



VocVille - A Casual Social Game for Learning Vocabulary

Prof. Juan Manuel Doderó Beardo
Departamento de Lenguajes y Sistemas Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz

Michel Jensen
michel.jensen@alum.uca.es
mjensen@uni-koblenz.de

Cádiz, Diciembre 2010

Contents

I	Introduction	4
1	Motivation	4
2	Idea of VocVille	5
3	Structure of the Thesis	5
II	Fundamentals	7
4	Definitions	7
4.1	Educational Entertainment	7
4.2	Casual Gaming	8
4.3	Social Gaming	9
5	State of the Art	10
5.1	Facebook	10
5.1.1	FarmVille	11
5.1.2	Facebook GraphAPI	11
5.2	Grails	13
5.2.1	The Framework	13
5.2.2	Groovy	14
5.2.3	Groovy Server Pages	14
5.2.4	Hibernate and GORM	15
5.2.5	Spring	15
5.2.6	Facebook Graph Plug-in	16
5.3	Adobe Flex	16
5.3.1	Flash	16
5.3.2	Flex	17
5.3.3	MXLM	17
5.3.4	Action Script	18
5.3.5	Flash Builder	18
5.4	Grails Plug-ins for Flex Integration	18
5.4.1	Web Service	18
5.4.2	Flex Plug-in	19
5.4.3	Flex Scaffold	19
5.4.4	Flex on Grails	20
5.4.5	GraniteDS	20
5.4.6	BlazeDS4	20
5.4.7	Descision for BalzeDS4	20
III	Conceptual Design	22

6	Game Story	22
7	Target Group Analysis	23
8	Business Model	23
9	Requirements	24
9.1	Non-functional Requirements	24
9.2	Functional Requirements	25
9.2.1	Functional Requirements for the Game Interface	25
9.2.2	Functional Requirements for the Administrative Interface	26
10	System Architecture	26
10.1	Model View Controller	26
10.2	System components	28
11	Use Cases	29
11.1	Build an Object	29
11.2	Show Object/Timer	32
11.3	Activate Area	33
11.4	Invite Neighbor	34
11.5	Visit Neighbor	35
11.6	Send Gift to Neighbor	35
IV	Implementation	37
12	Data model	37
12.1	Vocables	39
12.2	Areas	41
12.3	States of Vocables and Areas	43
12.4	Translations	46
12.5	Designs	47
12.6	Player	48
12.7	Neighbor Requests	49
12.8	Gifts	49
13	Administrative Interface	50
13.1	Game Instances	50
13.2	Game objects management	51
14	Game Interface	55
14.1	Home State	55
14.2	Query Process	57
14.3	Marketplace	58
14.4	Move Tool	59
14.5	Neighbor Requests	59

14.6 Gifts	59
14.7 Neighbor State	60
15 Connection between Grails and Flex	61
16 Deployment	62
V Evaluation	63
17 Analytic Evaluation	63
17.1 Requirements	63
17.1.1 Non-functional	63
17.1.2 Functional	63
17.2 Educational Entertainment	66
17.3 Social and Casual Gaming Aspects	66
17.4 Grails	67
17.5 Flex	67
17.6 System Architecture	68
18 Usability and Accessibility	69
18.1 Game Interface	69
18.2 Administrative Interface	69
VI Conclusion	70
VII Appendix	71
19 References	71
20 List of figures	73
21 List of abbreviations	74
22 Used technologies and tools	75

Part I

Introduction

This paper introduces VocVille, a causal online game for learning vocabularies. I am creating this application for my master thesis of my career as a “Computervisualist” (computer visions) for the University of Koblenz - Landau which I terminate as an exchange student at the University of Cádiz.

In the last one and a half year there is a new trend in the Internet. Little browser games, often called casual games, are played by millions of users. These games are all really simple to play, accessible from every computer with an Internet connection, and in general at no charge. The players normally have very small task to fulfill. To do so they have to do little actions, like clicking on a plant they want to seed, and then wait for a long time. These periods of time are measured in real time. This means the games continue even if the user does not play. Because there are a lot of parallel task with different amounts of time to wait for the next step, the user can nearly at any time do a little bit for his progress. That is the reason why for many people it is possible to play causal games. Because they do not have to learn complex game functionality like in online role-plays like World of Warcraft, these games do not need so much time to play and for a lot of people who have Internet access in their daily life, it is easy to play for a few moments and then go back to work.

In this first part I explain my Motivation for creating this game, give a short introduction of the idea of the game and describe the structure of this paper.

1 Motivation

Casual gamers, like gamers in general, spend a lot of time and mental energy in their games and gather a lot of knowledge related to the game. This knowledge comes automatically by just playing a game. For example after a week of playing FarmVille the users knows which plants he can cultivate and what amount of time every one of them needs to grow. So there are some kinds of learning mechanism taking place while playing a game. In every game, especially in casual games, the user has to repeat task a lot of times. In FarmVille choosing which plant to grow is a fundamental process in playing the game. So the user reads and uses the list of available plants a lot of times. This process is comparable with the process of learning vocabularies. For learning new words in another language one normally repeats reading, writing or listening to the translations. The only difference between learning content of a game and learning vocabularies is the motivation. The content of a game is learned because it is useful in the moment of learning. The motivation for playing a game is not to learn new things but to enjoy the play time. It is also no intended process of the player. On the other hand learning vocabularies is a fully intended process by a learner. The motivation for learning is knowing new vocabularies which can be used to communicate more precisely in another language. These new

words are not used in the moment of learning, like they are used in a game. For the learner the process appears like working. VocVille tries to use the potential of casual games to learn content without a lot of effort by the user to learn vocabularies.

2 Idea of VocVille

VocVille is an online browser game based on the idea of the really successful Facebook game FarmVille[6]. The user can create his own home with everything in it. For creating an object the user has to give the correct translation of it several times. After every query he has to wait a certain amount of time to be queried again. When the correct answer is given sufficient times the object is built. After building one object the user is allowed to build others. After building enough objects in one area (i.e. a room, a street etc.) the user can activate other areas by translating all the vocabularies of the previous area. Users can also interact with other users by adding them as neighbors and then visiting their homes or sending them gifts, for which they have to fill in the correct word in a given sentence.

3 Structure of the Thesis

The thesis is divided into five parts:

- *Introduction:* A short introduction which describes the topic of the thesis.
- *Fundamentals:* The second part gives fundamental knowledge the reader needs to understand the following parts of the thesis. This includes definitions of terms and introductions of the used technologies for the project.
- *Conceptual design:* The conceptual design of VocVille is described in the third part. First I narrate the game story and the world the game takes place in. In the target group analysis the supposed users are characterized. After that I list the requirements for the application and present a possible business model. At the end of this part the architecture of the program and the functionality in form of use cases are explained in detail.
- *Implementation:* The fourth part takes a look at the implementation of the program. In the data model section the class model is introduced and the mechanisms of each part of the game are described from a technical view. This section mainly describes the Grails side of the application. In the game interface section the graphical representations of the game are described. This section mainly describes the Flex side of the application. The next section explains how data is transported from Grails to Flex. At the end of this part the deployment including the integration to Facebook is discussed.

- *Conclusion:* The last part summarizes the thesis and gives a brief outlook on how to advance the application.
- *Appendix:* The appendix includes beneath the references and a table of figures a short glossary and a list of used technologies and tools.

Part II

Fundamentals

This part gives the basic knowledge for understanding VocVille and the environment around it. The definitions section specifies the three application types educational entertainment, causal games and social games. VocVille is an approach to combine aspects of these three types. The second section specifies the technologies which have been used to implement VocVille. These are the social network Facebook, the web application framework Grails and the software development kit Adobe Flex as well as methods of communication between them.

4 Definitions

4.1 Educational Entertainment

Educational entertainment is a not concrete defined concept of products which trying to educate the user by using entertaining activities. There are also the terms “Edutainment” and “Entertainment-education” in use for it. Educational entertainment includes TV productions, films, museum exhibits, computer games, and other educational methods which have a focus on the pleasure of the user while learning. Educational computer games, also often called serious games, are an important part of educational entertainment.

A lot of commercial computer games create very complex environments in which the users has to immerse into. This means for being able to play the game and understand the possible actions one can take, the user has to learn about the game world as well as the game interface. The worlds are mostly simplified models of real world environments. Because the main goal at creating the environment is to improve the entertaining aspect of the game a lot of concepts are changed and do not represent their real world equivalent. Most of the issues learned for a game do not bring any benefit for other situations than the game their were learned for. The learning process is normally integrated into the game so that it does not feel like a real learning activity [15]. It is also a learning process were the learner has an intrinsic motivation because he wants to know how to play the game. A lot of game designers and persons from the educational sector try to build games where the learned knowledge is usable beyond the bounds of the game world. The challenge of creating educational games is to find the right balance between the entertaining and the educational part of the game. These kind of games normally tend to be highly educational but less entertaining or the other way around. This is because the rules for a good, enjoyable game play and the structure of the knowledge mostly are not very easy to combine.

4.2 Casual Gaming

The traditional or sometimes called hardcore computer games are expecting the player to spend big amounts of time for playing. At the beginning it takes a lot of time to learn the mostly very complex usage of the game. The user has to learn how he can manipulate the game objects and how they will react for certain actions. After that period of learning the game the actual playing begins. Most games are divided into levels or areas which represent a closed part of the game story. To complete one of these parts it takes from one up to several hours. In every part the user has an amount of information about the current game situation to keep in mind, in order to success the given tasks. Because of the complexity of this information it is not easy to save the current state of the game and continue another day. Every time the user continues after a longer break he must gain all this information again before he can continue with the actual game goals.

Casual games on the other hand have very easy to understand game controls and game worlds. One of the most famous casual games is the classical Tetris game. It has a really simple game story and it just takes a minute to explain how to play the game. Because of this simplicity of casual games the user can start playing without a long, initial learning procedure. The Casual Games Market Report [14] found that in typical casual games the parts of the game story are really small and in the majority of the games are always the same task with just a little change. These short parts allow the player to play just a short time and then continue later with the game. The user does not have to remember a lot of details of the current game state which also eases to pause within parts of the game. That is why it is very easy to play a casual game just a few minutes but a couple of times a day. The overall play time to complete the game is much smaller than in traditional computer games. All this qualities of casual games allow a much wider range of users to play these games. Not only the typical younger hardcore-gamers who have a lot of experience with computers and games but also people who never played computer games before can use these simple games. Beneath the usability aspect the small time consumption of casual games is one of the reasons why they are playable for persons who usually do not play computer games. Since mobile phones got screen resolution high enough to display simple games, a lot casual games are developed for them. Because of the short playing episodes they can be used to fill waiting times for example in public transportation.

Characteristic	Casual	Hard-Core Enthusiast
Demographic	All ages, male and female, 100% of population	18 - 35 year old males, < 15% of population
Where Play	Home, work, airplane, transit stop	Home
Why Play	Fun, relaxation, escape	Exploration, stimulation, adrenaline rush
Themes	Family friendly scenarios	Sci-fi, edgy violence, horror, suspense, war
Time commitment	Game time or level 1 - 10 min to complete	Levels: 20 min - 2 hours, MMO: 5 - 40 hours/week
Time to completion	Single player story 15 hours to complete	Single player story 15 - 40 hours to complete
Primary Platform	PC, Mac, inexpensive console	Game consoles, hi-end PC
Game Price	Advertising supported - \$19.99	\$39.99 - \$59.99
Game Selection	Free trials, up-sells	Marketing campaigns, reviews and previews
Hollywood Equivalent	Sex and the City, Friends, ER	Horror, Silence of Lambs, Reservoir Dogs, Aliens

Table 1: Gameplay Characteristics of Casual & Hard-Core Enthusiasts, [14]

4.3 Social Gaming

Nearly every actual game has a multiplayer option which allows the user to play with or against players on other computer within their local network or over the internet. So called massively multiplayer online games (MMOG) are just online playable. The user creates an account on the server of the game developer where his game status is stored. MMOG have a lot in common with traditional hardcore games, like complex game worlds and time consumption needed to play successfully. Social games on the other hand can be seen as the online version of casual games. Their game world is less complex and the time consumption is as low as the ones from casual games. Social games have a main focus on the online aspect. The players can either play synchronously together or interact asynchronously. Playing synchronously means that the users play at the same time and see the same part of the game world. They also see a representation of the other player (e.g. the avatar¹ of other player) and what actions he takes. Playing asynchronously means that the users do not have to

¹Avatar: graphical user's representation of himself/herself

play at the same time and every player sees his own part of the game world. If they take actions that influence others the game of another player for the other player these actions happen when he plays the next time. Although games with these characteristic exists before, the term social game is used for them since they are deployed through social network sites. Because millions of users already have accounts at at least one social network, it is really easy to access social games. In the stand alone game sites the user has to invite and convince their friends to register for the game and play it, in social networks the friends are already connected and registered.

5 State of the Art

5.1 Facebook

Facebook is a social network site which means that registered users can create profiles about themselves and share them with other users of the network. It was founded in 2004 by four students of the Harvard University. Today (October 2010) it has 500 million active users. With the registration process a personal profile for the user is created. This profile shows information like name, age, hobbies, interest and a lot of other depending on the users' security settings. An important part of the profiles is the so called Wall, where actual information about the users' activities on Facebook are displayed. Users can connect their profile with profiles of other users by declaring them as their friends. To create this connection both of the users have to agree to it. For communicating with other users there are three different options. Similar to an email system the users can send private messages to another user who receives them in his inbox. It is also possible to write on the Wall at the profile of another user. This means that everybody who is allowed to view the users profile also can read this message. The last option is a chat program which works like any instant messaging application but also is within the Facebook site. Another reason for a lot of user to use Facebook is the possibility to upload own photos, manage them in albums and share them with other users. There are a lot of more functions on the Facebook site, but describing them all is beyond the scope of this thesis.

The function which is interesting for this thesis is the possibility to use and create applications and games within the Facebook network. At the moment (October 2010) there are 550,000 applications available on the network [13]. Users can subscribe for applications by going on the site of the application (within the Facebook network) and click on a start application button. Users can also recommend other users application they use by their self. At the first start of an application the user has to allow it to access his user data. The applications are integrated in the Facebook network, which means they have access to the users data (see 5.1.2)including friend-lists, email addresses etc. They can also generate message for publishing on the wall of an users profile.

5.1.1 FarmVille

The most famous one of the Facebook-applications is the casual game FarmVille [6] by Zynga [11] which is accessible through the Homepage of the developers, through Facebook, which is one of the reasons of the success of Farmville, and since June 2010 through iPhones and other mobiles phones. It was released in June 2009 [12] and after two months it already got 35 million players. At the moment there are 70 million players [7].

The purpose of the game is a simulation of a small farm. The user can plant crops and earn money by harvesting them. Between the seeding and the harvesting it takes, depending on the chosen plant, two hours or up to three days. For the earned money one can buy new seeds, animals, building or decorations for the farm on the market. For every action the user performs he earns experience points which are needed to get to a higher level. Higher levels allow using better plants and buying more different objects. A really important aspect of the game is the possibility to invite other players of the users friend network of Facebook to be their neighbor in FarmVille. Neighbors can visit each other's farms and see what their friends are doing. They also can help them by fertilizing the plants or feeding their animals which improves the production of these objects. The visitor gets experience for this help and the owner of the farm gets more money when harvesting. Another way of interacting with neighbors is sending gifts to them. Once a day every user can send every neighbor a free gift which can be anything also available on the market. The developers invent continuously new ways of interacting with neighbors. In higher levels it is possible to build special building like a horse stable for which the user needs construction materials one can only get as a gift from neighbors.

FarmVille is a very good example of a causal browser game. It is very easy to play, at the beginning there are a few short text message which explain how to plant and harvest. Everything else is explained later when the specific function becomes available. Because it is playable within the Facebook site it is very easy to access it since a lot of people already have a Facebook account. It does not take a lot of time every day and with the concept of neighborhood it has a big social component. Mark Skaggs, VP and GM of Zynga's Social RTS studio described it in one interview with these words: "By combining the best elements of social gaming, with people's instinct to nurture, we've created an incredibly fun, wholesome and rewarding experience." [12].

5.1.2 Facebook GraphAPI

The Facebook Graph API allows developers of Facebook application accessing the user data the Facebook site has stored for each user. This data is called open graph (formerly social graph). The open graph of an user contains all stored data about him like name, friends and others. The objects of the open graph can be accessed through simple HTTP-Request. The response data are objects in the JavaScript Object Notation (JSON) which can be treated by the most web application technologies. The API also allows webmasters to use the

authorization function of Facebook for their own internet sites. This means that the website owner does not have to manage the user accounts and the user does not have to create an additional account for that page. More than one million websites use this feature at the moment (October 2010)[13].

The Graph API Reference [4] defines the following 19 basic object types:

User An user profile.

Application An application registered on Facebook Platform

Page A Facebook Page

Group A Facebook group

Photo An individual photo within an album

Album A photo album

Checkin A checkin made through Facebook Places or the Graph API

Comment A comment on a Graph API Object

Event A Facebook event

FriendList A Facebook friend list

Insights Statistics about applications, pages, or domain

Link A shared link

Message A message in the new Facebook unified messaging system

Note A Facebook note

Post An individual entry in a profile's feed

StatusMessage A message in the new Facebook unified messaging system

Subscription A subscription to an application to get real-time updates for an Graph object type

Thread A message thread

Video An individual video

Every instance of these objects has a unique id with which it can be accessed. With additional query-strings the desired response attributes can be specified. The objects can have relationships, so called connections, between each other. With these connections the plain user data becomes a data structure with the form of a graph. For getting a list of all connections of an object the "meta-data=1" parameter can be used. The Graph API uses the Oauth 2.0 Protocol [9] to control the access of the open graph. If an application wants to access nonpublic data it has to get an access token for the desired data. For getting

access tokens the user of the application has to confirm the access once. With the correct access token the application can create new objects (like messages on the wall) and manipulate or delete existing ones. With real-time updates it is possible to inform an application when user data has changed and take appropriate reactions.

5.2 Grails

For the main program logic of VocVille I needed a framework which could handle a large amount of data in a relational database, supports rapid development and facilitates the development of web applications. I found these requirements in the open source web application framework (WAF) Grails.

Grails is based on the dynamic programming language Groovy. It is highly inspired by the WAF Ruby on Rails[10]. Actually the former name of Grails was Groovy on Rails, but because of the likeness of the two names the Ruby on Rails developers team asked for a change of it. Like Ruby on Rails also Grails is open source. The main goal of Grails is to make web development in the Java environment more productively. It follows the “Coding by convention” paradigm, which says that a developer should only change the aspects of his application which are not complying with the coding convention used by the majority of developers. Grails uses the Model View Controller design pattern (MVC) to separate the data, the user interface and the programming logic from each other. MVC is described in detail in section 10.1.

5.2.1 The Framework

The Grails framework consists of several frameworks and technologies which are shown in Figure 1. Beneath the Java components there are the already mentioned programming language Groovy, the object-relational mapping (ORM) Hibernate, the Java application framework Spring and the layout-rendering framework SiteMesh. Because for the development with Grails one has to work mainly with Groovy, Hibernate and Spring I will discuss them in more detail in the following sections. Grails binds these technologies together and configures them in a standardized manner following the Coding by Convention paradigm. This means, that when building a new Grails project, the framework creates a prototype like runnable application. The developer then can start with creating the desired model and manipulating the views and controllers. Additional functionality can be added by the very easy to use plug-in management from Grails. With just on command-line command a new plug-in can be installed to a project. The Grails community provides a variety of different plug-ins, which is constantly growing because everyone can create and publish own plug-ins.

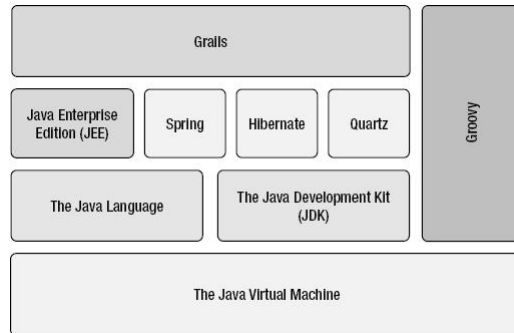


Figure 1: Grails architecture

5.2.2 Groovy

Groovy[8] is a dynamic programming language similar to Python, Ruby, Perl or Smalltalk. As such it executes at runtime common behaviors, like adding code, extending or defining objects, modifying type streams and a lot of more. It can be used as a scripting language for the Java platform. Grails code can interoperate with other Java code and libraries, because most Java code is syntactically Groovy code. Java developers can use their knowledge and techniques and expand them with Groovy techniques. Groovy code is compiled into Java classes which makes it theoretical usable in any Java application. Although Java code works fine in Groovy, the Groovy code is more compact because it does not require all syntactical elements of Java. For instance parentheses, return statements and semicolons at the end of statements are optional in Groovy. As a true object oriented programming language Groovy represents even basic types as objects and operators as methods. Many existing Java classes functionality is extended by additional methods. There are many other features which improve development with Groovy like closures, intuitive syntax for lists and maps, metaprogramming and ranges among a lot of others. Describing all of them is beyond the scope of this thesis. For further information see the documentation of Groovy. In the Grails framework the domain, controller and service classes are written in Groovy.

5.2.3 Groovy Server Pages

Groovy Server Pages (GSP) is a presentation language for web applications similar to Java Server Pages (JSP). It describes static and dynamic content in the same document. In the Grails framework GSP is used to describe the views. GSP offers a bundle of built-in tags which implement basic program logic like conditions and iterations. Like in JSP and a lot of other presentation languages GSP allows implementing program code in script blocks. Inside these blocks regular groovy code can be used. All built-in Tags are also available as Groovy methods and therefore can be called in script blocks as well as in Controllers.

5.2.4 Hibernate and GORM

Hibernate is an ORM for Java and, considering its high usage among the Java development community, can be seen as a de facto standard. It maps object-oriented models like Java classes to relational databases entities. The software developer can handle the data like normal objects of his application and does not have to know the underlying database queries. Through XML or Java Annotations he can decide how to store the data in the database. In this manner the application can achieve persistence by storing and retrieving data between several runtimes.

The Grails Object Relational Mapping (GORM) is an ORM implementation for Grails that is based on Hibernate. Like Hibernate GORM supports static typing and additionally dynamic typing. It is adjusted to the needs of Grails and its conventions so there is less configuration involved in creating domain classes. In general every domain class in Grails is mapped to a database table by GORM. The properties are mapped to a corresponding column in the table. GORM supports several types of relationships between objects. Beneath bidirectional and directional One-to-One relations also One-to-Many and Many-to-Many relations can be realized. By setting the injected static properties like *belongsTo* and *hasMany* the developer can define these relation types. There are also static properties to define constrains or even to change the GORM mapping directly.

5.2.5 Spring

Spring is an open source application framework for the Java platform. It extends the Java language in several aspects separated in various modules to facilitate creating and handling the infrastructure of applications. Many developers use it as a replacement or addition to the Enterprise JavaBean model, even though it supports a lot of other programming models. Spring supports implementing the Inversion of Control pattern by providing techniques for using callbacks². The integrated Spring Aspect-Oriented Programming Framework supports aspect oriented programming by integrating with AspectJ. The Data Access Framework provides template classes for several databases like JDBC, Hibernate and Oracle TopLink. The MVC framework and the Remote access framework provide functionality to create web applications with the MVC pattern. Beneath the mentioned modules and frameworks there are a lot of more but describing them all would be beyond the scope of this thesis. For more information see the documentation of Spring.

Grails uses the Spring framework intensively. Actual every Grails application is an (of course modified) Spring MVC application. The basic controller logic uses subclasses which inherit from Spring classes. Grails validation and data binding methods are built on Springs MVC functionality. Runtime configurations of Grails applications are stored in a Spring ApplicationContext-Object. Also GORM uses the transaction management of Spring for transactions.

²Callback : Using functions as parameters for other functions

5.2.6 Facebook Graph Plug-in

For connecting Grails with Facebook I will use the Facebook Graph Plug-in[5] since there is just one other Grails plug-in for Facebook connectivity, namely the Facebook Connect plug-in[3], which is in no actual development and also does just support the outdated Facebook Connect Service. The Facebook Graph plug-in offers GSP-tags for the login process and a Grails service called *FacebookGraphService* to access the social graph. The service includes methods for requesting the users' profile, the friendlist, the profile photo, and for publishing on the users' wall. An additional method gives access to the complete Graph API.

5.3 Adobe Flex

To let VocVille be an enjoyable and easy to understand game I wanted it to have a good look and fell. For accomplishing this I chose the Adobe Flex software development kit (SDK) for creating a Flash application. With Flex one can easily build a good looking user interface. Another reason to use Flex was that it compiles to swf files which can be played by the Adobe Flash Player. Because today the Flash Player is installed on the majority of the users systems using the format helps achieving the accessibility requirement for the application.

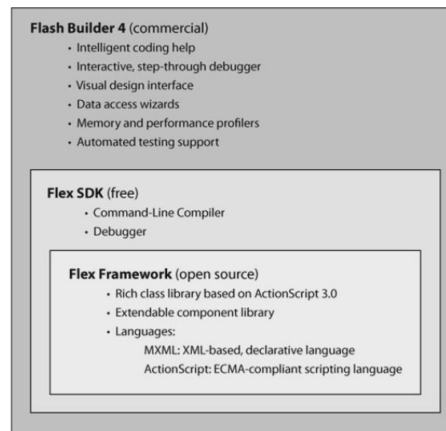


Figure 2: Flex architecture

5.3.1 Flash

Adobe Flash is a proprietary development platform owned by the Adobe Systems Incorporated. The purpose is the creation of interactive multimedia content. It allows creating applications, so called Flash Movies, which can contain text, video, audio and navigation controls.

These applications can be run within a virtual machine called Flash Player. The Flash Player is a browser plugin which is available for nearly every actual browser and all often used operating systems like Windows, Linux, MacOs and others. There is also a desktop version of the virtual machine called Adobe AIR but at the moment it is rarely used. The Adobe Flash Professional program is mainly intended for designers to build graphical parts of Flash applications, also it can build a complete application that just needs very simple program logic. The Adobe Catalyst program is designed to be used by both, designer and programmers because it allows the user to give graphical components a programmatically logic for their visualization. The third program, Adobe Flash Builder (formerly Flex Builder) is intended to be used by programmers. It has a graphical view but also allows manipulating the source code directly. In section 5.3.5 I will describe the Flash Builder tool in more detail.

5.3.2 Flex

The Adobe Flex software development kit is a web application API for creating rich internet and desktop applications for the Flash platform. It was created to be a development platform for programmers, because before there was only the Adobe Flash Professional program available which aimed web designer with no or less programming experiences. The applications are mostly the client side in a client-server architecture. For communicating with servers there are three remote procedure call (RPC) methods: HTTPService, WebService and RemoteObject. The Flex sdk itself is open source. It combines the XML dialect MXML and the scripting language Action Script. MXML is used to build the graphical interface and action script is used to build the program logic. I will describe both in more detail in the following sections.

5.3.3 MXLM

MXML is an XML dialect for creating graphical user interfaces (GUI). It provides components for typical GUI-elements like labels, buttons, and menus, among a lot of others. The predefined components can be modified by changing their properties within their corresponding tags. It is also possible to create own versions of the elements by inheritance using Action Script classes. MXML also offers a variety of options to layout the components and changing their graphical appearance by using styles. The language is composed of three libraries. The mx-library, also known as “Halo”, includes components for charting and data visualization as well as classes from Flex Version 3. The fx-library includes top-level Action Script tags, like the `<fx:Script>`- tag with holds Actions Script for the MXML file. The s-library, also known as “Sparx”, includes new versions of Flex 3 mx-components, references to the RPC-components and the text framework which are introduced in Version 4 of the Flex framework.

5.3.4 Action Script

Action Script is an ECMA-compliant scripting language that offers object oriented programming (OOP) in the Adobe Flex sdk. In a Flex application it is used for any programmatically logic like event handlers for buttons. As an OOP language it has the concepts of classes, instances, methods and inheritance. The Action Script code can be written directly into the `<fx:Script>`-tag of an MXML-file or in separate Action Script files.

5.3.5 Flash Builder

The Adobe Flash Builder[1] is an integrated development environment (IDE) for Flex applications build on Eclipse[2]. In contrast to Eclipse the Flash Builder is a commercial product. It offers developers to view applications in two manners. The visual mode shows the GUI-elements like they would be visualized in the running application. In this mode the components can be manipulated like in design tools. They can be inserted by drag and drop and their layout can be manipulated by mouse commands which facilitates the creation process of GUIs. In the code mode the source code of the files is shown, as in any other programming editor. Typical features like syntax highlighting, code-completion, debugging mode and basic refactoring methods are available. Selecting a component in visual mode and then switching to code mode opens the editor at the corresponding line of the code where the component is defined. Since version 4 of the Flash Builder so called data services are introduced. They are wizards to connect the application with data from remote data sources like BalzeDS, ColdFusion, HTTP, PHP, or WEBServices, among others.

5.4 Grails Plug-ins for Flex Integration

As I mentioned before I wanted to use Adobe Flex as the graphical front-end of the application. For this purpose I needed to send the output data of the Grails functionality of VocVille to the Flex framework. At the moment there are seven Grails plug-ins to connect Grails with the Flex framework. In the next sections I will describe how I tried to build test applications with each off them. At the end of this chapter I will explain my decision for using the Blazeds plug-in. Beside these seven plugins there is also a plug-in called Flash Player plug-in which only supports playback functionality for swf-files within an Grails application.

5.4.1 Web Service

The first method I tested was using the XFire plug-in to create Web services which Flex can access through its RPC-functionality using SOAP-Messages³. The XFire plug-in allows developers to transform their normal Grails services into Web services by simply changing the static property “expose” of the service

³SOAP = Simple Object Access Protocol

to 'xfire'. This will instruct XFire to create a WSDL-file⁴ for the service and publishing it on the applications server. Given this description file of the service, Flex can access the service methods. To register a Web service in an MXML-file the Spark component `<s:WebService>` is used.

With this method I was able to receive and add data entries. When I tried to update data entries Flash threw an error message indicating that it has synchronization problems. Another drawback of this method was that when I used a database with 65 thousand entries, the SOA-Protocol was not able to send this amount of data. This may be fixed by dividing the data into smaller packages.

5.4.2 Flex Plug-in

In the second approach I used the Flex plug-in. The plug-in was created as an experiment to proof the concept of connecting Grails with Flex and was not further developed since 2008. It offers an embedded Flex compiler which allows Grails developers to use MXML-files in their application. If an MXML-file is access by the Grails application the Flex compiler compiles the MXML-file into a runnable swf-file. The plug-in can also expose Grails services as remoting destinations which can be access by a Flex application. To expose a service its static expose property has to be set to 'flex-remoting'. In the Flex application the service can be access by an `<mx:RemoteObject>` component.

In my test application I was able to load MXML-files and let the embedded compiler compile them into swf-files. Admittedly the Flex application could not localize the Grails service. Another disadvantage of the Flex plug-in is that it just works with old Flex3 mx-components. The new Sparx-components introduced in Flex 4 are not available which include a lot of new features for the Flex framework.

5.4.3 Flex Scaffold

The next approach used the Grails Flex Scaffold plugin which scaffolds Flex code of views and controllers for Grails domain classes like the Grails scaffold method generates GSP code. The project has not been updated since December 2009. The plug-in gives domain class properties additional properties which define how a domain entity should be represented in the Flex application. It supports a variety of Flex components to visualize the data and its input views.

In my test application it was not possible to create a running application because the, internally used by the plug-in, Cubika classes could not be found. I suppose the plug-in would work properly with appropriate older versions of Grails and Flex but this would mean, that I have to use very old version of the frameworks including all bugs of them.

⁴WSDL = Web Services Description Language

5.4.4 Flex on Grails

This approach uses the Flex plug-in combined with the ActiveMQ and the JMS plug-ins. The Flex plug-in is used to run the Flex application on the server of the Grails application. For the communication between Grails and Flex the Java Message Service (JMS) is used. ActiveMQ serves as the message broker by translating the Grails objects to a JMS concurrent format.

My test application ran correct even with 65 thousand database entries. But because this approach uses the Flex plug-in it also does not support new Flex 4 components. Another disadvantage of this method is that it requires the Flex application files to be nested inside the Grails application directory tree which means that it is complicated to open it in the Flash Builder.

5.4.5 GraniteDS

This approach uses the GraniteDS Flex plug-in which sole release had been published in March 2008. It offers automatic code generation by using additional annotation for the Grails domain and service classes. The annotated Grails objects are exposed as Flex remoting destinations and can be access by Flex applications. The plug-in also includes a Flex compiler allowing MXML-files being compiled by the Grails application.

Since I discovered this method in a later state of the designing process of VocVille where I already decided which method to use for the Grails-Flex-integration I did not build a test application for it. Another reason was that this method also does not support the new Flex 4 components and just works properly with Flex 3 mx-components.

5.4.6 BlazeDS4

The last approach uses the BlazeDS 4 Integration plug-in. This plug-in uses the Remoting plugin-in which implements four different protocols in the Action Messaging Format (AMF). Hence AMF is a binary format it allows fast transportation of a huge amount of data. The BlazeDS 4 Integration plug-in furthermore installs the BlazeDS 4 and the Spring-BlazeDS integration plug-ins. BlazeDS 4 is used to expose data from Grails to the Flex framework. At the moment this is the only approach that supports the new Flex 4 components. The only disadvantage of the plug-in is, that it needs the not actual version 1.2.2 of Grails to run properly.

5.4.7 Descision for BalzeDS4

After I tested all available methods to connect a Grails application with a Flex front-end I decided to use the BlazeDS4 plugin because of the following reasons. I could not build a running test application with the Flex Scaffold therefore this method was no option. Because I wanted to use the actual version four of the Flex framework and the corresponding version of the Flash Builder the Flex plug-in, the Flex on Grails approach as well as the GraniteDS plug-in did

not match my desired criteria. The Web service solution did not come into consideration because it does not allow updating data and also does not work with a large amount of database entries. The only method fulfilling all my desired criteria is the BlazeDS4 approach. It supports Flex 4 components, runs with a large amount of data entries and is still maintained. The only deficiency at the moment is that it does not support the current version of Grails. This will hopefully be solved in the next version of the plug-in. Table 2 shows an overview of approaches, what I was able to realize in a test application and what problems occurred.

Approach	Plug-in	Realized	Problems
Web service	XFire	- receive data - add data to database	- update data - 65 thousand entries
Flex Plug-in	Flex	- compile MXML -site on server	- access Grails service - just Flex 3 supported
Flex Scaffold	Grails Flex Scaffold	- no runnable application	
Flex on Grails	Flex, ActiveMQ, JMS	- receive, add, update data - 65 thousand entries	- just Flex 3 supported
GraniteDS	GraniteDS Flex	- not tested	- just Flex 3 supported
BlazeDS4	BlazeDS 4 Integration	- receive, add, update data - 65 thousand entries - Flex 4 components	- just Grails 1.2.2 supported

Table 2: Integration methods for Grails and Flex

Part III

Conceptual Design

6 Game Story

The story of the game is that the user owns his house. He can build objects like furniture, new rooms or new family members. At the beginning the user has to build himself, his wife and the house. Starting with the three, probably already known by the user, easy words “man, women and house” the user will learn how the building process functions. For this reason, and for giving the user a first feeling of success, the waiting time between the queries for the first words will be short. After building these first three objects the user can chose what he wants to build next.

For buying objects the user needs gold. He can earn it every time when he answers a query to a vocabulary correct, he finishes building an object, activates a new area or if he translates already learned vocabulary. Besides gold there is another currency in VocVille. The so called Wisdom Stones are much more worth than gold. But there are really rare in the game. Just when the user achieves a higher level he earns one Wisdom Stone. There are also some objects that can only be paid with this currency. Both currencies may be bought for real money.

Really important for the game experience and the motivation to play the game are the social gaming components. Every user can invite other users from his Facebook friend list to be his neighbor in VocVille. Only friends who themselves play VocVille can be neighbors. Other Facebook users will be asked if they want to start playing the game before they can be a neighbor. Neighbors can visit each other’s homes. This means, that if a user clicks on the image of the neighbor on the lower bottom of the game interface, and choses to visit the neighbor, he will see the house of the neighbor. Now he enters the rooms and sees the objects and statistics of them. He can also help the neighbor with objects the neighbor is constructing at the moment. By clicking on an object that is in construction the user helps his neighbor constructing it. This means the next time when the owner of the object has to translate the vocabulary he will get a little tip, like one letter of the answer. The helping neighbor will get a little amount of gold for his help.

Another way of interaction is the option to send gifts to neighbors. These can be points the user has to collect to get a new design of one object, new objects or other things that can help or decorate the house of the friend. When the neighbor receives the gift he has to solve a question in order to use the object within the gift. The questions are sentences with blank spaces where the user has to fill in translations of vocables. A few possible words are shown and the user has to select the right solution. Which vocabularies are being used for the questions depends on the vocabulary the user has already learned. The user gets the gift if his answer is correct otherwise the gift disappears. Every user

can send every neighbor one gift per day.

Because of the design of the application which is separated into an administrative application and the actual game it is very easy to create another version of the game and therefore also changing the game story. In such a way one could create “Vocables” which are mathematical exercises and the “Translation” would be the solution for it. In this manner VocVille could be used for a lot of different learning situations. The concept of Game Instances which represent different versions of the game are discussed in section 13.1.

7 Target Group Analysis

In this section I will describe the assumed user group for VocVille. Knowing the target group for an application has the benefit to be able to design the system exactly for the needs of this group. Since social networks like Facebook are used by the majority of younger people it can be assumed that this will also count for VocVille. Also the subject of learning a foreign language is usually more often in the scope of younger users. As mentioned before players of casual games are from all ages or genders. Therefore the target group of VocVille also will be consisting of people from all ages or gender but with a majority of younger people who have more likely access to Facebook.

8 Business Model

The application will use the business model of selling virtual goods. This means that the user has the possibility to buy a virtual currency called Wisdom Stones. The Wisdom Stones allowing the user to buy special items, get help on the game tasks, and use options which enhance the usability of the game. To get Wisdoms Stones one has to pay real money by a credit card, Paypal or other payment methods. Every time the user gets a higher level he also gets one Wisdom Stone. This mechanism can be seen as an advertising instrument to show the user what he can do with Wisdom Stones and raise the will to pay for more. The virtual goods market for the United States for 2010 is expected to have a value of \$1.6 billion and will grow even more in the following years[16]. This demonstrates that people are willing to pay for virtual goods or, as they can also be seen as, for services.

The following list shows the advantages one can buy with Wisdom Stones:

- Reduce timer: The time the user has to wait till he can be queried about a new building vocable can be reduced.
- Easier area activation: The number of compulsive built objects in the previous area for activates a new one can be decreased.
- Tips for vocabulary queries: Helping mechanisms like showing a few letters of the answer or changing the query into a multiple choice question can be bought.

- Special items: Decorative items that can just be bought with Wisdom Stones.
- Special designs: Decorative designs for objects that can just be bought with Wisdom Stones.
- List of vocabulary: A list of vocabulary can be printed.

9 Requirements

Software requirements are typically separated into functional and non-functional requirements. Functional requirements describe the functionality of an application or in other words what “does” an application. Non-functional requirements describe qualities of an application or in other words “how” does it behave. The next two sections list the functional and non-functional requirements of VocVille and specify them.

9.1 Non-functional Requirements

- Available in Facebook: The application is accessible through the Facebook platform. Therefore it is registered as a Facebook application. The users can add it to their Facebook account and start it when logged in to Facebook.
- Easy to use: The usability is a very important aspect of VocVille. The game interface should be intuitive to be used and hence the user can start playing instantly. Throughout the process of the game new available options will be explained in short tutorials.
- Appears more like a game than a vocabulary trainer: Using VocVille should feel like playing a game. The user should use it because it is fun to play. The aspect of learning is the second goal of VocVille but it should not appear like an ordinary vocabulary trainer which appears more like working.
- Nice graphics: The graphics to visualize the game object should be in a nice comic style. They help to identify the player with his avatar and his house.
- Playable in every browser: VocVille should be runnable in every browser which has a recent version of the Flash Player plug-in installed. This also guarantees that it will be runnable in every operation system which fulfills these criteria. Using HTML5 instead of Flash for the frontend of the application would also achieve this requirement without obligating the user to install the Flash Player plug-in. But since HTML5 is expected to reach the Candidate Recommendation stage during 2012[20] and therefore it is possible that elements of the recent version of the specification will change I decided to use Flash.

9.2 Functional Requirements

9.2.1 Functional Requirements for the Game Interface

- Create player account: When a player starts the game for the first time a new player account is created for him. The player can choose a name, a picture for his avatar, the language he wants to learn and the language the game interface should use.
- Display home of the user: When the game starts the player sees his home and the objects he has purchased so far. Already created objects are painted by their normal visual representation. Vocables and areas that are in creation (building or activation process) are painted by another picture to indicate their recent status.
- Save current game state persistent: When the player ends the game application and restarts it again the game is shown in the state in which the player left it before. This includes all areas and objects and their building status among other state informations.
- Building objects: The player can buy vocable objects in the market and build them by translating the vocable of the object several times.
- Vocabable Query: A query of a vocable shows the vocable in the language which the player has chosen to use for the game interface. To solve the query the player has to input the translation of the vocable in the language he has chosen as his learning language.
- Time between vocabable queries: Between the queries of a vocable in the building process the player has to wait a certain amount of time. Between the queries the player can do other action within the game or end the application and open it at a later date.
- Show timer for vocabable: When the player hovers the mouse pointer over an object which is in the building process a timer is shown which displays the time the player has to wait till he can query this object again.
- Highlight elements with timer zero: The player can quickly indicate which vocabable objects can be queried because they are visual highlighted.
- Statistic for vocables: When the player hovers over a vocable object a tooltip displays how many times the player has translated the vocable correct and how many times incorrect.
- Invite neighbors: The player can invite other players to be his/her neighbor. When the other player accepts both become neighbor of each other and can interact within the game.
- Visit neighbor: The player can select a neighbor in a list for visiting a neighbor's home. The application then visualizes the home of the neighbor.

- **Help neighbor:** In the neighbor visiting state of the application a visiting player can select vocables which are in creation state and decide to help the neighbor. This will give the neighbor a little bonus for this vocable object. Every player can help each neighbor once each day.
- **Gift neighbor:** The player can select a neighbor in a list and send him/her a gift. The gifted neighbor has to solve a question to receive the gift. Every player can send one gift per day to each of his neighbors.

9.2.2 Functional Requirements for the Administrative Interface

- **Login mechanism:** Users have to log in with their user name and their password to access the administrative interface.
- **User account:** Only administrators can create user account.
- **Create GameInstances:** Users can create a new instance of the game by creating or selecting areas and vocables within the areas.
- **Edit current GameInstances:** Users can edit current game instances by changing the order or other properties of areas and vocables.
- **No access to player data:** The user cannot access or manipulate data which saves information about the current game state of players. Just the templates to create game objects can be modified.
- **Create new game object:** The user can create new game objects as templates which are used by the game interface to generate the game.

10 System Architecture

As a web application VocVille realizes the client-server architectural pattern. The server side is implemented by a Grails application. The client side is implemented by a Flex application. To use VocVille the user accesses the Flash application within the Facebook website or from the original website of VocVille. This application requests data of game objects and state information from the Grails application and visualizes them in the game. Every action in the game is reflected as a change of the database and therefore transferred back to the Grails application.

Another architectural pattern realized by VocVille is the Model View Controller (MVC) pattern. VocVille uses the pattern in different ways which will be described in the next section after I give a short overview of MVC.

10.1 Model View Controller

The MVC pattern describes a software architecture where the system is separated in the three parts model, view, and controller. The separation allows changing one of the parts without affecting another one. Figure 3 demonstrates

how these parts interact between each other. The model describes the data model of the system domain. It consists of data objects the application creates, stores, displays and manipulates. From an object oriented programming point of view the model represents the part of the world which is described by the application. The view consists of the visualization of the model for the user. The different views are templates which are filled with the data of the model. The controller consists of the program code which handles user inputs and delegates the data from the model to the corresponding view. MVC is in widespread use for web application development although it can be used for desktop applications.

The Grails framework is designed to build MVC applications. Therefore every Grails project has separated folders for the model (`\grails-app\model`), the views (`\grails-app\views`), and the controllers (`\grails-app\controllers`). The models are called Grails Domain classes. Domain classes are Groovy classes which are located in the model folder of the Grails project. They use GORM methods to describe the model. GORM allows to define constraints for the properties of the model and how to store them in a database. The views are Groovy Server Pages (GSP) which are located in the views folder. GSP-views usually receive a model from a controller and display it. They also can redirect to another controller. The controllers are Groovy classes in the controllers folder which by convention consist of a name (usually the domain class they are handling) and the suffix “Controller”. Controllers define so-called actions. Actions are methods which are mapped to an URI of the Grails application server. For example the action “show” of the `VocableController` can be accessed through the URI “APP\Vocable\show\1\”, where APP stands for the name of the application and 1 stands for an id of an instance of the `Vocable` domain. Actions can also be invoked by the GSP-views, which then render the resulting model.

Although the Flex framework does not force developers to use MVC like Grails does it supports the implementation of MVC applications. The data accessed by the RPC methods can be seen as the model. It is also possible to use the Data Transfer Object pattern (DTO) to implement ActionScript objects for representing the models data. Obviously the MXML-tags for the control and data display elements are the views. They allow creating a user interface that visualizes the data. The ActionScript code within the `<fx:Script>`-tag which defines the program logic can be seen as the controller of an Flex application. It handles the user input and accesses the data.

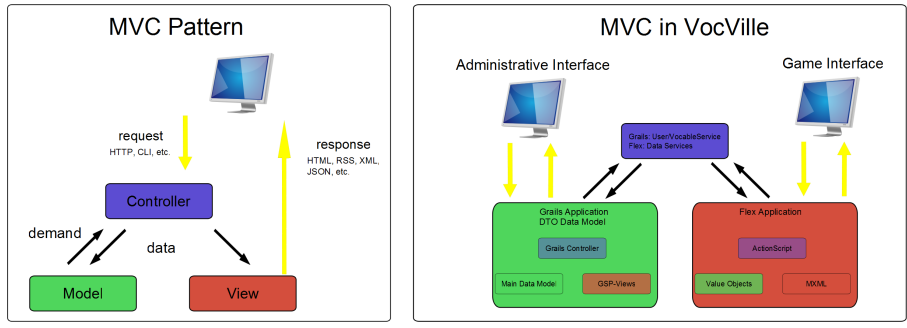


Figure 3: The MVC pattern

10.2 System components

The architecture of VocVille can be seen as a multilayered MVC architecture which is visualized in Fig.3. On the overall layer Grails functions as the Model, the Flex application functions as the View and the RPC-parts of both function as the Controller. On a lower level both parts can be seen as a separate MVC architecture as described in the previous section.

Grails creates, updates, and retrieves the data of VocVille to the database. Therefore it defines the data model in its domain classes. It also runs the server which allows publishing the data to the Flex framework. The Grails services handle the incoming requests from the user through the Flex application. The services of the Grails application are one part of the controller in the overall MVC architecture. On the Flex side the RPC-services are the other part of this controller. They transform user input into services request for the Grails application which then transforms these requests into database queries. The replies send to the RPC-services of Flex are visualized in the user interface of the Flex application which means that it represents the view of the overall MVC-architecture.

The data model of VocVille is separated into two sections. The administrative section holds the user account and the corresponding roles for them. It also holds the content for the actual game. This includes the vocables and their translations and questions, the designs as well as the gifts. The administrative part cannot be accessed through the Flex application. It is meant to be managed by a team of the provider of VocVille. This could include a teacher who decides what vocabulary should be learned with VocVille, a designer who creates the graphics for the game elements and a game designer who defines the balancing of prices of game objects, the experience points needed to get a higher level and other values influencing the game play. Additionally he can support the teacher to arrange the order in which the vocables and areas are allowed to learn. To access the administrative data there is a web front end of the Grails application which uses the normal Grails GSP-views. This front end will not be accessible

through Facebook and hence also not for the players of the game.

The user section of the data model of VocVille holds the game state information of the users. Here are the game actions executed by the player represented. This section includes the avatar of a player which describes his amounts of the two currencies, his language preferences, his level, and his visual representation. Furthermore it includes the home of the player which stores vocables the user has used in his game state as well as the users areas and gifts. VocVille uses the information of the administrative section like templates to create the game objects for the player. The user section of the data model is only manipulated by the Flex application when the user actually plays the game.

11 Use Cases

Fig. 4 shows the use case diagram for VocVille. It displays the six use cases which each describe one possible action a user can take and how the application reacts to this action. Each of this six use cases and their corresponding activity diagrams will be described in the following sections.

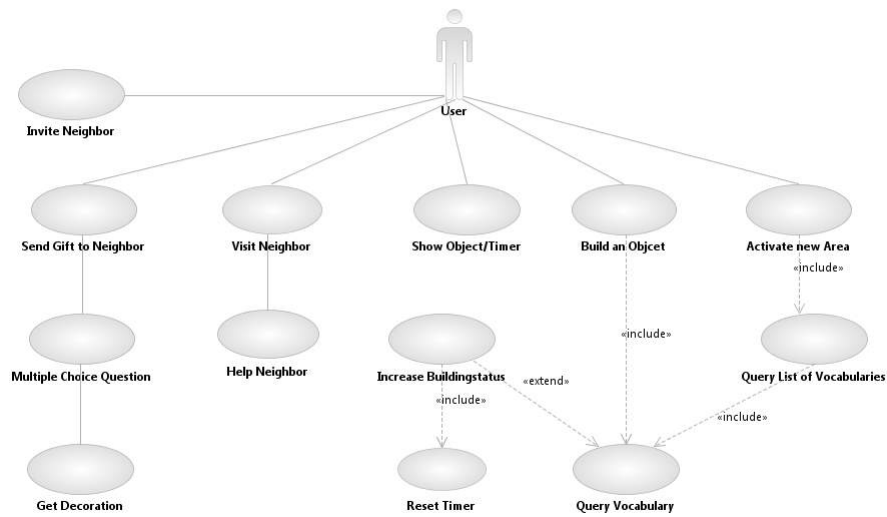


Figure 4: Use Case Diagram for VocVille

11.1 Build an Object

Building objects is the main process in VocVille. When the user wants to build an object he has to buy it in the market. It is depending from the user level and from the vocable objects he has built before which vocables the player can build. If he is allowed to build the object and can afford it, the user can place it in the area of the object anywhere he wants. With placing the object the building process begins. First the translation of the name of the object is displayed for

a short time. Then the user has to wait a short time before he will be asked for the translation the first time. If the answer is correct the timer of the object will be set and the user has to wait till the timer has run down to zero. This can take from five minutes up to a few hours depending on the values in the *Vocable* class. If the answer is not correct the translation is shown another time, and after a very short time the user has to give the translation again. This process of querying and waiting repeats till the user has translated the new word sufficient times. Then the object is built and will be displayed in the house of the player with its normal image.

For building an object the user has to buy it on the market. For this he has to click on the market symbol of the game interface. The market opens and shows a list of objects available in the current area. If an object is available it is painted in normal color, otherwise it is painted in gray. Objects are available if all the objects the user has to build before are built and the user has the required level. The market also shows the prices of objects. Object can be bought with gold or with Wisdom Stones. A few special items are only buyable with Wisdom Stones. If the user can afford the item he can place it somewhere in the visual representation of the area. After placing an object the program shows the translation of the word for a few seconds. Then the user has to wait another few seconds, without seeing the translation. After that the first query of the vocable appears. The user has to input the correct translation of the name of the object. When he enters the right answer he gets a few experience points for it and an internal timer is set. The timer says when the user can query the vocable for the next time. With every correct query the progress property of the object increases till the object is completely built. For every wrong answer of a query the progress will be decreased but never less than zero.

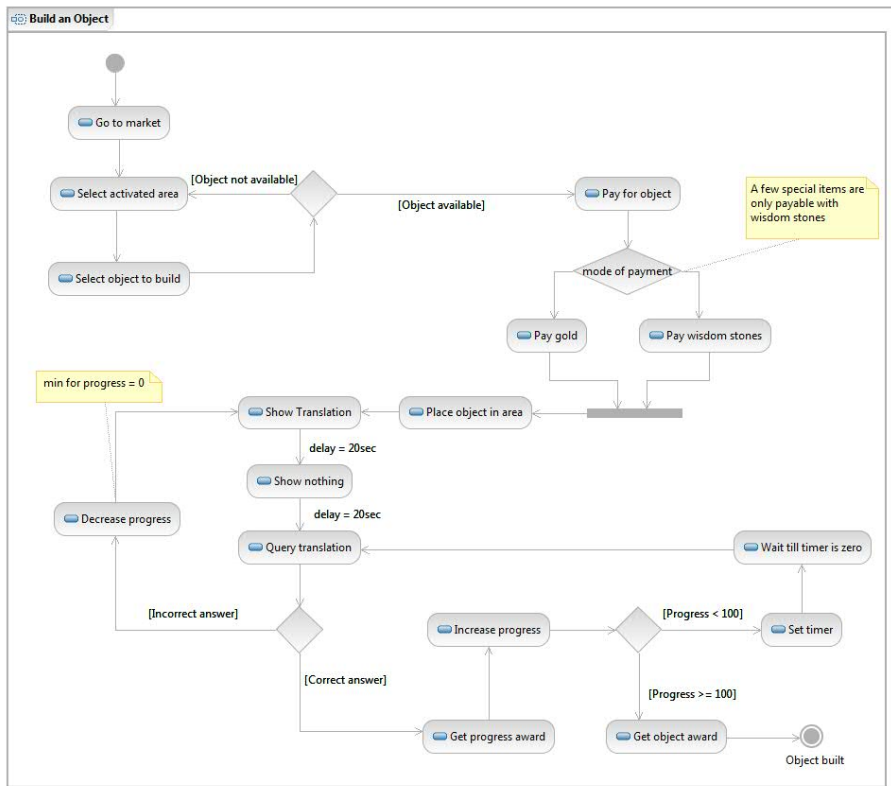


Figure 5: Activity diagram: Build an Object

11.2 Show Object/Timer

For showing the vocable or the timer of an object the user has to hover the mouse pointer over it. If the object is already built the application shows the name of the object in the language the user is learning. Also it shows how many times the user has translated this vocable correct and how many times not. If the object is in construction at the moment the application shows how long the user has to wait till he can translate it another time. In this case the translation is not shown but the progress of the construction process is displayed.

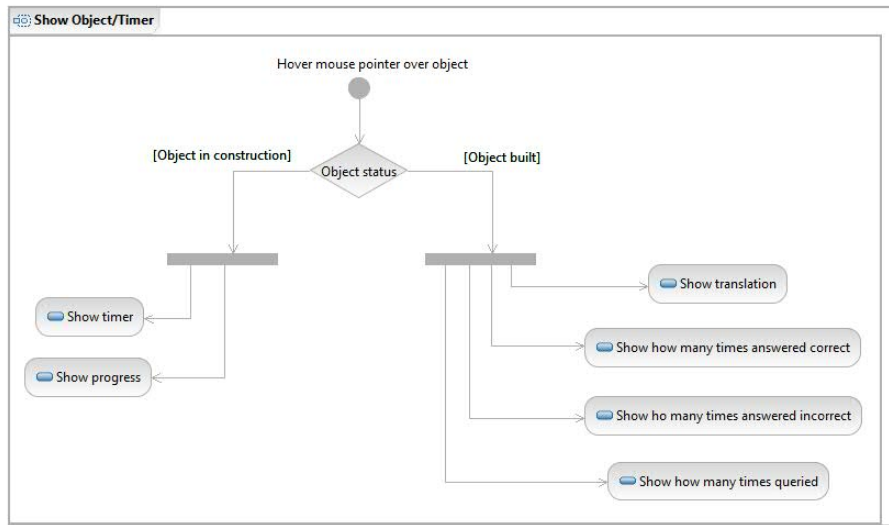


Figure 6: Activity diagram: Show Object/Timer

11.3 Activate Area

The order in which objects can be built is an important issue for the learning success of the user. The vocabularies are split into areas which represent rooms, thematic topics or other groups of objects in the game. The vocables are divided into areas because learning words that are semantically related to each other leads to better learning results. Before a user can activate an area he has to build all required areas which are defined in the *requiredAreas* property of the corresponding *AreaInProgress* class.

The user can select an area, which he wishes to active in the market. If the area is available the user can decide if he wants to pay for the activation with gold or with Wisdom Stones. After the payment process the application generates a list of vocabularies containing vocables from the required Areas of the recent area. A randomly chosen vocable of this temporary vocabulary is queried by the application. If the user gives the correct translation for a vocable it will be deleted from the temporary vocabulary. A wrong answer increases an internal error counter. If the counter reached a defined limit the activation process is canceled by the application. Otherwise the vocables of the temporary vocabulary will be queried till the list is empty. After translating all the vocabularies right the user gets an award of experience points and gold. The area is set active and the successive areas states are changed. All the states of vocables contained in the area are set to visible or available. (The different states of vocables and areas will be discussed in section 12.3)

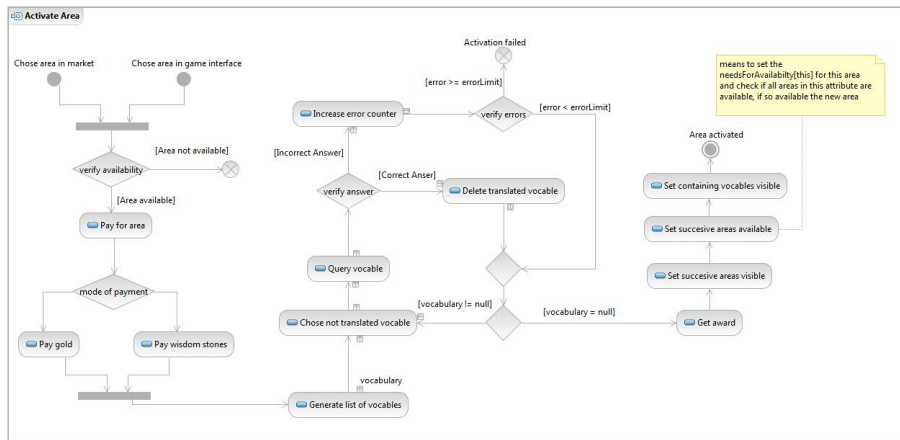


Figure 7: Activity diagram: Activate Area

11.4 Invite Neighbor

For inviting a new neighbor the user has to click on the “invite friends” button in the game interface. A list of his Facebook friends is shown where he can select the one (or more) friend(s) he wants to invite. After clicking on the “invite” button the new neighbor gets a message via Facebook which requests if the friend wants to be a neighbor of the requesting person. If the friend accepts the invitation the user gets a message explaining this. From now on the user sees the other player as a neighbor in his game.

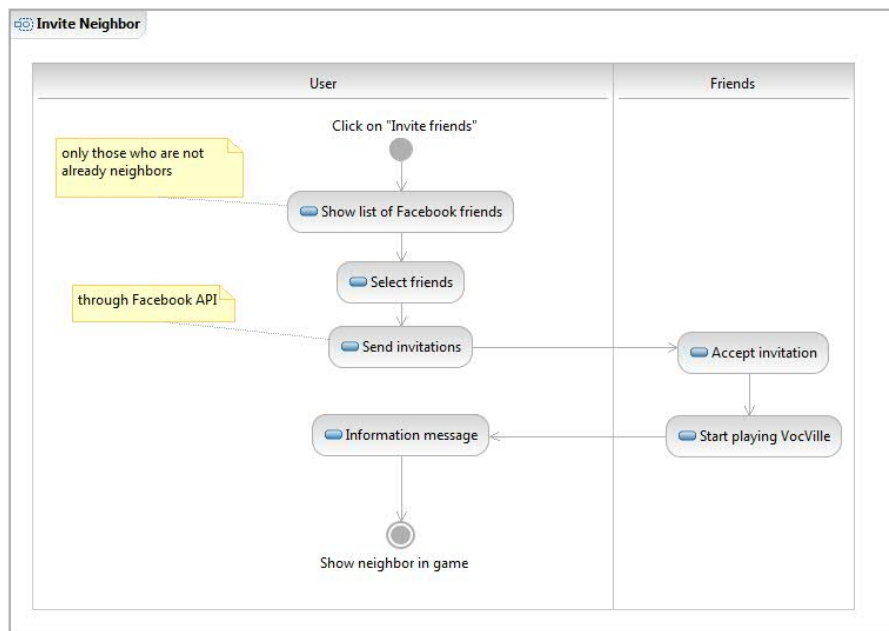


Figure 8: Activity diagram: Invite Neighbor

11.5 Visit Neighbor

For visiting a neighbor the user has to move the mouse pointer over the picture from the neighbor in the neighbor bar at the bottom of the game interface and click on “Visit ...”. Now the house of the neighbor will be loaded. After that the user can see the neighbor’s house like his own. He can hover over objects to see their translations and the statistic of them. He can also enter rooms and other areas. When the visited user is constructing any objects the visitor can help him by clicking on the object. This will have the effect, that the owner of the object will get a little aid at the next query of the vocable. The user who is helping gets directly a small amount of gold.

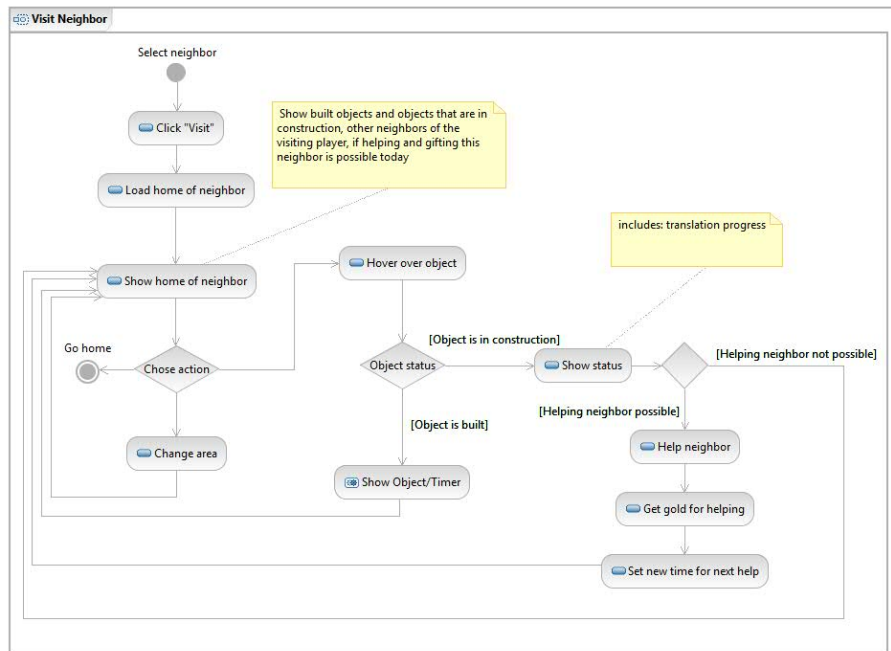


Figure 9: Activity diagram: Visit Neighbor

11.6 Send Gift to Neighbor

Users can send each neighbor one gift per day. The gifts are new designs for objects the neighbor owns or new vocables which the neighbor can build. Depending on the own level users can send better gifts. When the user has chosen a gift a message is sent to the neighbor. If the neighbor opens the message he sees what the present is. But for opening it he first has to solve the question that is shown. He can choose between three translations to put one of them into the free space from the question. If the answer is correct the neighbor can use the gift directly otherwise the gift can not be used at all.

There are three ways to initiate the gifting process. Users can chose neighbors in the neighbor bar which is displayed at the in bottom of the game interface, select a separate gifts button which allows sending gifts to more than one neighbor, or send a gift while visiting a neighbor. After one or more neighbors are selected a list of gifts is shown. Depending on the users level available gifts are listed. If the user has initiated the gifting process through the “Send gifts”-button a list of neighbors, which did not get a gift from the user within the last 24 hours, is shown and the user can select the neighbors he wants to send a gift. The final step on the users’ side of the process is that a message is send to the neighbors using the Facebook API. When receiving this message the neighbor can accept or ignore it. If he accepts the gift a randomly vocable from the list of his completed vocabulary will be queried. To use the item within the gift the neighbor has to translate this vocable correctly otherwise the gift will be deleted. Gifts have an expiration property which deactivates them over a certain amount of time after being sent.

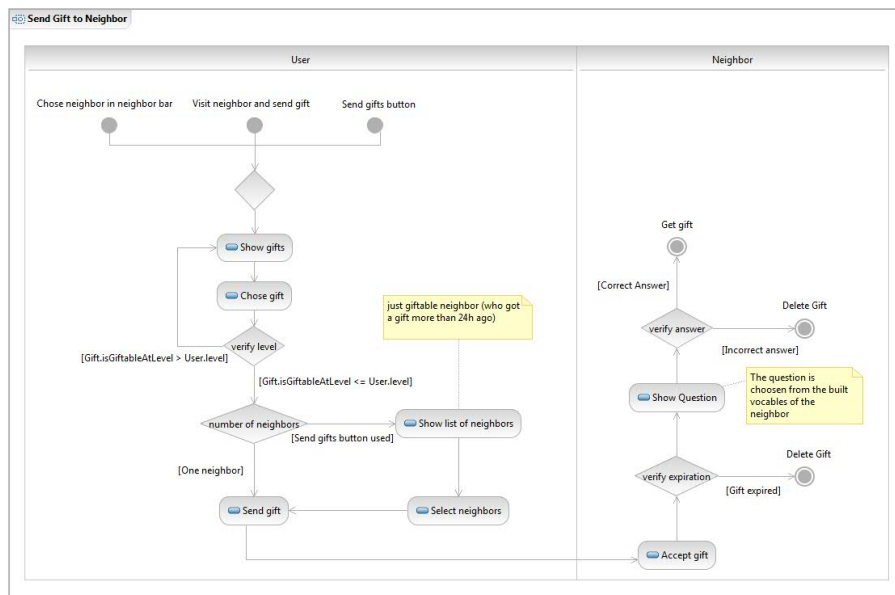


Figure 10: Activity diagram: Send Gift to Neighbor

Part IV

Implementation

As mentioned in the last chapter (10.2) the application is divided into two separate parts. The first part is the Grails application which stores all the game data in a database and offers a website for creating and managing templates for game objects like vocables and areas. The second part is the Flex application which is the actual game which uses the templates to generate a game. For the communication between the two applications the Grails side provides remote services in the Action Message Format. There are two remote services implemented. The *UserService* provides methods to receive or update data about the player. The *VocableService* provides methods to receive areas and vocable objects as well as updates about user actions like building objects or changing designs. The Flex side calls these services to receive the game status of a player and sends updates whenever a user action needs to be saved in the database. The separation is also reflected in the implementation of VocVille. The Grails part uses Groovy as programming language and GSP for the administrative website. The Flex part uses MXML for displaying the game interface and ActionScript for implementing the program logic and calling the remote services.

12 Data model

Also the data model is divided into two separate parts. The main data model describes the domain classes of the Grails application which are stored in the database. This includes the templates for the game objects as well as instances of these templates for every player. The second data model represents the Data Transfer Objects which the remote services of the Grails application generate from the main domain model. The DTO objects are sent to the Flex application.

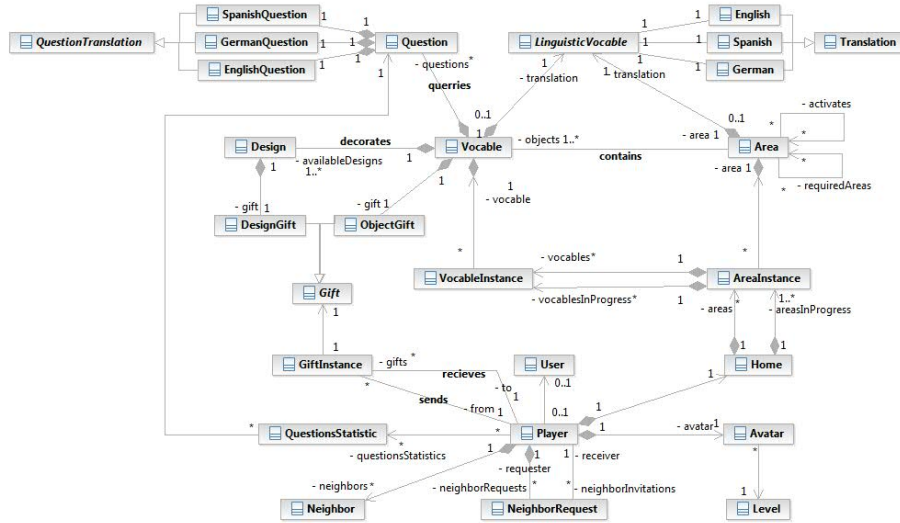


Figure 11: Main data model

The main class of the data model is the *Vocabable* class. It represents a game object and the vocabable of this object. It has associations to instances of the *Design*, *Question*, *LinguisticVocabable*, and *Area* classes. The *Design* class holds other visual representations for the object. The *Question* class holds exercises querying the vocabable which are used when a player wants to use a gift from another player. The *LinguisticVocabable* class is a proxy class which holds translations in different languages for the vocabable and offers them through the *getTranslation-method*. The *Area* class describes a collection of vocabables which are used in the game as thematic groups.

The *Vocabable* and the *Area* class both have instance classes (*VocabableInstance* and *AreaInstance*) which are related to them. The instance classes avoid storing redundant information in the database. They represent a unique instance of the corresponding object for a single player. Instance classes just describe the properties of an object which can be in different states for every player. The properties that are equal for all objects are just stored in the *Vocabable/Area* class and can be accessed through the corresponding association of the instance class.

Another central class of the data model is the *Player* class. From this class it is possible to navigate to all classes which store the current game state of a player. It has association to the *Neighbor*, *Avatar*, and *Home* classes. The *Neighbor* class represents other players of VocVille which have agreed to be a neighbor of the player. The *Avatar* class contains game relevant information about the player like his level and his amount of gold. The *Home* class represents the actual game status of the player by storing the areas which include the vocabables and the actual status for both of them. Furthermore it holds the gifts a player has received.

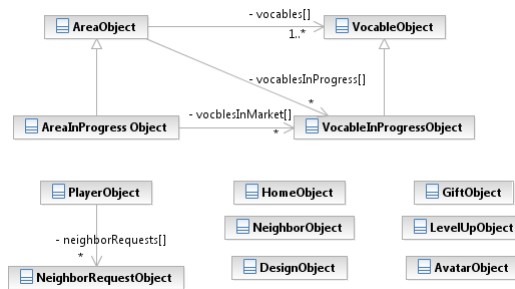


Figure 12: DTO data model

The DTO model describes the objects which are provided by the remote services and used by the Flex application to create the game. When the Flex application needs a certain objects it calls a remote service. The service then gathers the required data and packages them into a DTO class. The DTO classes are defined in the `src\java\dto` directory of the Grails project. The *Vocable* and *Area* objects both have two DTOs. Each has a superclass which holds just the information needed when the object is built/activated. The specialized subclasses, *AreaInProgressObject* and *VocableInProgressObject*, contain additional information which is necessary during the building/activation process. By transferring just the needed data the size of the requests can be reduced.

12.1 Vocables

Vocables are the main object in the application. A vocable (in VocVille) is an object which can be built by the player and therefore placed in the game interface. Vocables have translations in different languages and graphical representations depend on their status. Every vocable belongs to one area and can only be bought and build when this area is activated. Within the areas there is a certain order in which vocables can be built.

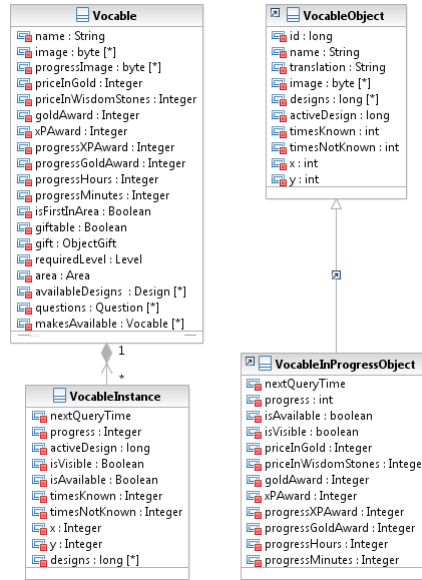


Figure 13: Vocable classes

The main data model stores information related to vocables in the *Vocable* and the *VocableInstance* classes. The *Vocable* class represents a template for vocables. The *name* attribute is a String which stores the English translation of the vocable used by the administrative interface as a human readable identification. The *image* attribute contains an image of the object which is painted on the game interface, if the object is built. The *imageInProgress* attribute contains an image of the object which is used when the vocable is in construction. All images in VocVille are stored as byte arrays within the database. It would be possible to store the images as files on the server and just store the url pointing at the image within the database, but storing them in this manner would mean that the database and the file system has to be synchronized every time a change is made. The *makesAvailable* attribute contains references to other vocables. If the building process of a vocable is completed, that is the vocable has been translated sufficient times by the player, the vocables in the *makesAvailable* attribute will be activated. The process of activating vocables will be described in more detail in the section 12.3. The *isFirstInArea* attribute is also used for the activating process. For buying and building a vocable a player needs a minimum level which is stored in the *requiredLevel* attribute. The price for buying a vocable in the market is stored in the *priceInGold* and *priceInWisdomStone* attributes. If either of them is zero the vocable cannot be bought with the corresponding currency. For the building process there are several attributes in the *Vocable* class. *ProgressHour* and *progressMinute* define the time period between two queries. *ProgressGoldAward* and *progressXPAward* define how much

gold and experience points the player gains for a single, correct answered query. *GoldAward* and *xPAward* define the awards the player gains when he completes the building process of the vocable. The *questions* attribute is used for the gifting process which will be described in section 12.8. The *availableDesigns* attribute points to different graphical representations for the vocable.

The second class in the main data model is the *VocableInstance* class. For every player every Vocable has a unique instance of this class in the database. It represents the game object of this vocable and the status of the building process for the vocable. To enable the remote services to generate a DTO of the corresponding vocable a reference to the Vocable class is stored in the *vocable* attribute. The *isVisible* and *isAvailable* attributes are needed for the activation process and therefore will be described in section 12.3. Depending on the definition in the *Vocable* class the user has to wait a certain amount of time between two queries. The time when the next query is available is stored in the *nextQueryTime* attribute. The *progress* attribute which is initially zero is increased at every correct answered query for this vocable. If the progress reaches 100 the building process is completed and the game object is built. The class also stores statistical information in the *timesKnown* and *timesNotKnown* attributes, which can be displayed to the user by the game interface. They count how many times the user translated the vocable correct and how many times not. A player can only change the graphical representation of a vocable with design he owns. Designs owned by a player are stored in the *designs* attribute. When a design for a vocable is changed this information is saved in the *activeDesign* attribute which references the recent design of the game object. Players can change the position of game objects in the area they are belonging to. Therefore *x* and *y* attributes store the coordinates of the game objects.

The DTO data model stores information related to vocables in the *VocableObject* and the *VocableInProgressObject* class. They are dynamically created by the *VocableService* which therefore uses combined information from the *VocableInstance* and *Vocable* classes of the main data model. The *VocableInProgressObject* class is a subclass of the *VocableObject* class. *VocableObjects* represent vocables that have been built by the player and hence do not need information related to the building process. The *image* attribute holds the recent graphical representation for the game object. Depending on the vocable status or the chosen design the *VocableService* populates this attribute with the *progressImage* or *image* attribute of the *Vocable* class or the image stored in the *Design* class. The *translation* attribute is a string representation of the translation depending on the language the player has chosen to learn.

12.2 Areas

Vocables are organized in areas. An area is a group of vocables which are semantically connected. It can represent a room like the living room but also a group of vocables describing an abstract set of things like clothes or electronic devices. Like every vocable within its area also every area is part of an order in which areas can be built.

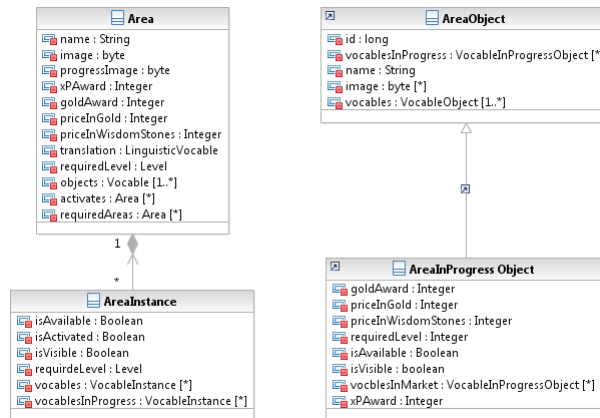


Figure 14: Area classes

The main data model stores information related to areas in the *Area* and the *AreaInstance* class. Like the *Vocable* class represents a template for vocables the *Area* class represents a template for areas. Since a lot of the attributes of the *Area* class have the same identifiers and semantically function as the ones for the *Vocable* class only the different and additional attributes will be described here. The vocables of an area are stored in the *objects* attribute. The activation process of an area includes a query of all vocables of the area (or areas) which are before the recent area in the areas order. To gather this list of vocables the *requiredAreas* attribute references to the previous areas. The attribute is automatically populated when an instance of the *Area* class is inserted or updated in the database. In these cases Grails adds a reference of the recent area to the *requiredAreas* attribute of each area that the recent area activates, that is the areas in the *activatesAreas* attribute (of the recent area). Because the area class is only queried one time it does not need progress awards and therefore only has attributes for the award gained when the query is completed.

The *AreaInstance* class is the equivalent of the *VocableInstance* class but for areas. It also represents an instance of an area related to one unique player and has a reference to its corresponding template in the *Area* class. *AreaInstances* store a unique instance of the *VocableInstance* class for every vocable that the corresponding area contains. The *VocableInstances* are stored in the two attributes *vocables* and *vocablesInProgress*. The first one holds all vocables that have been completely built. The second one holds all vocables that are in the market or in the building process. Because of the activation process instead of the building process of vocables, an *AreaInstance* does not need that much attributes for the recent state information. Just the two attributes *isVisible* and *isAvailable* are necessary. Their meaning will be described in the next section.

The DTO data model stores information related to areas in the *AreaOb-*

ject and *AreaInProgressObject* classes. Corresponding to the DTO classes for vocables they are dynamically created by the *VocableService* and combine information from the *Area* and the *AreaInstance* classes. *AreaObjects* represent areas that have been completely built, that is all their vocables have been built. *AreaInProgressObjects* represent areas that have not been activated yet or not all of their vocables have been built. The vocables of an area are stored in the three attributes *vocables*, *vocablesInProgress*, and *vocablesInMarket*. The *vocables* attribute stores completely built vocables as instances of the *VocableObject* class. The *vocablesInProgress* attribute stores vocables that are recently in the building process as instances of the *VocableInProgressObject* class. These two attributes are defined in the *AreaObject* class. Since this class represents completed areas they do not have any vocables in progress. So it would be logical to define the *vocablesInProgress* attribute in the *AreaInProgress* subclass. The reason for defining them here is that the method of the *VocableService*, which collects areas needed when a player visits a neighbor, needs to collect completed areas as well as the ones that are recently in the building process. These areas have to be included so the player can access areas with vocables in the building process which he can perform the help action on. Since in the neighbor state neither information about vocables that are in the market nor information about the state of the area (because it has already been activated) is needed they do not have to be included in the response for the service method. This has the consequence that every time the client calls for completed areas the *AreaObjects* of the response have an empty *vocablesInProgress* field, but on the other hand the response for requesting neighbor areas does not have all additional data of the *AreaInProgress* class which are not needed for that request. The *vocablesInMarket* attribute which is only defined in the *AreaInProgressObject* subclass holds all the vocables of an area which neither are completed nor in the building process.

12.3 States of Vocables and Areas

As mentioned before vocables and areas have an order which defines when they can be built or activated. This order is assigned when the objects are created in the administrative application of VocVille. For each object the user can decide which objects can be built after the building or activating process of it is completed. In this manner the user can connect each object with each other and create the building/activation order. To realize this order in the game every object has four different states which are shown for vocable objects in Fig.15. In the first state, called “visible”, the objects can be seen in the market but cannot be bought or activated. In the “available” state the objects are in the market and can be purchased. The “progress” state differs between vocables and areas. Vocables in the progress state are in the building process. Areas in the progress state have been activated but not all of their vocable objects have been completely built. The last state is the “built” state in which the objects are created which means for vocables that they have completed the building process and for areas that all of their objects are in the built state. The actual state

of an object is coded in the two boolean attributes *isVisible* and *isAvailable* of the *VocableInstance* and the *AreaInstance* classes. The meaning of the possible combinations of values for this attributes are explained in table 3.



Figure 15: States of vocable objects

isVisible	isAvailable	State
true	false	visible
true	true	available
false	false	progress
false	true	built

Table 3: Codification of states

The state changes for vocables and areas are not exactly the same and therefore will be described separately. Vocable states are changed when a vocable object completes the building process. Vocables are created and set to the visible state when the area they are belonging to is being activated. After the activation of an area all vocables of it are created and generally set to the visible state. Vocables that have the *isFirstInArea* attribute set to true are set to the second state, namely “available”. Without this initial vocables in an area, no vocable could be built and hence no other vocable could be set to the available state. When a vocable object is queried sufficient times so the *progress* attribute reaches 100 the vocable changes to the built state. All vocables referenced in the *makesAvailable* attribute of the *Vocable* class for this vocable are set to the available state. When the vocables are bought in the market they are set to the progress state and the building process for the vocable starts. At the end of the building process the cycle starts again for the next vocables.

When a new player starts the game for the first time the first area is set to the progress state. All referenced areas in the *activates* attribute of the corresponding *Area* class are created and set to the visible state. After this initial creation area states are changed whenever the player activates an area or an area is set to the built state. This happens when the last vocable of an area completes the building process. At this point all areas which are referenced in the *activates* attribute are being checked if they can change to the available state. An area can change to the available state when all of its previous areas,

stored in the *requiredAreas* attribute, are in the built state. All areas that can be set to the available state then create their following areas and set them to the visible state. Because an area can be activated by more than one area it is possible that referenced areas are already created. In this case no new area is created.

Fig. 16 demonstrates an example of the area state changing. First the initial *AreaInstance* “living room” is created and set to the progress state (1). In the *Area* class for the living room the *activates* attribute references to the “kitchen” and the “bedroom” areas. Therefore *AreaInstances* for them are created and set to the visible state. When the living room is completed the following area of kitchen, the “electronics” area, is being created (2). Now the following areas of the bedroom area are being created, but because the electronics area already exists just the clothes area is created. After that the player activates the bedroom area, which just changes the state of this area (3). When the player has completed all vocables of the bedroom, the areas in the *activate* attribute are being checked (4). Because the electronics area needs both, the kitchen and the bedroom, to be completed just the clothes area is set to the available state. Now the player activates the kitchen area (5) and completes it (6). The electronics area is checked again, because it is referenced in the *activates* attribute of the kitchen area, and is now set to available because all needed areas are in the built state.

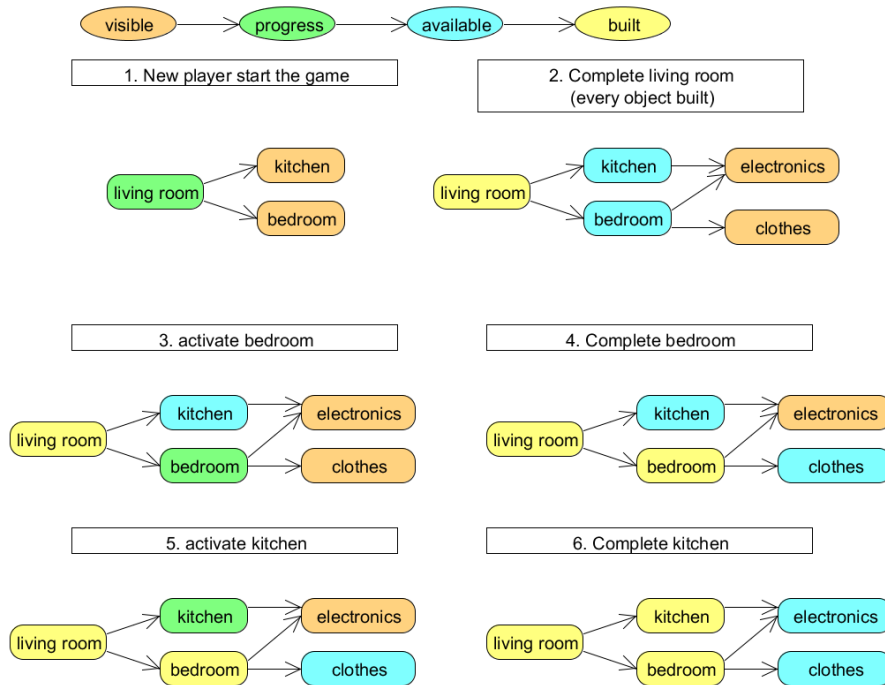


Figure 16: Area states example

12.4 Translations

Translations of vocable or area names are stored in instances of the *LinguisticVocable* class. This class implements the facade design pattern to simplify the access at the translations for different languages.

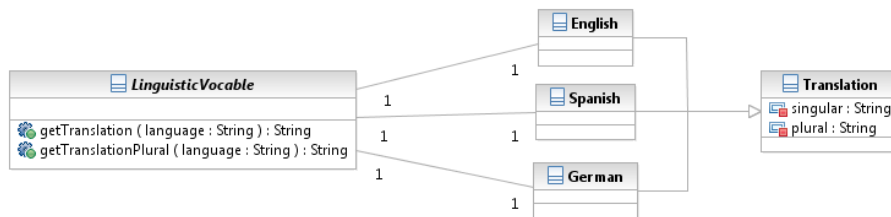


Figure 17: Translations classes

The languages are implemented as subclasses of the *Translation* class. Every translation has two strings representing the singular and the plural form of

the translation. The *LinguisticVocable* class has an attribute for each of the three implemented languages. To provide the translations a *linguisticVocable* offers the *getTranslation*- and *getTranslationPlural*- methods which take a string as a parameter. The string parameter defines the desired language for the translation in a language code. The codes are two letters in uppercase. “EN” stands for English, “ES” for Spanish (Español), and “DE” for German (Deutsch). Which language a player desires is stored in the instance of the *Player* class. When the *VocableService* generates the DTO for vocables or areas it calls the *getTranslation**-method to get the appropriate translation. In this manner it is very easy to implement more languages simply by creating a new subclass of the *Translation* class, adding an attribute to the *LinguisticVocable* class and modifying the *getTranslation**-methods accordingly.

12.5 Designs

Designs represent alternative graphical representations for vocable objects. To make the game more customizable the player can change the looks of his objects.

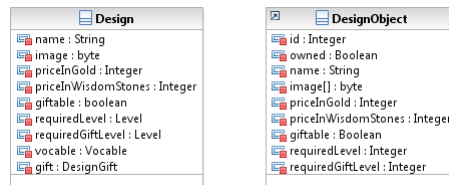


Figure 18: Design classes

In the main data model designs are defined in the *Design* class. The *name* attribute is displayed in the market and the interface component for changing the design of an object. The *image* attribute obviously represents the picture of the design with which the object will be displayed when using this design. Designs can either be bought in the market or sent as a gift to a neighbor. Which of the two methods is applicable is stored in the Boolean attribute *giftable*. The price for a design is stored in the *price** attributes. If one of them is zero this means that the design cannot be bought with this currency. Like other objects in the game the player has to reach a certain level to be able to buy a design. This level is saved in the *requiredLevel* attribute. To gift a neighbor with a design, the level of the player who wants to send the gift has to be at least the same level as referenced in the *requiredGiftLevel*. The DTO for designs, the *DesignObject* class, completely reflects all attributes of the *Design* class and adds just the Boolean *own*. When the player wants to change the design of an object the game displays all designs the user owns for this object and additionally designs he can buy at the market. To let the Flex application differentiate between owned designs and buyable ones, the *VocableService* sets the *owend* attribute accordingly.

12.6 Player

The *Player* class in the main data model is the main entry point for navigating through the model. From a *Player* instance one can access all data entries related to the recent player. Most of the *VocableService* methods have the id of an *Player* instance as a parameter to enter the main data model.

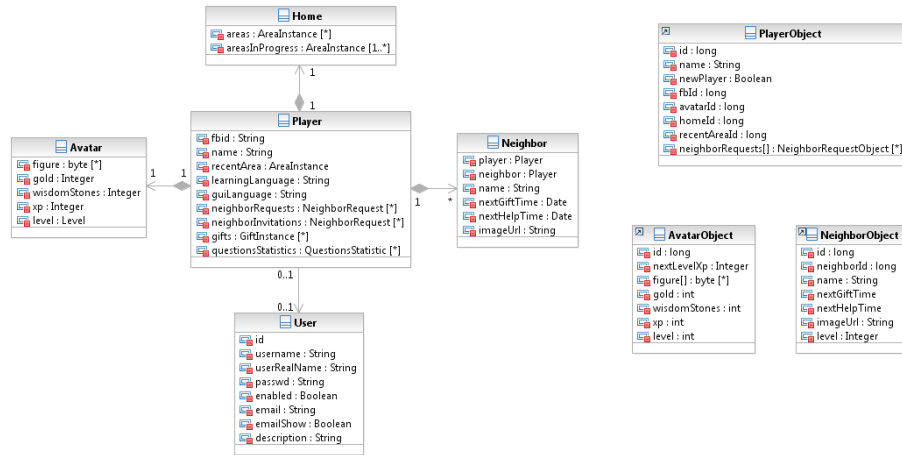


Figure 19: Player classes

The *Player* class stores the Facebook id of the player in a string attribute called *fbid*. The learning language and the desired language of the GUI are stored in the *learningLanguage* and the *guiLanguage* attribute. The format of their values is defined by the *LinguisticVocable* class to differ between the three implemented languages. The player class has a collection of instances of the *Neighbor* class. For every neighbor the id of the corresponding player and his level is saved. If the neighbor also has a Facebook account his name and the URL to the profile picture are also stored in the *Neighbor* class. A player can help and send a gift to each off his neighbor once every 24 hours. At what time the next action is available is stored in the *nextGiftTime* and the *nextHelpTime* attributes.

The *Avatar* class holds information about the achievements of the player like his amount of the different currencies, his level and his experience points. It also includes an image of the figure that is shown in the game interface. The home class holds references to all *AreaInstances* of the player. That is all the areas and the including vocables the player recently has created, is creating or can purchase in the market. It also holds references to the gifts the user has received and not yet opened.

The *User* class represents a user of the administrative application of VocVille and therefore does not exist for a regular player of the game. It holds the properties needed by the ACGI-security plugin for the Grails application.

12.7 Neighbor Requests

NeighborRequest are created when a player invites another player to be his neighbor. The *NeighborRequest* class in the main data model holds references to the player who send the invitation and the payer who receives the invitation. When the invited player accepts the invitation the request object is deleted and for both players a new instance of the Neighbor class are created and added to their instance of the Player class. The *NeighborRequestObject* class of the DTO data model represents a player which is not a neighbor of the recent player. This includes player for which no request related to the recent player exists. The Boolean attributes *requested* and *invited* are initially set false and just set to true if a requests for the certain player exists in the main data model.

12.8 Gifts

Gifts represent game objects that can be sent from one player to another. When a player wants to use the object in the received gift he/she has to answer a question. The questions are cloze tests which ask for a vocable the player has already built.

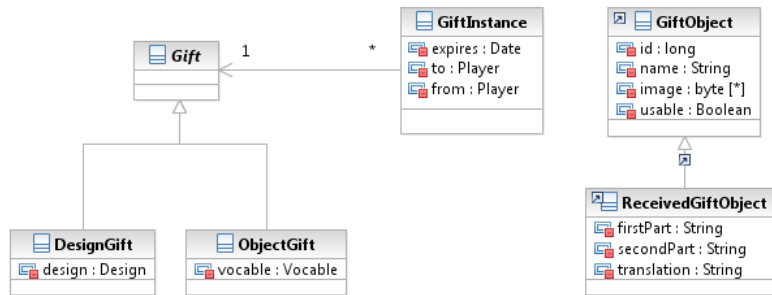


Figure 20: Gift classes

In the main data model the templates for gifts are represented by the *DesignGift* and the *ObjectGift* domain classes which are both subclasses of the *Gift* class. Design gifts contain a design, Object gifts contain a vocable, which the receiving player will be able to use. When a player sends a gift to another player a new *GiftInstance* object will be added to the *gift* attribute of the second players entry in the player database table. The *GiftInstance* class holds information about the received gift, the time when the gift will expire and therefore will not be usable anymore as well as references to both players. When a player opens a gift the application choses randomly a question from all the vocables the player has built yet. Therefore each vocable template references to questions that belong to it. The questions are represented by the *Questions* domain class which holds translation for each language. Similar to the *Translation* class this class

offers a *getTranslation*-method to request the corresponding translation for each player. The translations are subclasses of the *QuestionTranslation* class which holds two string attributes representing the textual parts of the question before and behind the demanded vocable. Additionally it holds the Boolean attribute *singular* which says if the vocable translation should be in singular or plural.

In the DTO data model the *GiftObject* and *ReceivedGiftObject* classes represent gifts. The first one stores information for all available gifts and if the recent player can use them. *GiftObjects* are sent when the game interface has to display all available gifts. The second class, which is a subclass from *GiftObject*, represents gifts, which have been sent to a player. Therefore they store additional information about the question which is queried when the gift is opened.

13 Administrative Interface

The administrative interface is a website which is created by the Grails application of VocVille and will not be accessible on the Facebook platform. It has user accounts separated from the user accounts for the Flex application which represent the players. The administrative users are stored as instances of the *User* domain class. Since the administrative interface is not accessible to everyone the access is restricted by passwords which are also stored in the *User* class.

The administrative interface allows access to all domain classes which represent the templates for game objects but not to data belonging to actual game states of players. For example *Vocables* for a certain game object can be created or manipulated but not *VocableInstances* for a certain player. This part of the VocVille application is designed for teachers who want to add vocables, designers who want to change graphics of game instances and for game developers who want to reorder the areas or create a complete new instance of a game. The next section explains the concept of game instances and the following describes how a user can manage them in the administrative interface.

13.1 Game Instances

Because of the design of VocVille which generates the game objects like vocables and areas depending on the data stored in the database it is very easy to create different instances of the game. Depending on the order of the vocables and areas each game instance allows the player to learn other vocables. It is even imaginable to create game instances which vocables represent other topics of learning which can be learned by repeated queries. In this manner it would be easy to build a game instance in which vocables represent multiplication tables to learn the basics of multiplication. Another game instance could ask for the capital of countries. These game instances can be manipulated or created in the administrative interface. A *GameInstance* represents a set of game objects which are belonging together through their order. Every *GameInstance* has an initial area which is activated when a new player starts to play the game with

this particular `GameInstance`. The initial area links to the next areas which are activated when the first one is completed and these areas link to their following areas. In this manner the whole game for a `GameInstance` is build.

13.2 Game objects management

The main page of the administrative Interface shows a list of all `GameInstances` (Fig. 22). From here new ones can created or existing ones can be edited. As figure 23 shows, the dialog for creating a new `GameInstance` requests all needed information to build the instance and the initial area for this instance. When the user clicks on an existing *GameInstance* the `GameInstance` view opens (Fig. 24). It shows the properties of the instance and a list of all areas within this instance. Since the domain class for `GameInstances` just stores the first area all the other areas are collected dynamically by their order. For each area its previous and following areas and the image of the area are displayed. From this `GameInstance` view the user can navigate to views for editing the properties of the `GameInstance`, creating new Areas, editing the existing ones, or viewing a list of all existing gifts for the objects of this `GameInstance`. Figure 21 shows the site map for the administrative interface and how the different views can be accessed. Since the views are all have the same underlying layout I will just describe the additional elements of each view. In the area view (Fig. 25) beneath the properties of the area the vocable objects of it are displayed with their image and the list of vocables they activate. In the vocable view (Fig. 26) the default image and the image used in the building process are shown. Next to them all designs for the vocable are displayed. In the vocable and the area view the corresponding *translation* attributes are displayed with a template which creates a list to show the translations in all three languages. Whenever an object of one of the two classes is created or edited the corresponding *translation* attribute is handled accordingly. The design view (Fig. 27) shows the image of the design next to the properties. The vocable and the design view both check if the *giftable* attribute is changed. When this attribute is changed to true an

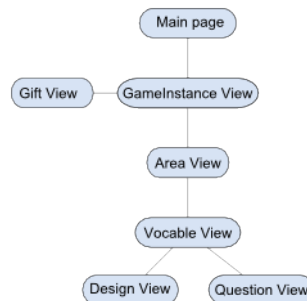


Figure 21: Sitemap for the Administrative Interface

instance of the *Gift* class for this object is created which means that players can send this object as a gift to a neighbor. When the *giftable* attribute is changed to false the corresponding gift object will be deleted from the database. The question view (Fig. 28) uses a similar template like the template for translation to display and edit the different translation of questions.

Id	Logo	Name	First Area	Owner	Query Language
1		Vocville	living room	koko	Falsch
2		Flags	Europe	koko	Falsch

Figure 22: Main page of Administrative Interface

GameInstance anlegen

Name:

Logo:

Owner:

Editors:

Query Language:

firstArea anlegen

Name:

Singular:

Plural:

Image:

Progress Image:

Figure 23: Creating a new GameInstance

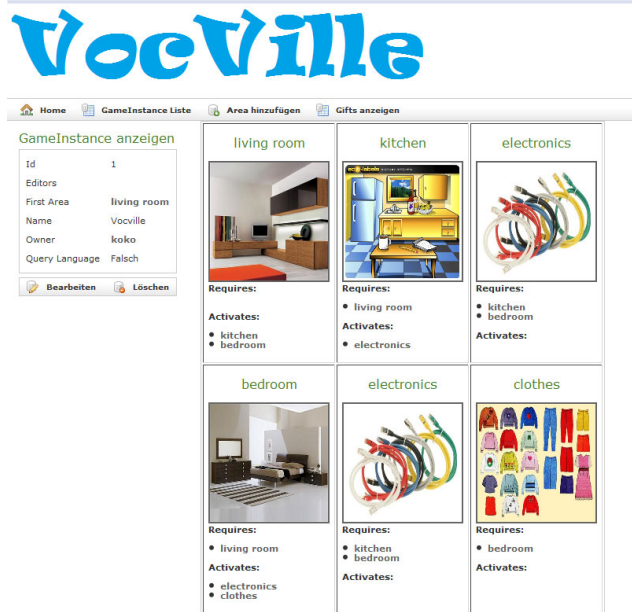


Figure 24: GameInstance View

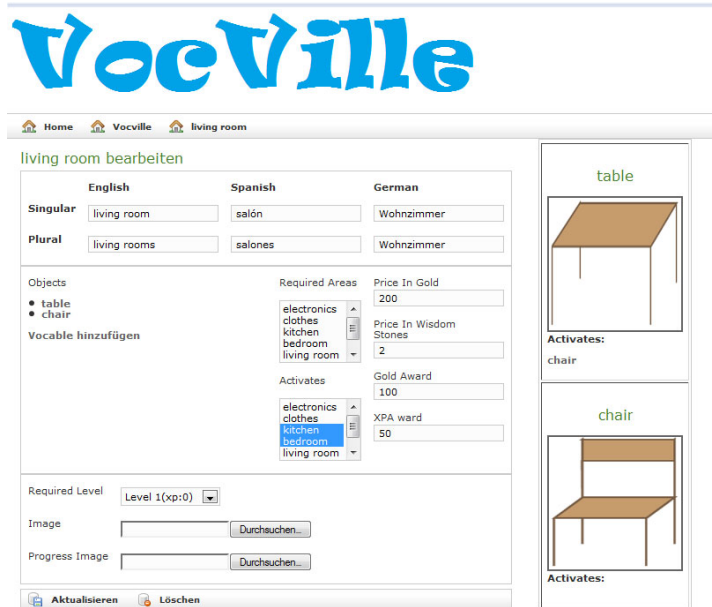


Figure 25: Area View

VocVille

Home Vocville living room chair

chair bearbeiten

	English	Spanish	German
Singular	chair	silla	Stuhl
Plural	chairs	sillas	Stühle

Area: living room Is First In Area Can be a gift

Price In Gold: 100 Price In Wisdom Stones: 1

Progress Hours: 0 Progress Minutes: 5

Progress Gold Award: 10 Progress XPA ward: 1

Gold Award: 100 XPA ward: 25

Required Level: Level 1(xp:0)

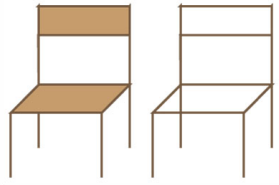
Makes Available: table chair Image: Durchsuchen

Progress Image: Durchsuchen

Available Designs: orange Questions: Question: 2, Question: 1

Design hinzufügen Question hinzufügen

Aktualisieren Löschen

Default Design: 


Available Designs: orange 

Figure 26: Vocable View

VocVille

Home Vocville living room chair orange

orange bearbeiten

Name: orange

Vocable: chair

Price In Gold: 10

Price In Wisdom Stones: 1

Required Level: Level 1(xp:0)

Required Gift Level: Level 1(xp:0)

Can be a gift:

Image:

Aktualisieren Löschen




Figure 27: Design View

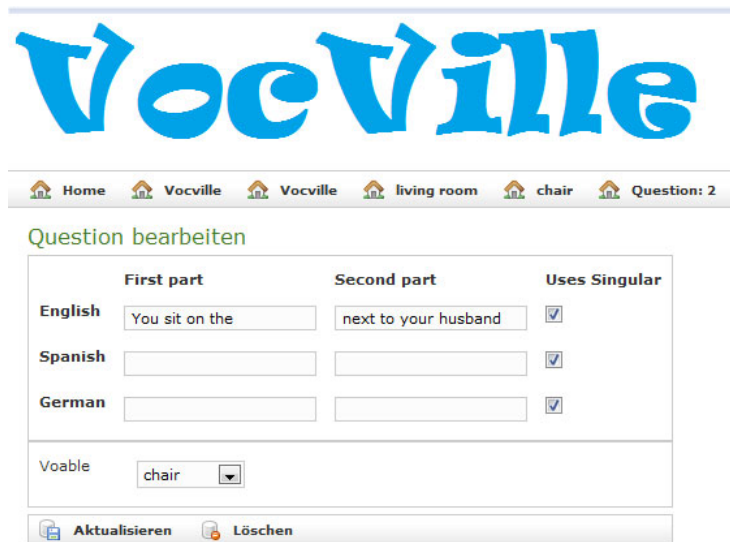


Figure 28: Question View

14 Game Interface

The interface of the game is a typical game interface and hence really simple and easy to understand for the player. On the top of the screen a status bar displays information about the game status and buttons to navigate between areas. The main part of the interface displays an area and its objects. At the bottom of the game interface the neighbors of the player are shown. The user can click on them to visit the homes of the neighbors or send them gifts. Next to the pictures of the neighbors there are buttons to access different features like the market. The main application (vocville.mxml) has two different states⁵. The “normal” state is activated when the player sees his own home. This state will be used most of the time. The second state, namely “neighbor”, is activated when the player visits the home of another player. In this state objects and the interface have a different behavior. In the next section I will describe the normal state and in the following section I will describe the neighbor state.

14.1 Home State

When the player starts the game the home state is activated. The status bar contains buttons for every area the player owns. Clicking on them changes the area which is displayed in the main part of the interface. Next to these area

⁵The term “state” is used in the Flex framework

buttons the gold, wisdom stones and experience point are displayed. Beneath the experience points a progress bar is indicating how much more points the user needs to reach the next level. The main interface displays the current area and the object the user has bought so far. If the user hovers with the mouse pointer over an object, the object will be highlighted by a colored frame. For completed objects their translation and a statistic about how many times the vocabulary has been asked and how many of these queries were answered correct or incorrect are displayed. If the player clicks on a completed object a popup is opened where the player can change the design of the object (see fig. 30). For objects that are in the building process a timer is shown, which indicates how long the player has to wait till the vocable of that object can be queried again. When a timer of a vocable is zero the object will be visually highlighted even when it is not hovered by the mouse. If the player clicks on an object which timer is zero a popup for the query process will be opened (see next subsection). Next to the neighbor bar at the bottom of the interface buttons for the market, overall statistic, neighbor requests and the move-tool are placed. Each of these features will be described in the next subsections.

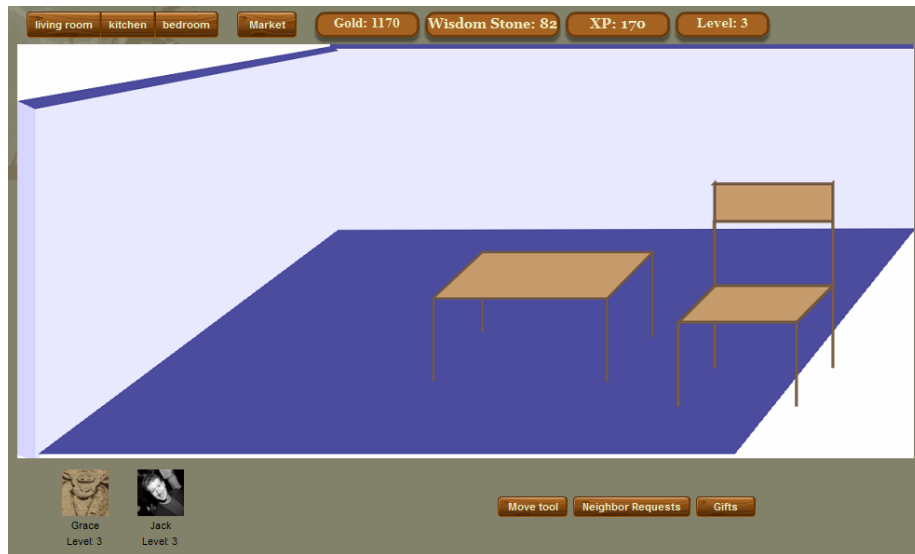


Figure 29: Home state



Figure 30: Change Design Popup

14.2 Query Process

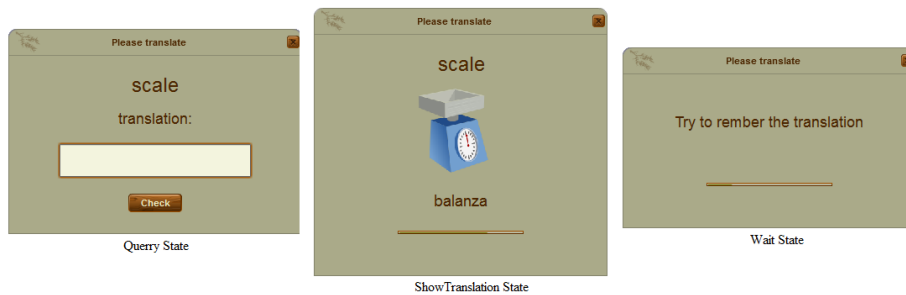


Figure 31: Vocable Query

When the player clicks on an object which timer is zero a new query popup is displayed. The query popup shows the name of the vocable (which is the translation of the vocable in the language of the player) and asks the player to enter the translation of it. If the user clicks on the check button the application compares the given answer with the translation of the vocable. If they are equal, which means the player has translated the vocable correct, the query popup closes, the timer of the vocable is reset and the progress of the vocable

is increased. The application also checks if the object now is completely built. If so the corresponding actions are taken (see 12.3). If the answer is not correct the progress of the vocable is increased and the translation of the vocable is shown for a few moments. After that a waiting screen is shown which asks the player to remember the translation for another few moments. Then the player is queried again. This mechanism repeats until the correct answer is given.

14.3 Marketplace

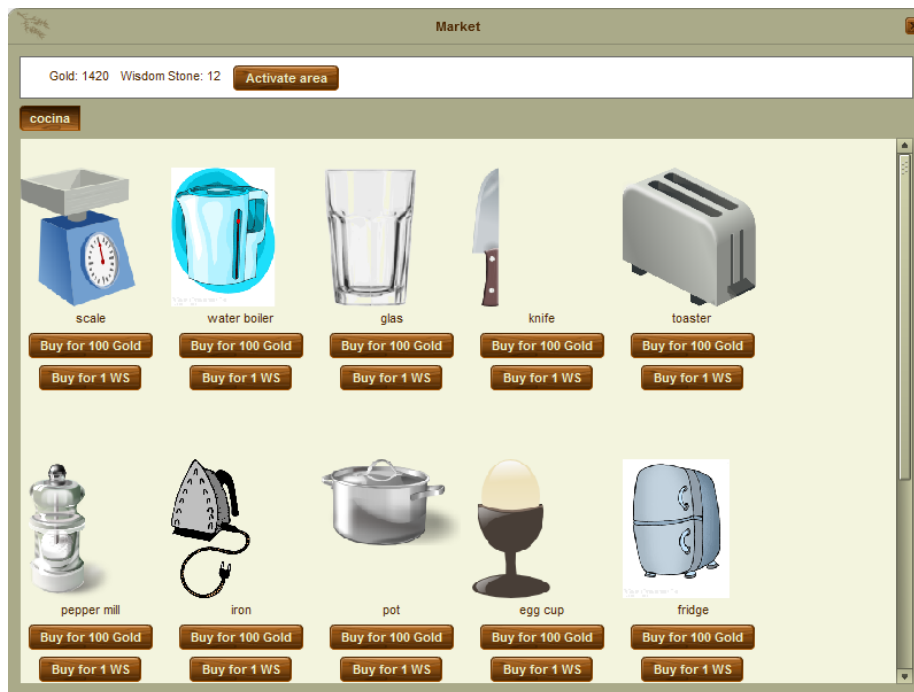


Figure 32: Market

At the top the market shows the recent amounts of gold and Wisdom Stones the player owns. Beneath this a button for activating new areas is placed. Under this a row with buttons for each area the player can buy vocables from is displayed. If the player clicks on one of these areas the vocable objects of this area are shown in the main interface of the market. Each object is displayed by its image (from the complete state) and its name. Depending on the building order of the vocables they are shown as available or not available. Available objects have additional two buttons under the name. With these buttons the player can buy the object using gold or Wisdom Stones. The buttons display the price in the corresponding currency. If an object can only be bought in one currency just one button is added. When an object is bought the market

popup closes and the new object is placed in the area. Not available objects are displayed with a grey layer over them and a text saying that this object is not available. To activate a new area the player has to click on the “activate area” button on the top of the market. Now all areas which are not activated yet for this player are shown in the main area of the market popup. The areas are also shown as available or not available. Available areas have the same buttons like available vocables and not available areas are displayed like not available vocables. When the player activates an object by clicking on a “activate area” button a list of all vocables of all areas which are directly before the area in the area order are displayed in a new popup. The player has to translate all the vocables correct to activate the new area. If one translation is not correct the area will not be activated. This means the player has to pay again for the next try. After the activation the new area is displayed in the market and the objects of it can be bought. The area is also display in the main application.

14.4 Move Tool

The move tool allows the user to change the position of objects in an area. If the player clicks on the Move-tool button it is displayed in a down state. Now the behavior of the objects in the main interface is changed which means clicking on them will not perform the normal actions like querying or changing the design. Instead when the player clicks on an object he can change its position by holding the mouse button down and dragging it to the desired place. Clicking another time on the move-tool button gives the objects their normal behavior back and displays the button in normal mode.

14.5 Neighbor Requests

The neighbor requests popup which appears when the player clicks on the corresponding button in the main interface allows the player to invite other players as neighbors or accept invitation from other players. The popup displays a list of other players and some information about them. This list contains all other players which are not already a neighbor of the current player. If no request between a player and the recent player exists an invite button is displayed for a player item. When the player clicks on it a request is sent to the player which asks him/her to be a neighbor of the player. If a request between the player and the recent player exists this is shown by a button which allows the player to accept an invitation which was send to him/her or to cancel an invitation which the player has sent to the other player. If a player accepts the invitation of another player both are added as neighbors for each other and hence can visit and help each other.

14.6 Gifts

The gifts popup allows the player sending gifts to his neighbors and opening gifts he has received from them. For each of these functions the popup has a



Figure 33: Gifts

separate tab. In the receiving tab all gifts the player has received are displayed with their image, the name of the player who sent the gift and the date when the gift will expire. Under each gift a button to open the gift is placed. When the player clicks on it a new popup appears. This popup shows the questions for the gift and the image of the queried vocable. To use the item within the gift the player has to type the correct translation of the demanded vocable. After clicking on the “check” button the given answer is checked. If the answer is correct the item of the gift is added to the home of the player. If the answer is not correct the player cannot gain the item. In both cases the gift will be deleted after the query.

14.7 Neighbor State

To visit a neighbor the player has to click on the neighbor item in the neighbor bar and choose “visit neighbor” in the appearing menu. This will activate the neighbor state of the main application. The neighbor state looks like the home state but has fewer options than it. It displays the areas of the chosen neighbor. The player can switch between them like in the home state but he cannot access the market, the moving tool or the requests popup. Also the behavior of the vocable objects is different in this state. Designs of completed vocable objects cannot be changed but their translation and statistics (for the neighbor) are displayed as tooltips. The player can help the neighbor for vocable objects that are in the building process. To do so the player has to click on an object. Then it is checked if the last time the player helped this neighbor is more than 24

hours ago. If so the timer of this object is reduced by a small amount and hence the neighbor can query it a little bit earlier.

15 Connection between Grails and Flex

As mentioned before the VocVille application is separated in two applications: the Grails application which maintains the main data model and offers management features through the administrative interface and the Flex application which is the original game the player uses. To transfer the data from one application to the other they have to have a connection both sides can implement. For this reason the Grails application uses the BlazeDS4 plugin which allows implementing a remote service which can be accessed by the Flex Framework. The plugin uses the BlazeDS web messaging technology which formerly was a part of the Adobe LiveCycle Data Services but since 2007 is a self-contained, open source technology. BlazeDS implements a call and response model for accessing external data over HTTP services. The service calls and objects are transferred in the binary Action Message Format. The Grails application of VocVille implements the two remote services *UserService.groovy* and *VocableService.groovy*. When the Flex application needs data for displaying the game it calls one of these two services. They generate Data Transfer Objects (DTO) by gathering the necessary data from the different domain classes in the data base and sending the response to the Flex application. When the player performs any action in the game, the Flex application sends information about it to the remote services. The services transform the received information and save it in the adequate location and format into the data base.

To describe the services I will now explain how the Flex application initiates and runs the game. In the process I will discuss the service methods when they are called by the game. The first and most important information the Flex application needs is the id of the current player. Since the game is either started directly on the Facebook platform or from the VocVille webpage, where the player also has to log in using his Facebook account, the Flex application knows the Facebook id of the player. It uses this id to call the *getPlayerByFbId*-method of the *UserService*. The method searches for a player with the given id. If no player is found a new player is created and the first area is activated. When the Flex application receives the player DTO it calls the services methods for receiving the avatar, the neighbors, the areas and the areas which are in the market. It also switches the language of the game interface depending on the preferred language of the player. In the Flex application areas are stored in three different arrays. The *areas* array holds all areas that are completely built and therefore just need basic information. For example they do not need information about their price since they already have been bought. This information is transferred in *AreaObject* DTOs. The *areasInMarket* and the *areasInProgress* arrays represent areas that need additional data and therefore are transferred in the bigger *AreaInProgress* DTO which include all information needed for the building process of an area and its vocable objects. The database holds

information about the last area that the player has viewed in his game. This information is stored in the player DTO and can be a completed area or an area which is recently in the building process. To display this area the Flex application has to wait till it has received both types of areas. Because the service calls to the Grails application are asynchronous it is not sure when exactly a call is responded and the transferred data can populate the designated variable in the Flex application. Therefore when the completed areas are received the Flex application sends an additional request to get the areas which are recently in the building process. In this manner the Flex application can be sure that both, the *areas* and the *areasInProgress* arrays, are populated when it receives the result from the second request. Now it can switch to the last viewed area of the player.

After the initialization of the game the player can perform several actions which result in remote service calls. When a vocable query is performed, the *sendQuery*-method is called. This method increases or decreases the progress value of the vocable depending on the correctness of the answer the player has given. If the progress is increased the method also checks if the vocable is completely built with the new value and if so takes the corresponding action. This includes checking if after the building of this vocable all vocables in the area are built and therefore the area is completed. Another remote service call is taken when the player changes the position of an object. The *moveObject*-method updates the database with the new coordinates of the moved object. When the player wants to activate an area the Flex application makes a service call to the *getRequiredVocables*-method, which collects all vocables which the player has to translate for activating the area.

16 Deployment

Thanks to the friendly people of daureos.com I could deploy the application on a real webserver. They set up the subdomain www.michel.daureos.com where I can run a Tomcat server. Therefore I deployed the Grails application as a war file and loaded it into the Tomcat. After that I copied the compiled Flex application in a subdirectory of Grails application within the server. I use MySQL as the database which the Grails application accesses. In Facebook I created an application with the name VocVille which points to my Tomcat server on the daureos page.

Part V

Evaluation

17 Analytic Evaluation

17.1 Requirements

In this section I will list all requirements defined in chapter 9 and evaluate how the application realizes them.

17.1.1 Non-functional

Available in Facebook The application is registered in Facebook as “VocVille” (Application id = 162991117070950). It is accessible through the Facebook platform as well as through the web site of the game (www.michel.daureos.com/vocville).

Easy to use The game interface is designed to be easy to understand and the player can start playing instantly. At the moment there are no tutorials which explain the features but they will be implemented in the near future.

Appears more like a game than a vocabulary trainer The player learns the vocables to build new objects in the game. From the point of view of the player he primarily creates an object and in the process has to learn the translation. Because the vocables are game objects which the user can perform several actions with, they do not appear like abstract learning items like they do in ordinary vocabulary trainers.

Nice graphics Since the graphical representations of the game objects are chosen by the users of the administrative interface this requirements depends on their choices. Nevertheless the possibility to choose a picture for the avatar helps the player to identify himself with it.

Playable in every browser The application has been tested in four of the most used browser (Firefox, Opera, Google Chrome, and Internet Explorer) in different Versions of Microsoft Windows and different Distributions of Linux. Since none of them had problems running the application it can be assumed that it will run without problems on most of the recently used systems. Since the administrative interface generates plain HTML pages it can be assumed that it will be usable in every environment that is able to display HTML pages.

17.1.2 Functional

Game Interface

Create player account When the player starts the game for the first time the Grails application creates a new player account with the Facebook id of the player after a popup in the game interface asks for his name, his languages and a picture of the avatar.

Display home of the user The Flex application gathers the required information to display the home of the user from the remote services of the Grails application. Every object is painted depending on its current state and on the location where the player has placed it.

Save current game state persistently Every action the player performs in the game initiates a remote call to the Grails application which saves the changes in the database immediately. When the game is started the recent game state is loaded from the database.

Building objects and vocable query Objects can be bought in the market depending on their state and the level of the player. The building process queries the translation of the vocable several times in the language the player has chosen.

Time between vocable queries The time a player has to wait between vocable queries is defined by the vocable template created in the administrative interface. Between two queries the user can perform any other action in the game or end and restart it later.

Show timer for vocable The tooltip for vocables in the building process shows the amount of time in hours, minutes, and seconds till the next query is available.

Highlight elements with timer zero Vocables which can be queried are highlighted by a colored border. In this manner the player can see which vocables have a timer that is zero without having to hover over every single object.

Statistic for vocables The tooltips for completed vocables and vocables in the building process both show the number of time the vocable has been translated correct and incorrect. For completed Vocables also the translations are displayed.

Invite neighbors Through the request popup the player can invite other players as Neighbors as well as accept invitation sent to him.

Visit neighbor The neighbor bar enables the player to visit his neighbors and therefore enter the neighbor state of the game interface. When visiting a neighbor the player sees the home of the neighbor and can switch between all areas the neighbor has activated.

Help neighbor In the Neighbor state the player can help for vocables in the building process depending on the time he has helped the neighbor the last time.

Gift neighbor The player can send his neighbor gifts and open the received ones through the gifts popup. It can be accessed through the neighbor bar, the separate gifts button or in the neighbor state. To open a received gift the player has to answer the question belonging to the gift.

Administrative Interface

Login mechanism The login mechanism is realized through the ACGI plugin for the Grails application. To access the administrative interface the player has to log in with his/her username and the corresponding password.

User account The Administrative Interface offers no registration functionality. Just administrators can access the user account management and create new users.

Create GameInstances At the main page of the administrative interface users can create new *GameInstances*. Therefore they also have to create a new initial area for the instance.

Edit current GameInstances All templates for areas, vocables, designs, and gifts can be edited through the administrative interface.

No access to player data No data that stores information about game states or players can be viewed or edited through the administrative interface. Just the game interface can access these data through the remote services.

Create new game objects The user can create new game objects. Therefore the criteria that are defined in domain classes of the Grails application have to be followed. The application just creates objects which are valid to these criteria.

17.2 Educational Entertainment

Like for every educational entertainment product the main challenge for VocVille was to find a good balance between the game play and the learning potential. The first task was the integration of the queries of vocable translations into a game story, or more precisely said creating a game story around the queries. Creating objects is a good metaphor for learning a vocable in that way that in both processes the result can be used afterwards. The metaphor also allows integrating objects that represent vocables that the user can relate to because they are objects of the real world. A disadvantage of this metaphor is that only things can be created and therefore only vocables for nouns that can be represented graphically can be used in the game. However it would be possible to create areas which do not represent real spaces like a kitchen or a restaurant but abstract terms like electricity or the weather. In this manner it is also imaginable to create areas with vocables for adjectives or even verbs.

Another question is how the learning process is influenced when a player does not play for a long time. At the moment the application assumes that the player knows a translation when he has enter it several times. When the player starts building an object, translates it a few times and then does not play for a longer time, it is possible that the desired learning process is not fulfilled. Instead after the break the learning process of the player starts again from the beginning but the building process within the game continues at the old state. Also it is very easy to access translation services on other web pages since the game takes place when the player already is surfing in the internet. But this problem lies in the responsibility of the player and his/her motivation to really learn something with the game.

17.3 Social and Casual Gaming Aspects

VocVille can be seen as a casual game because the player has to perform a lot of small, separated task. The amount of time for a single game session can be very short. Because every interaction with the game is directly reflected in the database the player can choose if he wants to perform a lot of task or just do a few and continue the game at a later time. The simple game story and the easy to use game interface allow even inexperienced players to use the game.

With the ability to invite neighbors and visit their homes as well as send them gifts VocVille has several features that define a social game. The accessibility through Facebook also indicates that VocVille is a social game. The benefits of playing it together are; having access to game objects that can only be achieved as gifts and making faster progress by the help of neighbors. Beneath that the players are making the same game experience and can talk about them outside the game. Because they learn the same vocabularies it is very easy to have conversation in the learned language using this common base of vocabulary.

17.4 Grails

The Grails framework is a very powerful tool to create web applications that manage data stored in a database. The programmer does not have to deal with database related details since Hibernate transforms the domain classes into tables of the DB and offers a simple to use API to create, save, update and delete entries in these tables. Nevertheless it is necessary to learn the basics of database management mechanisms in order to create well designed and functional domain classes. The dynamic methods that are injected by Grails and Groovy at runtime are also very time-saving mechanisms of the framework. When building the first application with Grails it can be hard to remember all these methods. Therefore it is really important to use an IDE which helps the programmer with this by offering code compilation. When I started developing VocVille I used the Grails plugin in for the NetBeans IDE which does not have code compilation. It made it very time consuming to consult the Grails reference every time I needed to use a unknown feature of the framework. At the middle of the project I discovered the IntelliJ IDEA IDE which (in the commercial “Ultimate Edition”) has full support for Groovy and Grails including code compilation for the dynamic methods. With this IDE it was much faster and more comfortable to implement the Grails part of VocVille.

A disadvantage of the Grails framework is its very slow compiling time especially when compiling the application as a war file. For this Grails compiles the application into a war file and afterwards decompresses this file into a folder to run the application on the local host. This process can take up to several minutes depending on the size of the project. Because the Blaze4DS plugin needs the application to run as a war in order to connect with the Flex application I was forced to compile VocVille this way. Fortunately when developing the Administrative Interface the application did not need to be compiled as a war and therefore I could realize the implementation of it much faster.

17.5 Flex

The Flex framework is a very good choice to create graphical interfaces. It offers a variety of typical GUI-components like buttons and text inputs. Because the components are rendered in the Flash Player plugin they have nearly the same appearance in all browsers and operating systems. For developing the Flex application the Adobe Flash Builder was the IDE of choice since it is just designed for creating Flex applications. Its design-mode allows arranging the components in a WYSIWYG-manner. In the code-mode standard IDE features like code compilation and syntax highlighting help to create the program logic for the GUI-components and the application. Although the recent version 4 has a lot of new features like the data services it still has some bugs which affect the development process. Every once in a while the IDE throws an error and closes itself when opening the workspace during the initialization of the IDE. To be able to open the IDE again one has to delete (or at least rename) the directory in which the recent workspace is located. This lets the Flash Builder create

a new workspace where the project can be imported again. Another annoying behavior of the IDE is that it very often cannot create a connection to the Debugger of the Flash Player and therefore does not compile the application. This can be resolved by cleaning the project and recompile it again.

17.6 System Architecture

The multilayered MVC architecture of VocVille allows a fine-granular modeling of the data. The main data model is optimized to reduce redundant data in the database by separating the data into templates of objects and instances for game state information. The DTO model is optimized to reduce redundant data transfers between the two applications of VocVille. When running the game, the game interface just requests the data it needs in the particular situation. To achieve these goals it was important where certain functions were implemented. It would have been possible to implement the whole program logic into either the Flex application or the Grails application, which both would not be a good solution. On the other hand it would have been possible to implement the program logic in both applications which could lead to different behavior. I decided to implement the most of the logic into the Grails application since it manages the persistence data which at any time reflects the recent game state. This also has the result that new game interfaces, e.g. a HTML5 version of the game, do not need to implement the whole application again. But to reduce the number and size of the requests between Flex and Grails I also had to place parts of the logic into the Flex application. That is why for example in the querying process, the validation of the user input is implemented in the Flex part since at that moment it already has the information about the correct translation. After the validation the result is sent to the Grails application which then performs the corresponding actions on the database and sends the updated data back to the game interface. The architecture of VocVille also allows deploying the application on different servers. The database can be deployed on a separate database server, the Grails application can be deployed on an application server and the Flex application can be deployed on a regular web server. In this manner the application is scalable and can change the deployment environment according to the required needs.

A small disadvantage of the architecture just shows up when the application is implemented by a single programmer. This disadvantage is not a unique one for the VocVille architecture but is typical for web applications in general. Using four different programming languages (Groovy, GSP, ActionScript, MXML) in two different IDEs demands the programmer always to be aware of in which environment he works at the moment. A good example for this is a regular variable declaration. In Groovy as a programming language the order for a declaration is: type definition followed by the identifier. In ActionScript as a scripting language the order is the other way around.

18 Usability and Accessibility

The usability and accessibility requirements vary for the two parts of VocVille depending on the expected user groups. The game interface which is designed to be used by the end user has a big focus on these requirements. The administrative interface which is intended to be used by professionals has a bigger focus on the correctness and the functionality requirements. The next two sections describe the different usability and accessibility requirements for each application.

18.1 Game Interface

As a casual social game the game interface of VocVille has to be easy to understand and should be usable intuitively. To achieve this every function of the program is displayed in a different, modular popup window. Allowing the user just to open one popup at a time helps him/her to focus on the recent task. Tasks that are related to vocable objects like querying are accessed by clicking on the object. This reduces the number of buttons in the interface and therefore makes it less complex. The highlighting of objects on which certain actions can be performed also support the user recognizing the recent possible action he/she can take.

The game will be accessible through the Facebook website. The users can use their accounts in Facebook and do not have to go through a complete registration process for playing the game. This also makes it easier to invite friends because the users can access their already created social network within the Facebook platform. Another benefit of using Facebook is that the user, while he is waiting for repeating vocabulary queries in the game, can do other things on Facebook and then go back to the game.

18.2 Administrative Interface

The administrative interface will be used by professionals to create and manipulate the game objects. Therefore it can be expected that these users are willing to handle a more complex interface. Nevertheless the interface allows easy access to the database. It visualizes the abstract data entries by displaying the images of the objects and presenting the properties in a logical order where related values are shown beneath each other. Relations between the game objects are shown when a certain object is displayed. For example the view for a vocable includes links and image for all designs and questions of the certain vocable object. This helps the user to analyze the structure and order of the whole game instance. Menu buttons at the top of each site allow to navigate to the next higher level within the data structure and also help the user to orientate within the structure.

Part VI

Conclusion

With the help of educational entertainment products like VocVille people can learn things in a more amusable manner. They make it possible to use the motivation of playing a video game for animating the users to learn otherwise abstract appearing content. The creation of an educational game requires a well-defined balance between the game play and the educational aspect of the product. In the case of VocVille it was important to design the game story interesting enough so it motivates the player to learn the vocabulary in order to progress in the game. By creating objects the player can see his learning success directly. The social components of VocVille allowing the comparison with other users of the game which can result in an even higher motivation.

The administrative interface of VocVille allows authorized users to edit the existing game objects or create new content for the game. This allows teacher or game designer to continuously enhance the game and the learning subjects. Also complete new instances of the game can be created. Because the templates for game objects are very flexible it is possible to create game instance for learning other topics than just vocables, like mathematical knowledge or other subjects that represent knowledge that needs to be memorized.

During this project I once again realized how important the first phases of a software developing process are. The initial analytic of the system helps to understand what the application should be able to do and what requirements have to be accomplished for this. In the case of VocVille I started with thinking about possible use cases. After I figured out what the application should do and how it should react on user actions it was easier to define a first version of the data model. Of course the model was modified several times during the development process. The initial analysis also helped to define a domain specific terminology for the application.

Like every software and especially web applications, the VocVille application could be improved by new features. It is desirable that the game could also be used to learn verbs, adjectives and other types of words. In the case of verbs it would be possible to add animations to vocable objects that visualize the meaning of the verb. Adjectives could be realized by designs which add the meaning of the adjective to the game objects. Also features for learning grammar are imaginable.

Part VII

Appendix

19 References

References

- [1] Adobe flash builder homepage. [Online]. Available: <http://www.adobe.com/products/flashbuilder/>
- [2] Eclipse homepage. [Online]. Available: <http://www.eclipse.org/>
- [3] Facebook connect plug-in for grails. [Online]. Available: <http://www.grails.org/plugin/facebook-connect>
- [4] Facebook graph api reference. Just accesible with a valid Facebook account. [Online]. Available: <http://developers.facebook.com/docs/reference/api/>
- [5] Facebook graph plug-in for grails. [Online]. Available: <http://www.grails.org/plugin/facebook-graph>
- [6] Farmville homepage. [Online]. Available: <http://www.farmville.com/>
- [7] Farmville page on facebook. [Online]. Available: <http://www.facebook.com/FarmVille>
- [8] Groovy homepage. [Online]. Available: <http://groovy.codehaus.org/>
- [9] Oaouth 2 homepage. <http://wiki.oauth.net/OAuth-2>.
- [10] Ruby on rails. [Online]. Available: <http://rubyonrails.org/>
- [11] Zynga game network inc. [Online]. Available: <http://www.zynga.com/>
- [12] Zyngas farmville becomes largest and fastest growing social game ever. [Online]. Available: <http://www.sys-con.com/node/1084929>
- [13] (2010, 10) Facebook statistics. [Online]. Available: <http://www.facebook.com/press/info.php?statistics>
- [14] (2007) Casual games market report 2007. Casual Games Association.
- [15] J. Gee, "What video games have to teach us about learning and literacy," *Innovate 1 (6)*, vol. 6, 2005.
- [16] M. Helft. (2010, Sep) Virtual goods expected to grow by 40 percent next year, study says. <http://bits.blogs.nytimes.com/2010/09/28/virtual-goods-expected-to-grow-by-40-percent-next-year-study-says/?scp=2&sq=farmville&st=cse>. The New York Times.

- [17] M. Papastergiou, "Exploring the potential of computer and video games for health and physical education: A literature review," *Comput. Educ.*, vol. 53, no. 3, pp. 603–622, 2009.
- [18] G. Rocher and J. Brown, *The Definitive Guide to Grails*, S. Anglin and T. Welsh, Eds. Apress, Springer-Verlag New York, 2009.
- [19] L. Shklar and R. Rosen, *Web Application Architecture*. John Wiley & Sons, Ltd, 2003, iISBN-10:0-471-48656-6 ISBN-13:978-0-471-48656-5.
- [20] Whatwg wiki. Web Hypertext Application Technology Working Group. [Online]. Available: http://wiki.whatwg.org/wiki/FAQ#When_will_HTML5_be_finished.3F

20 List of figures

List of Figures

1	Grails architecture	14
2	Flex architecture	16
3	The MVC pattern	28
4	Use Case Diagram for VocVille	29
5	Activity diagram: Build an Object	31
6	Activity diagram: Show Object/Timer	32
7	Activity diagram: Activate Area	33
8	Activity diagram: Invite Neighbor	34
9	Activity diagram: Visit Neighbor	35
10	Activity diagram: Send Gift to Neighbor	36
11	Main data model	38
12	DTO data model	39
13	Vocable classes	40
14	Area classes	42
15	States of vocable objects	44
16	Area states example	46
17	Translations classes	46
18	Design classes	47
19	Player classes	48
20	Gift classes	49
21	Sitemap for the Administrative Interface	51
22	Main page of Administrative Interface	52
23	Creating a new GameInstance	52
24	GameInstance View	53
25	Area View	53
26	Vocable View	54
27	Design View	54
28	Question View	55
29	Home state	56
30	Change Design Popup	57
31	Vocable Query	57
32	Market	58
33	Gifts	60

List of Tables

1	Gameplay Characteristics of Casual & Hard-Core Enthusiasts, [14]	9
2	Integration methods for Grails and Flex	21
3	Codification of states	44

21 List of abbreviations

- AMF** *ActionMessageFormat* Binary format to serialize Action Script objects.
- DTO** *DataTransferObject* Design pattern to transfer data between applications. DTO are objects classes for storing data without any additional behavior except storage and retrieval behavior. DTO are formely known as Value Objects (VO). This name is still used by the Adobe Flash Builder.
- GORM** *GrailsObjectRelationalMapping* ORM implementation for Grails based on Hibernate.
- GUI** *GraphicalUserInterface* Interface for an application allowing the user to interact with it through graphical items like icons or buttons.
- GSP** *GrailsServerPages* Groovy version of JSP.
- IDE** *Integrateddevelopmentenvironment* Software application for facilitating software development. Usually contains a source editor, a compiler/interpreter, tools for build automation and a debugger.
- JMS** *JavaMessageService* API for sending messages between clients.
- JSP** *JavaServerPages* Web application language to describe dynamically generated web pages .
- MVC** *ModelViewController* Architectural pattern, separates domain logic (model), user interface (view) and program logic (controller) .
- ORM** *ObjectRelationalMapping* Technique for converting object instances to database entities.
- SOAP** *SimpleObjectAccessProtocol* Protocol specification for structural information about Web services.
- VO** *ValueObject* Old term for DTO.
- WAF** *WebApplicationFramework* Development framework for dynamic websites, Web applications and Web services.
- WSDL** *WebServicesDescriptionLanguage* XML-based language to describing a Web Service. Defines methods syntax including their parameters and return types.

22 Used technologies and tools

In the development process of VocVille is used several different technologies and tools. For better comprehensibility and to help other who may need some of these tools for their own theses the used version of them are listed below. To create this thesis and organize the whole project I used furthermore: Lyx, JabRef, MS Word, MS OneNote, UMLet, DBVisualizer and WinSCP.

Technology/Tool	Version
Grails	1.2.2
ACEGI plugin	0.5.3
Hibernate plugin	1.2.2
Remoting plugin	1.1
Tomcat plugin	1.2.2
Facebook-graph plugin	0.4
Blazeds plugin	1.0
NetBeans IDE	6.9.1
IntelliJ IDEA	9.0.4
Flex SDK	4.0.0
Adobe Flash Builder	4.0.0
Rational Software Architecture	7.5.0