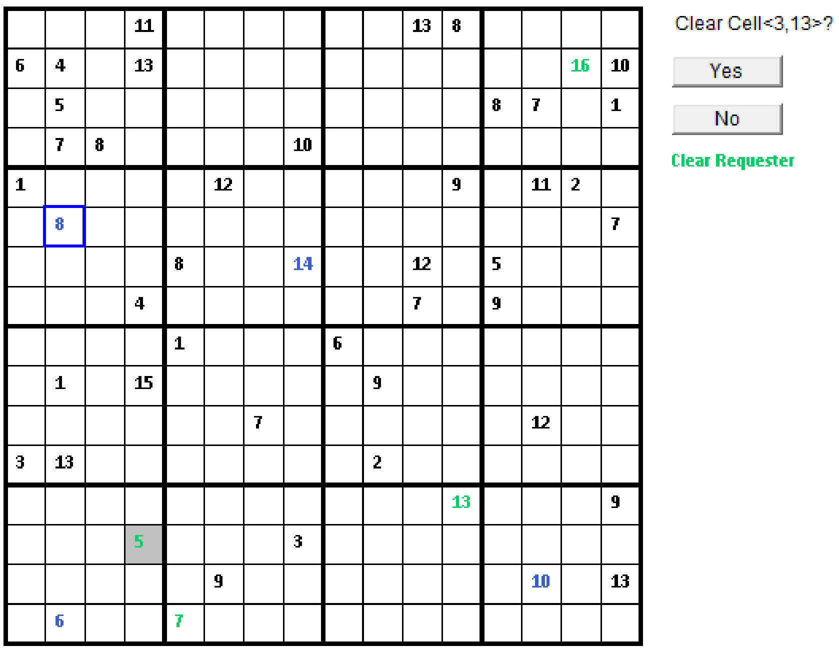


Figure 4.6: Buttons to vote if keep or remove a value.

The game continues until there are no more possible values to add, then the game is finished and correctly played. Then, if the players want to play a new game the server must be restarted and the users reconnected.

4.2 Technical Details

As the tool is programmed with an object-oriented language, it is composed by classes related in different ways and filled by methods. The main class of the tool is GameController. This class defines the common properties of the two different applets who are an inheritance of this one. But it also defines constants like the dimensions of the applets, the size of the Sudoku and the colors of the grid and initializes the game using the CHOCO library, when the server requires it.

This class additionally controls if all values added in the grid are correct (SudokuCP-Controller class): This is done by the definition of constraints using the CHOCO library. The process consists in the creation of a model and adds all the necessary properties that represent the rules of the game. In this manner, every time an empty cell is selected, it is checked which values can fill this position and only this numbers will appear for the users.

Besides, it controls all the mouse actions (SudokuMouseController class): This class has been implemented using the methods of the Java^{TM} Platform. This class permits to know where the mouse is pointing at every each moment and it also allows to add a Java Listener every time there is a click. In this way, every time a user clicks on a cell, a different action will be produced depending on the listeners associated.

Finally, it also controls the values in the grid (CellVariable class): Mudoku does not use different states to control in which situation is a determinate cell. As is not possible to add wrong values, all the numbers in the grid are correct. Because of this, all the cells not initialized will have the value -1 and the others a value between 1 and 16. However, in the user's applet, a matrix is used to know if a cell is initialized by the server, if it is an empty cell or if the value is added by the own user or others. This matrix is used to choose the color of every value in the grid. When a user connects to the server, it receives the encoded matrix and it is varying depending on the user.

4.2.1 Server Applet

This framework is responsible to keep the communication among the users. It is formed by the inheritance of the Class GameController and the following classes:

ServerGameController. It is the main class of the applet controlling all the actions that it could produce and the user interface. Basically, the class works waiting for messages from the users and according to it, it responds in different ways. In the Section 4.3 are explained the different messages and how the server and clients act to answer them. The functions of this class are:

- Initialization: To call the method to initialize a random problem and to launch one thread of the class ServerNetworkController.
- To process and respond in a different way every message received.
- To add to the console all the text for every action produced.
- To control the timer and the resolution of the voting sessions.
- To encode the state of the game every time a new user connects to the server.

ServerNetworkController. This class is implemented to be launched as a thread whose function is to listen by the public port to the new users who want to connect to the server. Once a user sends a message, the class adds the user's information in an array of type ClientHandler to make possible future communications. This class also has the following functions:

- To control when a user is disconnected to remove him/her from the array of users.
- To simulate a broadcast message sending a private message to each user of the array.
- To distribute the ID of the clients connected, ensuring there are not two users with the same one.

ClientHandler. This class is an aggregation of the last one. When a client is connected to the server, an instance of this class is created launching a thread of ServerReader for each user. This allows the server to send messages to this client by its private socket. For this reason, the server is saving the attributes of the user like the socket and the ID.

ServerReader. This class is implemented to be launched as a thread. Its only function is to listen to the messages sent by only one user. In this way, we will find as much threads running as users connected.

4.2.2 Client Applet

This framework permits the interaction between the user and the software. It is formed by the inheritance of the Class GameController and the following classes:

ClientGameController. It is the main class of the applet processing all the message received from the server and the user interaction. Basically, the class works waiting for the actions of the user on the grid and sending messages depending on those actions. In the Section 4.3 are explained the different messages and how the server and client act to answer them. The functions of this class are:

- Initialization: To connect to the server by the public socket and receive the encoded state of the game.
- To draw the grid and all the values instantiated, draw the cell where the mouse is pointing at, the active cell, the domain of the cell if this one is empty and all the necessary buttons depending on the situation.
- To control the timer of the user every time there is a voting session. 10 seconds: after this time, the user has not the option to vote.

ClientNetworkController. This class is used to make the connection with the server. For this reason, a new socket is created for each user and is launched a thread of type ClientReader to listen to the messages from the server. This class also permits to send message to the server by the private socket.

ClientReader. This class is implemented to be launched as a thread. Its only function is to listen to the messages sent by the server.

4.3 Message Protocol

In this tool, a new message system has been defined to allow the communication between the server and the clients and vice versa. The possible situations are:

To *connect a new user* to the server the user should send the following message

"request#type=init".

If the connection is possible (IP and port are correct) the server will receive the last message and it will send the encoded state of the game. First of all the type of the message: *"init#"*, secondly the Sudoku Size: *"ss=XX#"*, thirdly the client ID: *"ci=X#"* and finally the state of the game: Only the values instantiated will be encoded, not all the positions of the grid. In the following way:

"iv=positionX,positionY,currentValue,currentState&..."

The current state only differentiates if the value has been initialized by the server or an user. An example of one of this message could be the following:

init#ss=16#ci=0#iv=0,3,12,-1&0,4,2,-1&0,8,1,-1&0,9,15,-1&0,11,3,-1&...

Once the client receives this message, it can draw the grid with the values and the game can start.

To *contribute with a new value* any member selects an empty cell and then one of the possible values below the grid. The client will send a message like this:

"instantiate#positionX,positionY,value"

The server will receive the message and will try to add the value in the model with the constraints for the Sudoku game. Then, two situations can be produced:

- The value is accepted: If the value is correct the server will send the next message to all the users: *"instantiate#positionX,positionY,value,state"*. The user who instantiated the value will add it as his/her own value while the other users will add it as a value added by another user.
- The value is rejected: Theoretically, this situation never happens. But just in case, the next message is sent to the user who tried to instantiated the value: *"instantiate_failed"*.

To *try to clean a position* when an user detects a conflictive value and push the "clear" button the next message will be sent to the server:

"clear#positionX,positionY".

Then the server will process it in the following way:

- If there is a voting active: The following message will be sent to the user who asked to clean the position *"rejected#clear=voting_exists"*.
- If there is not: The voting will be accepted and the next message will be sent to all the users: *"vote#clear="positionX,positionY,clientID"*. The ID is of the client who asks to clean the position. Then the timer is set and when the users receive the message, all of them have the option to vote for that position. So the buttons will appear and depending on the button pushed, they will send the following message:
  - Remove the value: *"voted#positionX,positionY,1"*
  - Keep the value: *"voted#positionX,positionY,0"*

  The server will receive all the votes and will act in the next way:

- If the final counting is higher than 0: All the users will receive the message *"clear#positionX,positionY"*. And the value will be removed from all the grids.
- If the final counting is not positive the server will not send any message and when the timers of the users will be finished, always before the server's timer, the value will remain there.

5. MUDOKU OSS COMMUNITY PLATFORM

The platform we have developed is using the same model than the last one, but the aim from the beginning is quite different. Now, what we are trying to do is simulate an OSC in the way it has been explained in Section 3.2.3. This can be done in the tool through three types of communities:

*Agent Community.*In this mode, the role of the real user is only to observe the behaviour of the agents and how the different roles interact among them to solve the problem.

Human Community. It is possible to create a community only with real users. In this mode, different users will have different roles with the options of every role and they will collaborate to solve the problem.

Mixed Community. How the tool is designed in the server-client model, we can have different types of clients. In this way, it is possible to connect human clients and agent clients creating a mixed community who will collaborate in the same levels to solve the problem. In Figure 5.1 we can see how this community is composed.

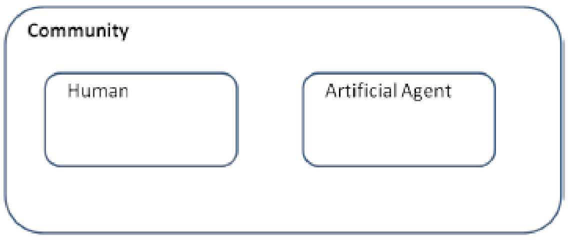


Figure 5.1: Composition of a Mixed Community in the platform.

In this chapter are explained all the details of the work we have implemented. For this reason it will be the longest chapter of the thesis, trying not to miss any characteristic. As in the last chapter, first is written the features of the tool on how it looks like and then can be found technical details to go deeper in the operation of the tool.

5.1 Overview of the Tool

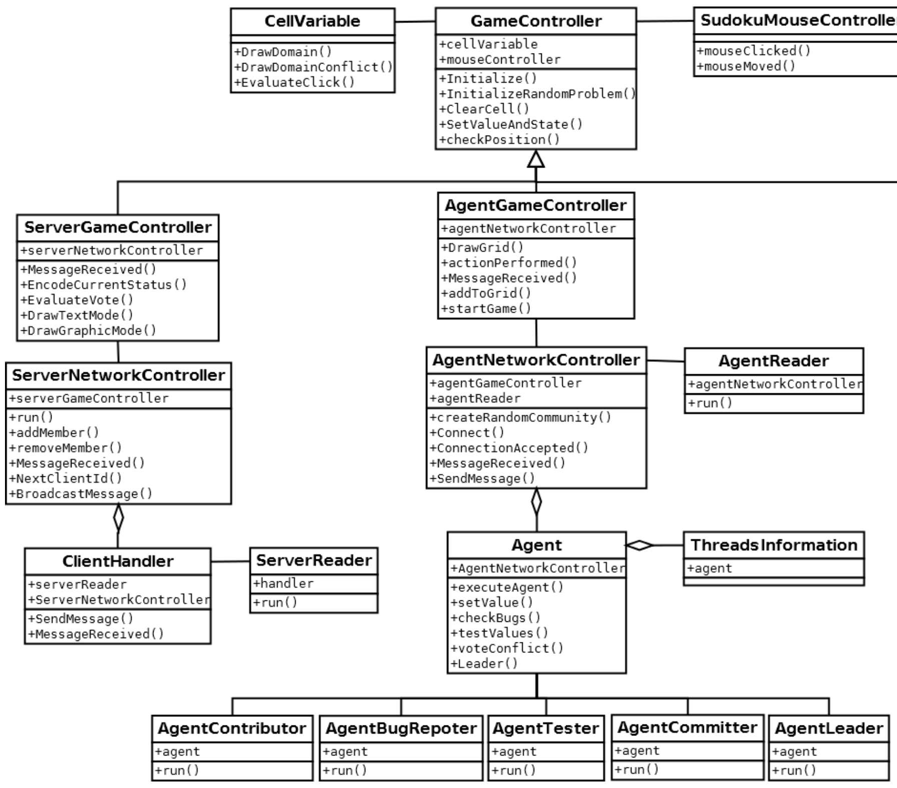


Figure 5.2: UML Diagram of the new platform.

To develop this tool, we have reused some of the code of the previous one. The programming language is still being JAVA^{TM} and it has been developed with Eclipse using GitHub as version repository. In Figure 5.2 there is the UML Diagram of the platform. The tool has been implemented using the pattern of the previous tool, but with some main differences to adapt it to the simulation.

The first big difference is that the platform is composed of three different Applets, one for the server, one for launch agents and the last one for the users:

- Several changes on the Server Applet to adapt it to different types of clients.
- The Agents Applet permits to add agents to the community creating a random community or adding a selected type of agents.
- The Users Applet allows the users to join the community in different roles. This has been thought with a game perspective when the users are surpassing levels until they reach the Project Leader role.

Also, the CHOCO library is not used. The members of the communities commit mistakes. To simulate this part, it is necessary to permit the users to add wrong values to the grid that will be corrected by other members later. For this reason, this library can not be used. Thus, both the initialization of a new game and the control of correct values had to be redesigned.

The grid has been redesigned. As there are different states for each cell and the members could belong to the different subsystems, the colors of the cells had to be changed. In this way, every cell is composed of a background color, foreground color in a circle and the current value of the cell in white. The background will indicate the type of agent who has changed the last state of the cell (row, square or user) and the foreground will show the current state of the cell. In Figure 5.3 and Figure 5.4 can be seen the colors chosen for every state or member.

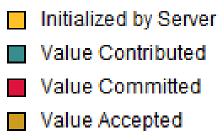


Figure 5.3: Possible State colors for each cell.

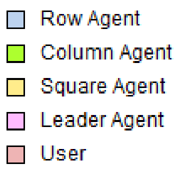


Figure 5.4: Possible member colors for each cell.

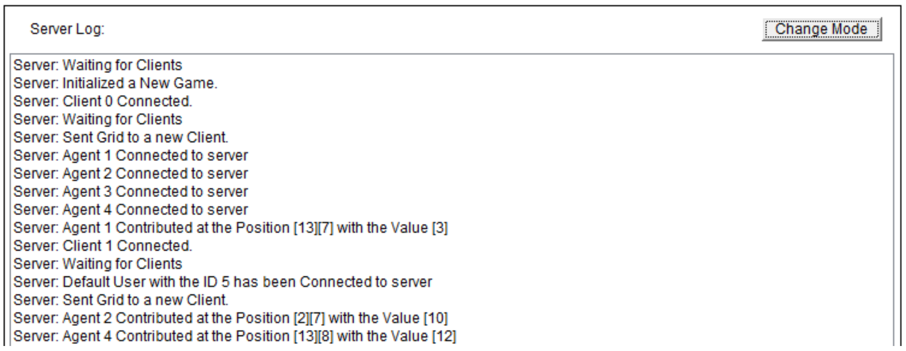
Finally, the following features have been added to the grid:

- When a cell is selected, the borders will change to green and the background of the scope cells will change from white to beige.
- Below and on the right, there are the numbers of the cell to make easier for the user to follow the game. When a cell is selected, the corresponding numbers of these lists will change to violet.
- Every time there is a voting session, the cell selected will change to this voting cell, to make easier for the user to follow the game.
- When the mouse is pointing at the grid the concrete cell will be in blue.

The aim of the server applet is to keep the communication among the clients. Note that the word "client" refers to the different types of applets connected to this one, so a client could be both an Agent Applet and a User Applet. In this way, there should be an only server running where many of the other clients can be connected. A server will represent a community alive, that many users or agents can join.

It has been defined two modes or two interfaces that can be switched at any moment pushing the button on the top-right corner of the applet:

Text Mode. The interface is very similar to the previous tool. The applet consists in a console where all the actions are registered. In Figure 5.5 we can see how this mode looks like. *Graphical Mode.* This is the default mode and the interface



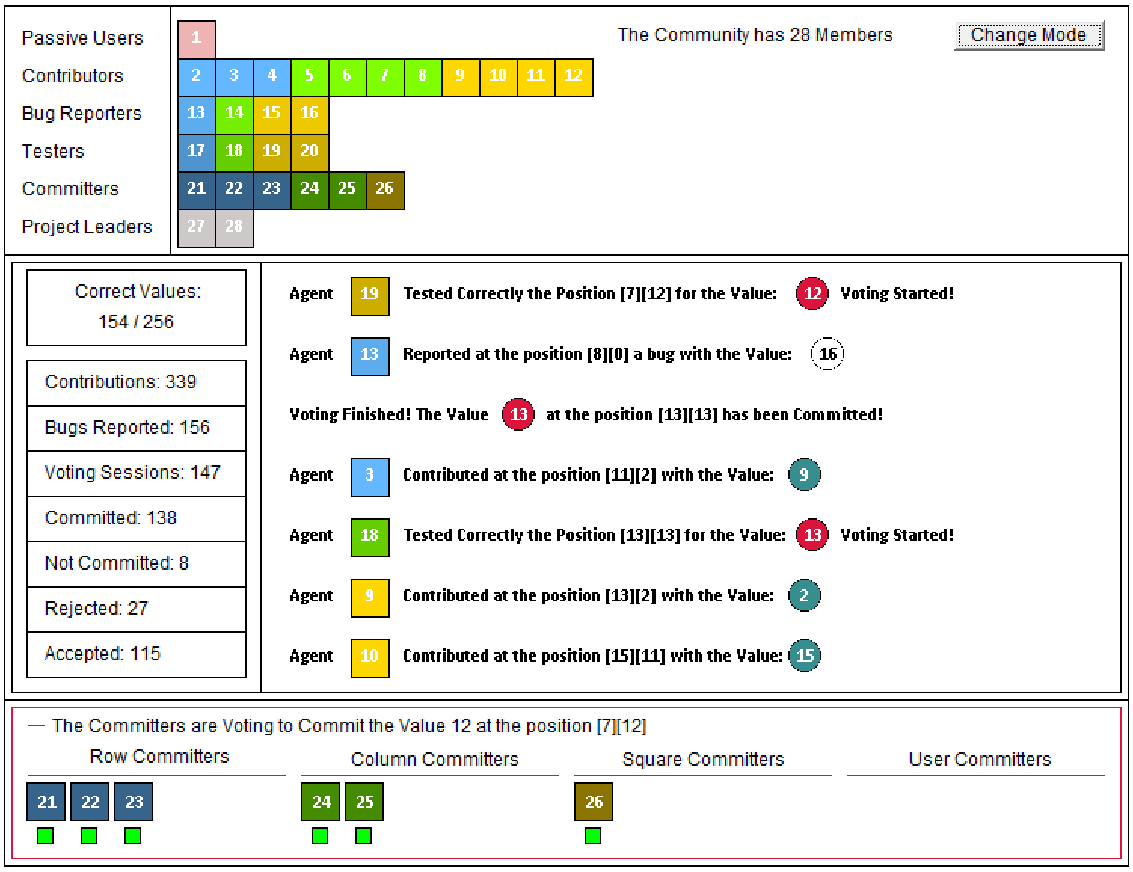
```
Server: Waiting for Clients
Server: Initialized a New Game.
Server: Client 0 Connected.
Server: Waiting for Clients
Server: Sent Grid to a new Client.
Server: Agent 1 Connected to server
Server: Agent 2 Connected to server
Server: Agent 3 Connected to server
Server: Agent 4 Connected to server
Server: Agent 1 Contributed at the Position [13][7] with the Value [3]
Server: Client 1 Connected.
Server: Waiting for Clients
Server: Default User with the ID 5 has been Connected to server
Server: Sent Grid to a new Client.
Server: Agent 2 Contributed at the Position [2][7] with the Value [10]
Server: Agent 4 Contributed at the Position [13][8] with the Value [12]
```

Figure 5.5: Example of the Server Text Mode.

is rather different. The objective is to observe the behaviour of the members of the community in an intuitive way. As we can see in Figure 5.6 the applet is divided in four different parts:

On the *top* of the applet there is the histogram of present roles in the community. This histogram represents all the members connected to the community and which type they are. The different roles are: Passive Users, Contributors, Bug Reporters, Testers, Committers and Project Leaders. Every member is represented with a square filled with the color of the subsystem he/she/it belongs, in the case of the agents, and the ID in the middle. The possible colors of the squares are: Blue for the Rows Agents, Green for the Column Agents, Light Brown for the Square Agents and Salmon for the users that do not belong to any subsystem. Notice that these colors are getting darker in the different roles.
In this part we also find the total of members connected and the button to change the mode.

In the *mid-left* section of the applet there are 8 boxes with different statistics about the game. The first one is called *Correct Values*. Its role is to show the number



The Community has 28 Members

Correct Values: 154 / 256

Contributions: 339

Bugs Reported: 156

Voting Sessions: 147

Committed: 138

Not Committed: 8

Rejected: 27

Accepted: 115

Agent 19 Tested Correctly the Position [7][12] for the Value: 12 Voting Started!

Agent 13 Reported at the position [8][0] a bug with the Value: 16

Voting Finished! The Value 13 at the position [13][13] has been Committed!

Agent 3 Contributed at the position [11][2] with the Value: 9

Agent 18 Tested Correctly the Position [13][13] for the Value: 13 Voting Started!

Agent 9 Contributed at the position [13][2] with the Value: 2

Agent 10 Contributed at the position [15][11] with the Value: 15

— The Committers are Voting to Commit the Value 12 at the position [7][12]

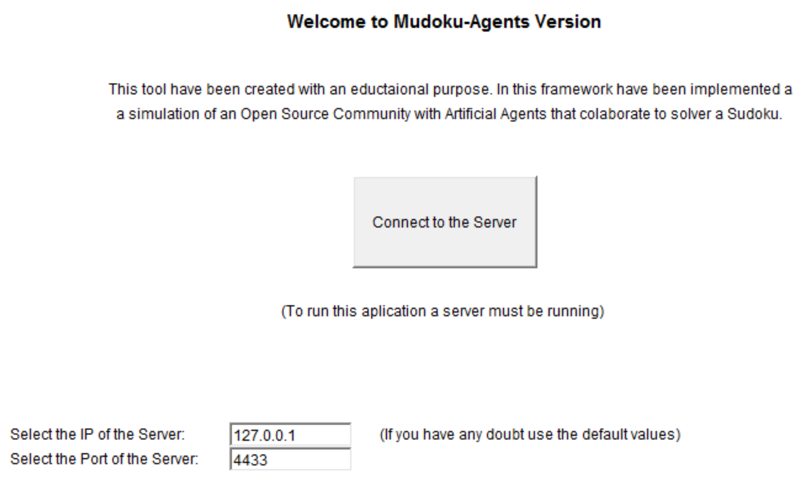
Figure 5.6: Example of the Server Text Mode.

of values that are correct, values initialized by the server and values accepted by the project leaders. The second one *Contributions* indicates the total amount of contributions produced in the game by the contributors. *Bugs Reported* indicates the total amount of bugs reported in the game by the bugs reporters. As for *Voting Sessions* it indicates the amount of voting sessions produced in the game. *Values Committed* indicates how many values have been committed after a voting session and *Values Not Committed* indicates how many values have been not committed after a voting session. The last two are *Values Accepted* that indicates the number of values accepted by the Project Leader and *Values Rejected* that indicates the number of values rejected by the Project Leader.

In the *mid-right* part of the applet there is a big box where are shown the last seven actions produced in the game. Depending on the action could appear the square (color + ID) that represents the member that has produced the action, the position of the grid where the action has been produced and the value. The different actions that could appear here are: *Game Initialized, Value Contributed, Value Reported, Value correctly tested and start of a voting session, Ending of a voting* *session and its result, Value Accepted* or *Value Rejected*.

In the *bottom* of the applet there is a box where all the committers are represented. The box is divided depending on the subsystem the committers belong. Thus it is possible to follow the result of the voting in the moment the votes are received by the server. In this way, when the small square below a member is in green it means that this member has voted to commit the value voted, when is in red the member voted to remove the value. When there is a voting session, the borders of this box change the color to attract the attention of the user. When there is not a vote, the small squares below each member are in white.

The Agent Applet has been created to launch/disconnect agents who will join the community. It is possible to connect many of this type of clients and from each of them launch agents, but every agent launched will ask to join the community separately. In this way, every agent will have an own ID and it can be treated separately. The user must know the IP and the port where the server is running to be able to connect. In Figure 5.7 can be seen how the presentation page of the applet looks like, with the parameters to connect to a server locally.


Welcome to Mudoku-Agents Version

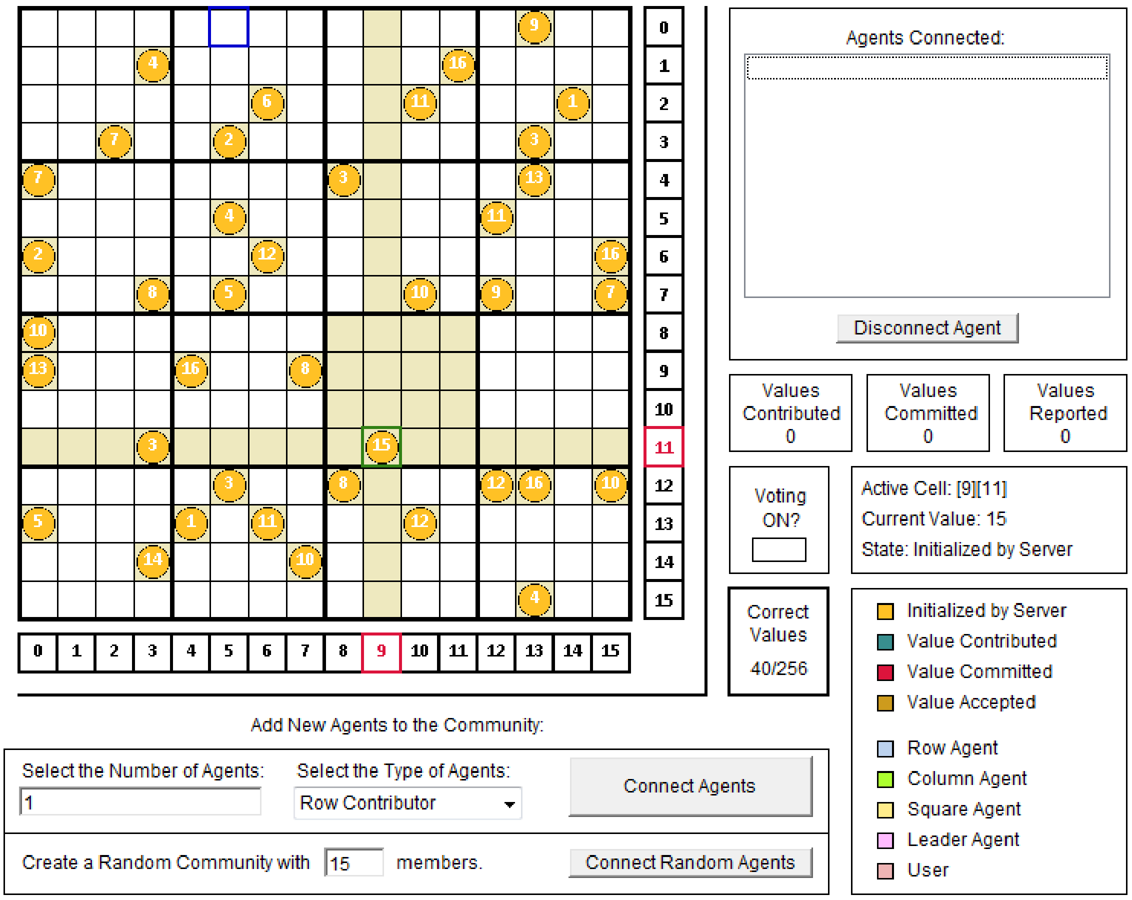
This tool have been created with an eductaional purpose. In this framework have been implemented a
a simulation of an Open Source Community with Artificial Agents that colaborate to solver a Sudoku.

(To run this aplication a server must be running)

Select the IP of the Server: 127.0.0.1 (If you have any doubt use the default values)
Select the Port of the Server: 4433

Figure 5.7: Presentation page of the agent's Framework.

Once the server has accepted the connection, the framework page changes to the main page of the applet which looks like Figure 5.8. In this page, there are all the available options to control the agents. In the first steps of the software, it was possible to stop and resume the agents, but this feature was finally removed because the methods used were deprecated and they could have caused some deadlock problems. In the last versions, the available options are launch agents, disconnect agents and observe their behaviour.



Values Contributed 0

Values Committed 0

Values Reported 0

Active Cell: [9][11]

Current Value: 15

State: Initialized by Server

Correct Values 40/256

Select the Number of Agents: 1

Select the Type of Agents: Row Contributor

Create a Random Community with 15 members.

Figure 5.8: Example of a new game in the Agent Framework

In the Applet, the following parts are differentiated:

Grid. The major part of the framework is for the grid. In this applet, the grid only has an observable function. It means that the user can see what is happening in the grid, but never interacts. The grid has all the features explained at the beginning of the chapter.

Launch Agents Box. On the bottom of the applet there are the two different options to launch agents:

- Selecting type: The first alternative allows the user to select the type of the agent that will be launched and the number of them. In this way, for example the user, could launch six row contributor or three square committers agents.
- Launch a random community: This option permits to select the number of

agents that will join the community, with a minimum number of fifteen. The percentage of each role is the following:

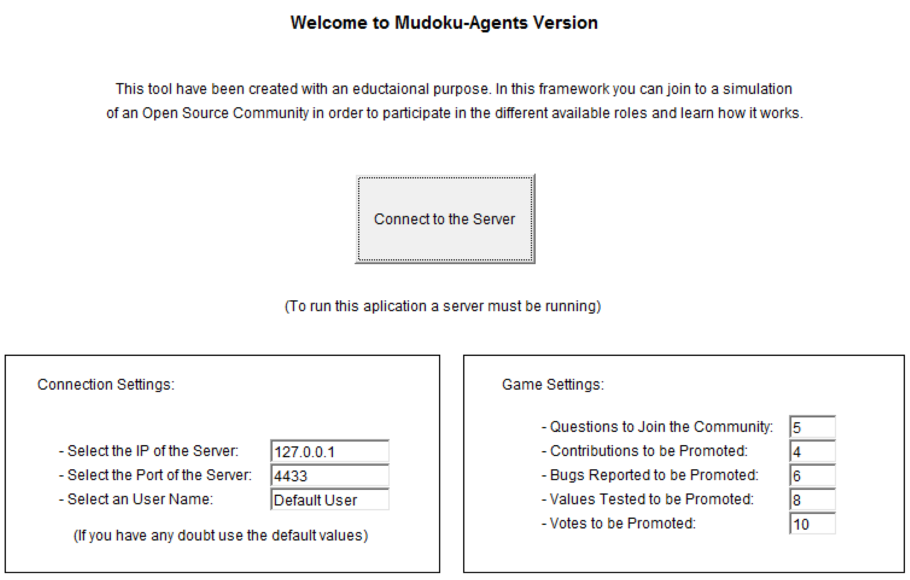
- 45% of Contributors with at least one in every subsystem.
- 15% of Bug Reporters with at least one in every subsystem.
- 15% of Testers with at least one in every subsystem.
- 20% of Committers with at least one in every subsystem.
- 5% of Project Leaders.

List of Agents. In the top-right corner, there is the list of the agents launched by the concrete applet. Every agent is written with the ID and the type of agent is it, for Example *Agent [3]: Tester by Rows.* From this list, the user can select any agent and remove it from the community.

Game Information and Legend. The rest of the grid, in the bottom-right corner, is composed of different information box and a legend with the different colors that appear on the grid. The information box are the following: *Values Contributed by the agents of the applet*, *Values Committed by the agents of the applet*, *Values Reported by the agents of the applet* and *Voting Box*, that is every time there is a voting active the square is filled in violet. It is thought to tell the users there is a voting session active. We can also found *Cell Information* that is when the user selects a cell of the grid in this box appears all the information of this cell, concretely the position (x and y values), the current value and the state. The last two information are *Correct Values* that indicates the amount of values that are correct in the grid and *Legend* that are the explanations of the different colors that appears on the grid.

The User Applet framework is thought with a game perspective. It means the user will join the community in the lowest possible role and he/she will have to get some points to increase his/her weight in the community until he/she reaches the project leader role. The final aim is to teach the users how an OSC works and how it is organized through the motivation of surpassing different levels. The presentation page of the tool looks like Figure 5.9. In this page, the user has to choose the connection settings and the game settings:

- Connection Settings: As in the previous Applets, the client needs the IP and the port of the server, but also it is possible to choose a username for the game.
- Game Settings: It is possible to choose how complicated will be the game choosing the number of points that the user will have to get to pass each level.



Welcome to Mudoku-Agents Version

This tool have been created with an eductaional purpose. In this framework you can join to a simulation of an Open Source Community in order to participate in the different available roles and learn how it works.

(To run this aplication a server must be running)

Connection Settings:

- Select the IP of the Server:	127.0.0.1
- Select the Port of the Server:	4433
- Select an User Name:	Default User

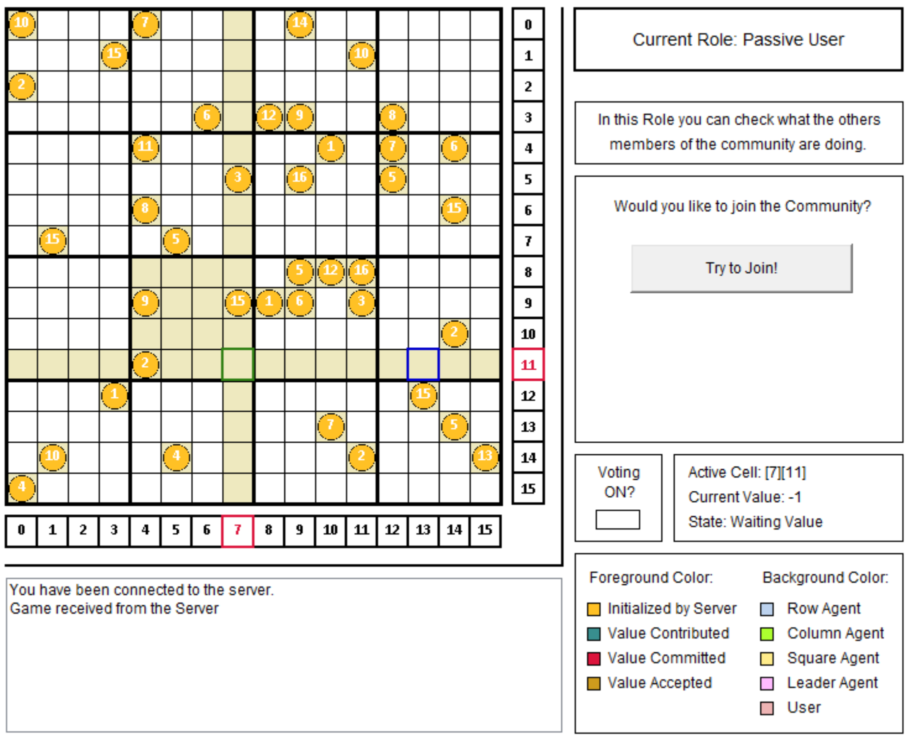
(If you have any doubt use the default values)

Game Settings:

- Questions to Join the Community:	5
- Contributions to be Promoted:	4
- Bugs Reported to be Promoted:	6
- Values Tested to be Promoted:	8
- Votes to be Promoted:	10

Figure 5.9: Presentation page of user's Framework.

Once the client is connected, the user will have a passive user role which looks like Figure 5.10. All the levels of the game will have common features in the framework and only in the top-right corner there is the part that will change with every level. In the following list can be seen the common ones:



Current Role: Passive User

In this Role you can check what the others members of the community are doing.

Would you like to join the Community?

Voting ON?

Active Cell: [7][11]
Current Value: -1
State: Waiting Value

You have been connected to the server.
Game received from the Server

Foreground Color:	Background Color:
🟧 Initialized by Server	🟦 Row Agent
🟦 Value Contributed	🟩 Column Agent
🟥 Value Committed	🟨 Square Agent
🟧 Value Accepted	🟪 Leader Agent
	🟥 User

Figure 5.10: Example of Passive User Role.

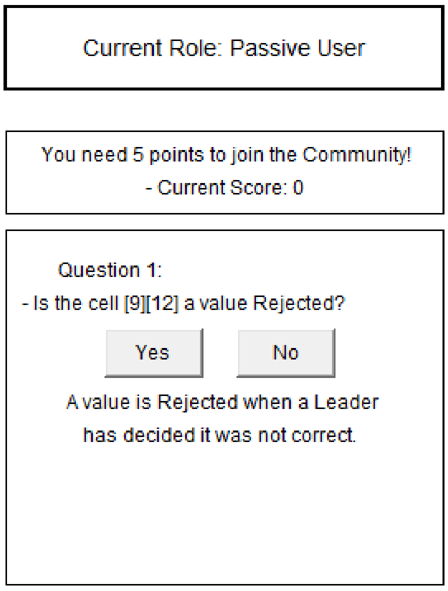
As in the previous applet, the grid will be present in the major part of the framework. In the bottom-left corner there are a small console in order to know easier the last movements in the grid and to inform the user about some facts related to the game. Besides, there are three different boxes that are also in the agent's applet. The first one is to know if there is a voting active, the information cell box is to know the details of a cell selected and the last box is a legend to understand the different colors of the grid.

The applet is divided in different roles or levels of the community. In this way, depending on the behaviour of the player, the user will be asked to get promoted to the next role or to be displaced to the previous one. In the following list, there are the levels of the games and the explanation of what to do in each:

Passive Role. As we have commented above, only this level has an observable function, but in any moment, the user can ask to join the community pushing the right button. Once the button is pushed, the right-top corner looks like in Figure 5.11. On the top there is the box that informs about the current level, under there are different buttons in order to switch levels once unlocked (not visible yet in the figure) ; the next box informs the user about the current score in the level and some information about the role. To join a real community, the person has to show his/her skills about the topic. Then, he/she could be accepted. To simulate this part, the user will have to answer correctly a determined number of questions about the simulation of an OSC with Sudoku.

On all levels, when the score is the minimum to be able to get the next level, a button will appear below the information box of the role. In this way, the user can still play in the same level or move to the next one. If the behaviour of the player is not good for the community, if the answers are wrong, the user will be moved to the previous level. In this case, he/she would be moved to the presentation page.

Contributor Role. Once the user has answered correctly the questions and pushed the promotion button he/she will be moved to the contributor level. In this level, the user adds values to the grid. The top-right corner looks like in Figure 5.12. In the figure, there are the same common parts than in the previous level changing in the mid zone. Here there are the different numbers that can be added to the grid. The user has to select an empty/reported/not committed/rejected cell and then choose a value to add it to the grid. The user will receive a point every time one of the values added is committed. If the value is not committed, the user will


Current Role: Passive User

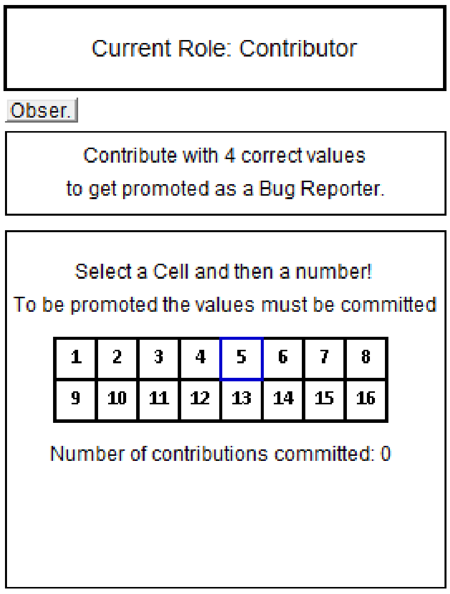
You need 5 points to join the Community!
- Current Score: 0

Question 1:
- Is the cell [9][12] a value Rejected?

A value is Rejected when a Leader has decided it was not correct.

Figure 5.11: Example of question to join the community.

have -1 point and he/she could be moved to the passive role.


Current Role: Contributor

Contribute with 4 correct values to get promoted as a Bug Reporter.

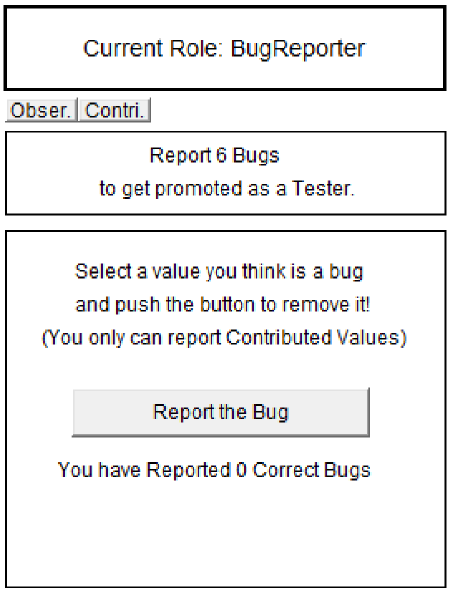
Select a Cell and then a number!
To be promoted the values must be committed

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

Number of contributions committed: 0

Figure 5.12: Option of the contributor Role.

Bug Reporter Role. In this level the user reports the values that are considered wrong. The top right-corner of the applet looks like Figure 5.13. To do this, the user selects any cell of the grid and push the button. If the value selected is not a contribution, the user will be informed that this value can not be reported. The player will receive 1 point if the value was incorrect and -1 if it was correct.


Current Role: BugReporter

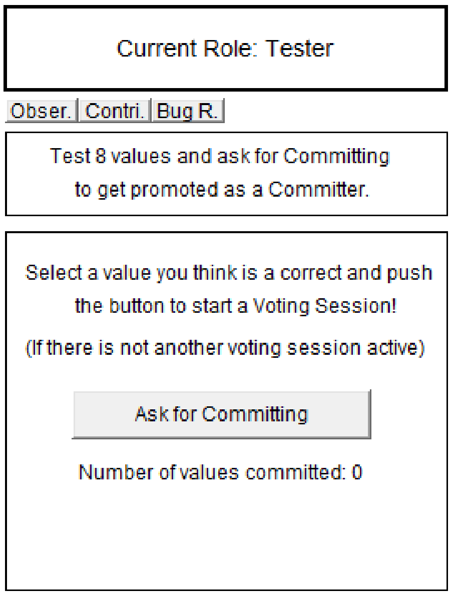
Report 6 Bugs
to get promoted as a Tester.

Select a value you think is a bug
and push the button to remove it!
(You only can report Contributed Values)

You have Reported 0 Correct Bugs

Figure 5.13: Options of the Bug Reporters Role.

Tester Role. In this level the user selects contributed values and then push the button to ask to commit them. All the values selected will create a voting session and the committers will vote to keep or remove the value. The top right-corner of the applet looks like Figure 5.14. The user receives 1 point when the value is committed and -1 when the value asked is not committed.


Current Role: Tester

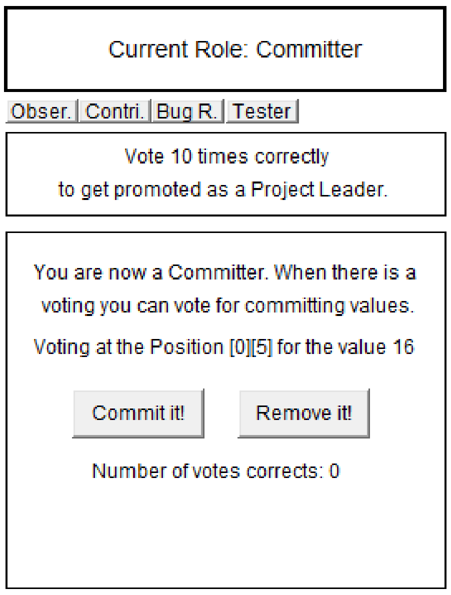
Test 8 values and ask for Committing
to get promoted as a Committer.

Select a value you think is a correct and push
the button to start a Voting Session!
(If there is not another voting session active)

Number of values committed: 0

Figure 5.14: Options of the Tester Role.

Committer Role. In this level, the user takes part in the voting sessions to commit or not to commit values. Every time there is a voting session, two buttons will appear and the user can choose one of them to make a vote. The top right-corner of the applet looks like Figure 5.15. The user receives 1 point when his/her vote helps to win the voting and -1 in the contrary case.


Current Role: Committer

Vote 10 times correctly
to get promoted as a Project Leader.

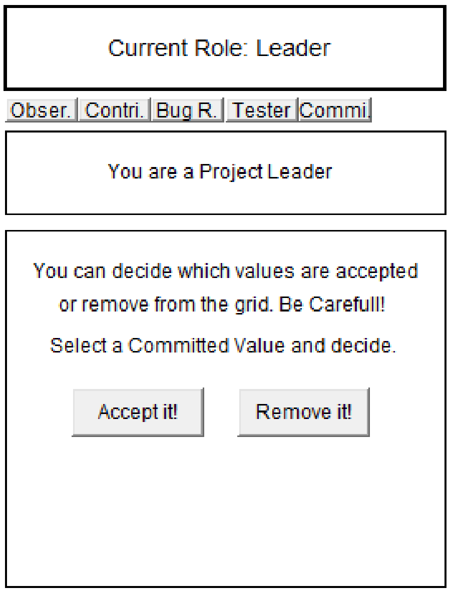
You are now a Committer. When there is a voting you can vote for committing values.

Voting at the Position [0][5] for the value 16

Number of votes corrects: 0

Figure 5.15: Options of the Committer Role.

Project Leader Role. It is the last possible role. In this level, the player selects the values that have been committed and choose if they are accepted or rejected. The top right-corner of the applet looks like Figure 5.16. In this role the user can not be promoted, but he/she can be moved to the committer role if the values accepted are wrong or if the values rejected are correct. Later, another project leader will take care of the wrong values. The game does not finish here though, the main aim is, when all the levels have been reached, to finish the Sudoku problem.


Current Role: Leader

Obser. | Contri. | Bug R. | Tester | Commi

You are a Project Leader

You can decide which values are accepted or remove from the grid. Be Carefull!

Select a Committed Value and decide.

Figure 5.16: Options of the Project Leader Role.

5.2 Technical Details

As in the previous chapter, in this section, all the features of the tool are explained deeper. The main class of the tool is still being GameController, but it has some differences with the previous tool:

The class defines all the constants needed in the applets: cell states, game states, last actions states, sizes of the applets, colors of the applets, etc. It initializes the game by new functions that check value per value if it is correct.

Every cell of the grid is an instantiation of the class CellVariable. In this way, it is possible to control easily the state and the current value from the applets. The possible states of a cell are the following: *Cell waiting for a Value, Value initialized by the Server, Value contributed by Rows, Columns, Squares or User, Value reported by Rows, Columns, Squares or User, Value committed by Rows, Columns, Squares or User, Value not Committed, Value accepted by Agent or User or Value rejected by Agent or User.*

As in the previous tool, the main aim of the Server Applet is to keep the communication among the users while it is also running waiting for new connections. Most changes are given by the necessity to interact with different clients. In this way, all the messages expected are differentiated if they come from an agent's applet or user's applet. Besides, there are many changes in the interface as the graphical mode adds complexity to the applet. Because of this, it is necessary to save all the actions produced in the game. A file is stored with a log of the statistics of the whole game when the problem is solved. The classes present in the applet are the following:

ServerGameController. It is the main class of the applet processing all the messages received from the users and giving them an answer. It has the same functions than the previous tool, but also it calls the method to initialize the game (GameController) and put the graphic mode as default. It saves all the actions produced by the users in order to show them in both modes. It also saves statistics about the game and lists all the member connected as well as control their IDs not to do have two members with the same one.

The most important part of the server is to know how to react in front of the input messages. For this reason, in all of them it is controlled if they are sent by the correct type of member and if the state of the cell is correct for that message. Also, in the server there is the original grid of the game. All the clients have they own grid, but the original one is on the server. It is very important that all the clients receive the same information and at the same time.

ServerNetworkController. This class is very similar to the one in the previous tool. It is implemented to be launched as a thread whose function is to listen by the public port to the new users who want to connect to the server. Then, it creates an instance of the class ClientHandler to save all the information to reply to each client. The only new feature is this class controls the members of the community. Every time a member wants to join the community he/she has to send a connect message which is processed by this class. This is well explained in Section 5.3.

ClientHandler. This class is an aggregation of the last one. When a client is connected, an instance of this class is created by ServerNetworkController class and a thread of type ServerReader is launched, enabling the server to send messages to the concrete client by its private socket. For this reason, it has saved the attributes of the client as the socket and the ID.

ServerReader. This class is implemented to be launched as a thread. Its only function is to listen to the messages sent by one user only. In this way, we will find as much threads running as users connected.

The aim of the Agent Applet is to launch and disconnect different types of agent. In this way, the applet works as a client who connects to the server and then multiple agents, running in threads of the applet, ask to join the community in the server. It is one client which simulates to be multiple members. The classes of the applet are the following:

AgentGameController. It is the main class of the applet processing the actions produced by the users, connect agents, and the messages received from the server. The principal functions are:

- Show the grid with all the different states and values.
- Save the statistics of the agents of the own applet.
- Permit to connect and disconnect agents at any moment.

When the applet is closed, this class informs the server that the agents are disconnected. This way, there are not dead members in the community.

AgentNetworkController. This class is responsible for the connection of the client and the agents. Besides, it permits to send messages to the server and it launches the thread of type AgentReader to receive messages for all the agents launched. When an agent is launched, it calls the method from the class Agent which will launch the threads.

AgentReader. This class is implemented to be launched as a thread. Its only function is to listen to the messages sent the server.

Agent. This class launches the threads of all the agents which will join the community and it also implements the methods that simulates the behaviour of the roles of the community. All the information about each type of agent (contributor, bug reporter, tester, committer and project leader) launched is saved with an instance of the class ThreadsInformation. With this class, it is easy to stop, pause and resume every agent separately, but, in this tool, only stop agents is a feature available.

The methods that implements the behaviour of the agents are included in this class because all of them must be executed by only one agent of the applet at the same time. Otherwise, it would be not possible for the agents to know the correct values on the grid, and there could be contributions over others contributions or many mistakes at the end of the game. For this reason, Java Monitors are used, every method is static and synchronized. This means that all the agents are waiting in a queue for one agent to finish the method and when it finishes another one will start to run it. Also, all the agents are stopped when there is a voting session active. This is done so that when a committer agent is checking the grid to vote to keep or remove a value, there will have not be interferences. If there were agents running, it would add randomness to the game. But, there is already randomness added in the behaviour of the agents, so there is no necessity to add it also at this point. The methods that implement the behaviour of the agents are:

Contributors Agents. The behaviour is implemented with the Method SetValue which aim is to add values to the grid. The method differentiates the subsystem of the agent: Row, Columns and Squares, so every type of agent will work only in its scope. There are three steps to add a value to the grid. First, the agents check the scope depending on the subsystem the agent is working in. For example, a row agent creates a list of lists ordered by rows. Secondly, the agents choose randomly an empty position and random value in the scope's possibilities. Finally, they send a message to the server trying to add the value on this position.

Bug Reporter Agents. The method is called CheckBugs and it also works depending on the type of the agent. Every time an agent wants to report a bug, it has to check the scope of the subsystem the agent belongs to. For example, a row agent creates a option list with all the values contributed by Columns, Square and User members. Then, it has to choose randomly one of that options and its position. Once it is done, it has to checks if the value is correct for the scope of the agent. For example, a row agent will check if the value added by a columns agent is correct per rows. At last, if the value is incorrect the agent sends a message to the server reporting the bug.

Tester Agents. The method is called TestValues and it also works depending on the type of the agent. The behaviour is exactly the same as a Bug Reporter but looking for correct values instead of wrong values.

Committer Agents. The method is called VoteConflict and it also works depending on the type of the agent. To emit a vote, an agent has to check the scope of the agent for that position. Then, it has to check if the value voted is correct for the scope of this type of agent. For example, a row committer only will check if the value is correct by rows. If the value is correct then there are a 80% of possibilities than the vote is to keep the value and a 20% to remove it. If the value is incorrect it is the contrary. Besides, there are a 10% of possibilities than the agent does not produce any vote.

Project Leader Agent. This kind of agent will have implemented two different methods:

- Normal Behaviour: To check the committed values if they are correct or not. If a value is correct, it is accepted for the final solution ; if it is not, it is rejected.

- End-game Behaviour: Sometimes, at the end of the game, it is possible to end up with a deadlock situation. This could happen when there are less than 20 empty positions but all the values added are wrong. In this situations, the project leader will check these conflictive positions and will clean a position to try to redirect the game to a correct solution and go through the deadlock.

The morphology of the Users Applet is very similar than the client in the previous tool. To apply the game of roles only the class ClientGameController was modified. In this platform, this class is divided in different states. Depending on the level, the applet will offer to the users different features related with the characteristics of an OSS community. The different levels are:

The *Pre-game* refers to the presentation page with the game and connection settings.

Passive User. The only option the user has is to ask to join the community. Once the button has been pushed, there are stored 10 different questions about a random position of the grid. An example of these questions could be *is the cell [3][14] a value Initialized by Server?* or *is the value [6][2] a value Accepted?*. When the user reaches the minimal points, then a new button appears to move to the next level.

Contributor. In this level, the user has the option to add numbers from 1 to 16 the grid. To implement this part, it is controlled where the mouse is pointing at. It is possible to know which number has been selected depending on the coordinates the mouse is pointing at when the user clicks the right button of the mouse. Then, the active cell is checked and if it is an empty position, a bug reported, a value not committed or a value rejected, the instantiation is sent to the server.

The user will be asked to get to the next level when a concrete number of his/her contributions are committed. To develop this, there is a list where all the contributions of the user are stored and when a value is committed, it is checked if the user is in the list. If he/she is, the user will have one point. However, it is also checked whenever there is a value not committed, in this case, the user will receive -1 point.

Bug Reporter. In this level, the user has the option to report values. The implementation is done checking the active cell. If it is a contributed value, a button will appear to report it. In the moment to report it, it is checked if the value was correct or not to give the user +1 or -1 point.

Tester. It is very similar than in the last case. When the active selects a contributed value a button will appear to ask to commit the value. This is saved in a list, as in the contribution case, and if the value is committed, the user will receive 1 point (-1 otherwise).

Committer. When there is a voting session, two buttons will appear to keep or to remove a value and the buttons will disappear when the voting is finished or a vote had been sent. Each vote sent is stored and if the user voted the same than the result of the voting, then he/she will receive 1 point, (-1 otherwise).

Project Leader. When the user selects a committed value, buttons will appear to accept or reject the value. All the actions are checked and if there is any incorrect value accepted or any correct value rejected, the user will receive -1 point.

In all the levels, if the user has less than -2 points, he/she is moved back to the previous level, losing weight in the community. Besides, the points, in every state, are stored during the whole game, but if the users is moved to a previous level, the counting of that level is reset.

The classes ClientGameController and ClientReader are exactly the same than in the Mudoku platform.

5.3 Message Protocol

The message protocol had to be redesigned because of the implementation of different clients, but also there are many more new types of messages because of the new features added. In order to differentiate if a message is coming from an agent or an user, each message includes the type of the member. This way, the server will know quickly how to process the message. The member types are the same commented during the thesis (contributor by rows, tester by user, etc.

As in the previous tool, the first step is to receive the state of the game. This is done exactly as in the previous tool, encoding all the positions of the grid that has a state different of "waiting a Value".

To connect a new member to the community, it is necessary to send a message to the server to get an ID. This way, the message is different if it comes from an agent or a user:

- Agent: *"connect#typeAgent"*.
- User: *"connect#typeUser,userName"*. Where the username is used to inform about all the actions the user will produce. For example in the console: *The user Johnny contributed to the value 6 at the position [5][14].*

When the server receives this message, it will assign an ID for the new member and will respond with the message: *"memberConnected#ID,memberType"* and the member cam join the community.

To disconnect a member from the community, the message is *"disconnect#ID,memberType"*.

To add a new value to the grid, the semantic of the previous tool is used. The message is:

"instantiate#memberID,memberType,positionX,positionY,value".

As the message is only to inform the rest of the client the server will not process anything, it will just reply all the clients with the message:

"instantiated#positionX,positionY,value,contributedState".

where "contributed state" can be contributed by Rows, by Columns, by Squares or by User.

This message and the following ones have been implemented trying to follow the same structure than the last one. To report a bug:

"bugReported#memberID,memberType,positionX,positionY".

The server will clean the concrete position in the grid and will answer with the message: *"bugFound#positionX,positionY,reportedState"* where the "reported state" can be bug reported by Rows, by Columns, by Squares or by User.

When a tester finds a value to be committed the following message will be send:

"clear#memberID,memberType,positionX,positionY".

Then, if there is not any voting active, the server will start the voting timer and it will send to all the clients: *"vote#clear=positionX,positionY,testedState"*. where the "tested state" if the type of the agent which asked to commit the value: tested by Rows, by Columns, by Squares or by User.

When the committers have chosen what to vote in a voting session, they will send a message like the next one:

"voted#memberID,memberType,positionX,positionY,vote".

The vote can be 1 or -1 depending if the member is voting to keep or to remove the value. Once the voting is over, the server can send these two possible messages:

- If the value is committed: *"committed#positionX,positionY,votedState".*
- If the value is not committed: *"notCommitted#positionX,positionY,votedState".*

where the "votedState" is the agent which asks to commit the value, as in the previous message.

To accept a new value in the grid, the message looks like the following one:

"accept#memberID,memberType,positionX,positionY".

The server will reply the clients with the message:

"accepted#positionX,positionY,acceptedState".

where "accepted state" can be accepted by agent or by user.

To reject a value is very similar than the last case:

"reject#memberID,memberType,positionX,positionY".

The server will reply the clients with the message:

"rejected#positionX,positionY,rejectedState".

where "accepted state" can be rejected by agent or by user.

A user can be moved from one level to another in the situation when he/she reaches the next level or when he/she is moved back to the previous one. The message is the next one:

"getPromotion#memberID,memberType,newMemberType".

In this case the server will not send any message back.

6. DISCUSSION

During this thesis, it has been demonstrated that it is possible to simulate a complex system as an OSS community with collaborative games. The platform we have developed is one of the first simulations of an OSS community created and the most complete nowadays. It is able to teach how a community works and how the different roles work in a community.

The simulation of a complex system as an OSS community is a difficult exercise. However, when all the agents simulated are launched, our simulation runs well enough. All the roles are well synchronized and the community, either of agents or mixed, are able to reach a good solution for the Sudoku problem in a determined time.

In addition, it has been demonstrated that collaborative games combined with serious games are a good way to solve complex problems. In this case, a community is solving a Sudoku problem inadvertently. Each role is working in its scope doing a small job and it is through the synchronization of these roles that, at the end, we obtain a good solution. The Sudoku game is suitable for this simulation because it permits to divide the problem even in smaller parts with the different subsystems. In this manner, the implementation of the behaviour of each agent is not complicated and they run very fast in any machine. Even the real users have to do a small job, as selecting a number in a row or voting if a value is correct in a determined position.

Nevertheless, the limitations to apply to the simulation on a Sudoku game must be taken into account. First of all, it should be delivered with a document explaining how to use the platform and how each feature simulated is represented. Otherwise, it could be complicated for users that are not familiar and do not know about OS and OSS Communities. Also, we did not implement any forums or discussions mails, all the decisions made are based on the information every member has. As it is one of the main feature of an OSS community, it would have been better to develop it. Moreover, in the case the agents do not acknowledge meritocracy, the agents are launched with a determined role and it never changes. In a real community, the users are regularly changing roles. Besides, the main properties of an

OSS community have been adapted to the characteristics of the game, however, the simulated parts are limited by the intrinsic possibilities of the game.

The platform could better simulate an OSS community implementing more attributes. For example, we could add agents with different roles. The idea would be to allow the agents to be for example contributor and tester at the same time, as it happens in the real communities. One user could have more than a role, so it could be implemented. We could also add agents changing roles. As in the real communities, if a concrete user is working well for the community, he/she receives more weight in the organization. It could be implemented in the way the agents who are getting better results can be moved to positions nearer to the power. Agents offline could be created. In the real communities, all the members are not working regularly. This can be implemented making that in some moment some agents could leave the community and adding agents who want to join. This would make the community more dynamic.

We could implement more types of making decisions systems to accept values. We implemented a pure voting system, but it could be interesting to try to implement different types of voting, for example where one of the agents of a subsystem has more weight in the voting, as well as some dictatorial decisions. Agents in different subsystems could be generate. It would also be interesting to create agents that could belong to two different subsystems. In this way, there would be more possibilities to have good values without checking the whole grid, increasing the speed of execution. Also, adding more clients could be an interesting option. Thanks to the server-client model it is possible to add more clients with new features. Thanks to this, we could create more different types of applets with other goals. Finally, we could add more role. As in the last feature, the code of the tool could be implemented to permit to add new roles very easily. To do this, only a new class has to be added with the behaviour of the new role and then it could launch as many threads as agents of this type which will join the community.

According to all these limitations, the next step would be to analyse if this kind of communities would work better that a single entity to solve complex problems. Since we did not try to make tests in this directions, we could not extract any conclusion. But trying to solve large Sudoku problems with this kind of systems could be very interesting.

Also, the simulation can only be done in the terms of the Sudoku game. Because of this, what we suggest is to try to apply the simulation to another type of problem not related with games or with another game. Probably, the new simulations will have limitations too, but different OSS community features could be implemented. The idea is to create a set of tools with educational purpose where most of the features of an OSS community could be implemented.

7. CONCLUSIONS

The platform developed is a tool with an educational purpose. So, it should be used to learn what is an OSS Community, how they are structured and which are the functions of each role. To know if our tool is valid for this work, it should be compared with other similar tools. Unfortunately, the simulations done about this topic are too different from our tool.

Consequently, as there is not any similar tool to compare it, the validation has been done testing the platform with fifteen users, where most of them did not know anything about OSS communities. At the end, we have collected their impressions. Thanks to that, we knew that the work was more or less a success. Most of them said they learnt what is an OSS community and that the tool was really useful. However, some of them did not understand the abstraction to apply a game to a software developing model and they suggested to write a document to explain exactly what represents to add a value to the grid and why this is a contribution, for example. As a conclusion of this test, we can say that the tool has some interesting features and it is a useful model to teach.

The topic of this thesis can be widely expandable. Although OSS communities are known and used for decades, there are not many studies about how each role is working. Most of papers published are focused on the explanation of the success of these kind of communities in front of the traditional business model. It is more complicated to find an abstract analysis about a concrete community. This is why we propose to analyse deeper if an OSS community simulation can be done in a better way than we did or with more features added. Thanks to some modifications, it could be improved in the future, to make it more complete, more user-friendly and maybe even more interesting. These changes could be the topic of another thesis.

REFERENCES

[1] M. Bakardjieva (2005) Internet Society. *SAGE Publications Ltd. United Kingdom* 221 pages.

[2] W^3 Tech. Usage of operating systems for websites. Available online at http://w3techs.com/technologies/overview/operating_system/all/ (Last visited on February 2013).

[3] NETCRAFT LTD. The netcraft web server survey. Available online at http://www.netcraft.com/survey/ (Last visited on February 2013).

[4] C. Abt Clark (1970) Serious Games. University Press of America. 176 pages.

[5] G. Yongquin Gao, V. Freeh. Modeling and Simulation of the Open Source Community. *Agent-Directed Simulation Conference, San Diego, CA, April 2005.*

[6] E. Bonabeau (2002) Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America.*

[7] Jose P. Zagal, Jochen Rick, Idris Hsi. (2006) Collaborative games: Lessons learned from board games. *Simulation Gaming March 2006 vol. 3 no. 1 24-407*

[8] Robert L. Glass (1998) In the Beginning: Recollections of Software Pioneers. *Los Alamitos, CA: IEEE Computer Society Press.* 318 pages.

[9] Sam Williams (2002) Free as in Freedom: Richard Stallman's Crusade for Free Software. *O'Reilly Media. Available under GFDL. United States.* 240 pages.

[10] GNU's Bulletin, Volume 1 Number 1, page 8. (1986) Available online at http://www.gnu.org/bulletins/bull1.txt (Last visited on February 2013).

[11] The Free Software Definition. *GNU Operating System* Available online at http://www.gnu.org/philosophy/free-sw.en.html (Last visited on February 2013).

[12] The Open Source Definition. *Open Source Initiative* Available online at http://opensource.org/docs/osd (Last visited on February 2013).

[13] Categories of Free and Non-free Software. *GNU Operating System* http://www.gnu.org/philosophy/categories.en.html (Last visited on February 2013).

[14] R. Goldman, Richard P. Gabriel (2005) Innovation Happens Elsewhere: Open Source as Business Strategy. (p. 28) *Morgan Kaufmann Publishers.* 424 pages.

[15] D. Nafus, J. Leach, B. Krieger (2006). Free/Libre and Open Source Software: Policy Support. *Integrated Report of Findings. Cambridge: FLOSSPOLS.*

[16] R. A. Ghosh, R. Glott, B. Krieger, G. Robles (2002) Free/Libre and Open Source Software: Survey and Study. *International Institute of Infonomics (University of Maastricht)*

[17] Richard Stallman. Articles: On Hacking. Available online at http://stallman.org/articles/on-hacking.html (Last visited on February 2013).

[18] E.S. Raymond, B. Young. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. *O'Reilly, Sebastopol, CA, 2001.* 279 pages.

[19] J. Preece (2000) Online Communities: Designing Usability and Supporting Sociability. *John Wiley.* 439 pages.

[20] Y. Ye, K. Kishida (2003) Toward an understanding of the motivation Open Source software developers. *Proceedings of the 25th International Conference on Software Engineering in 2006.*

[21] C. Boyer, (2007). Sudoku's French ancestors. *Pour La Science, June 2006, pages 8-11.*

[22] Oficial Webpage of the CHOCO library Project. Available online at http://www.emn.fr/z-info/choco-solver/ (Last visited on February 2013).

[23] G. Yongquin Gao, V. Freeh. Modeling and Simulation of the Open Source Community. *Agent-Directed Simulation Conference, San Diego, CA, April 2005.*