UbiqGames: Casual, Educational, Multiplayer Games for Mobile and Desktop Platforms

by

Matthew Ng

S.B., Computer Science and S.B., Management Science

Massachusetts Institute of Technology, 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 2009

[September 2009]

Author _____
Department of Electrical Engineering and Computer Science
August 21, 2009

Certified by _____
Eric Klopfer
Associate Professor, Department of Urban Studies and Planning
Thesis Supervisor

Accepted by _____
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

UbiqGames: Casual, Educational, Multiplayer Games
For Mobile and Desktop Platforms
by
Matthew Ng

Submitted to the
Department of Electrical Engineering and Computer Science

August 21, 2009

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The mission of the UbiqGames project is to develop a suite of casual, educational, multiplayer games that are playable across a wide variety of operating systems and devices. To accomplish this goal, we plan on making use of the expanding availability of wireless networks and the increasing capabilities of mobile browsers to build engaging and dynamic web-based games. This paper will document the progression of the UbiqGames project and take a deeper technical look at the first two games built under its heading: Virus and Weatherlings. Initial questions we wanted to answer were whether mobile browsers could handle complex web applications and whether the small screen size would significantly hinder game play. Results from multiple play tests have demonstrated that these games can be executed successfully, and user feedback has shown that players are receptive of the educational aspects of the game and enjoy the overall experience.

Thesis Supervisor: Eric Klopfer
Title: Associate Professor, MIT Department of Urban Studies and Planning

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my thesis supervisor, Eric Klopfer, for his guidance on the project and on the thesis, for his insight that kept the games going on the right track, and for his effort in creating such an interesting and fun lab environment to work in. I would like to thank Josh Sheldon for his management over many of the non-programming tasks that were essential in getting Weatherlings up and running, and to thank Judy Perry for her tremendous brainstorming skills and humorous comments that sparked so many of the ideas in Weatherlings (thanks for hiring me too!). Thanks to John Pickle for sharing his vast knowledge of weather with the team and for teaching us how difficult it is to make a correct forecast.

Of course, none of this would have been possible without the help of the UbiqGames development team, past and present. Thanks to Patrick Doyle, Julia Ye, Eric Wong, Nicole Bieber, and Erin Fennacy, who all helped hack away at designs, ideas, prototypes, and eventually code, in order to make the project as successful as it is today.

And finally, I would like to thank my parents, who are my source of inspiration behind everything that I do.

TABLE OF CONTENTS

LISTING OF FIGURES

I. INTRODUCTION

Previous research has shown that games can be used as an effective tool to teach kids about a wide range of academic subjects such as math, science, history, economics, etc. Many educational games, such as "Oregon Trail" and "Where in the World is Carmen San Diego?" have gained popularity over the years and newer games are making a big impact, such as "Spore" and "The Sims". However, these desktop games limit the students' engagement to the moments when they are by their computers and have a large block of free time.

What if the model is changed so that instead of the user having to make time for the game, the game is designed to fit into the user's schedule? In this scenario, students would be able to play while waiting in between classes, while traveling on a school bus, or even while standing on line to go see a movie. With advancements in portable device technology, there are fewer reasons why this model can't be accomplished. In fact, by taking advantage of the growing wireless network availability and increasing mobile browser capabilities, we can create educational, interactive, multiplayer games that are usable across a variety of platforms. For my Masters of Engineering project, I plan on exploring this idea further and creating a framework for developing these new types of "ubiquitous" games.

The paper will begin by describing how and why the UbiqGames project was created. Next, it will explain the thought process behind the project's goals, and discuss the feature requirements and technical requirements associated with the project. The focus will then shift to a more technical standpoint and delve into the design, implementation, and testing of the two initial games, Virus and Weatherlings. The paper will conclude by pulling together the practical and

reusable elements into a common framework and describe the ways the framework can be extended.

II. UBIQGAMES

The mission of the UbiqGames project is to create a suite of ubiquitous games that promotes learning in a fun and interactive way. The term ubiquitous is used to describe games that can be played wherever and whenever the user has a few minutes to spare, such as on a desktop at home or on a mobile device in the hallways at school. Furthermore, the games should be educational, meaning that players learn about a concept and how to apply related skills as they interact with the system, and should be engaging so that students will be inclined to play during their free time.

In this section, I will start by reviewing the predecessor projects to UbiqGames and describe their limitations that this new project tries to tackle. I will then move on to discuss some of the high level features that we want to include in our games, as well as the types of technologies that we thought would best support those features. Based on the technical assessment, I will talk about the ways that a few of the initial goals were relaxed, although there are plans to resolve them in the future.

Project History

Participatory Simulations are one of the early inspirations for the UbiqGames project. They are activities that involve people interacting with one another in order to accomplish a goal in a simulated environment. Around the year 2000, a group at MIT developed a simulation about viruses that ran on a custom-made wearable computer. The game was a success as a proof of concept, but the device itself was fragile and costly to make (Colella, 2000). A couple of years

later, a more viable alternative came along when developers in the Teacher Education Program (TEP) group at MIT revisited these games, and rebuilt them on the Palm OS. Seven such games were created on the platform, including "Big Fish-Little Fish", "Discussion", "Live Long and Prosper", "Tit for Tat", "Virus", "Sugar and Spice", and "Nets Work".

The simulations were distributed, tested, and played among high school students as well as Fortune 500 employees. Field studies have shown that using mobile devices encourages students to learn by increasing motivation to participate, developing data collection skills, and promoting collaboration and communication between players (Vahey and Crawford, 2002). Trial runs of these simulations have been executed among various groups of students and teachers, and have consistently yielded positive results. Students enjoyed being part of the simulation and felt a responsibility to do well. Teachers were able to connect concepts from the game to a wide variety of subject matters and reported a high level of engagement among their students (Klopfer *et al.*, 2005).

Since then, TEP has continued to work with participatory simulations and developed a platform, myWorld, which provides programmers with an API to build mobile peer-to-peer simulation games using Microsoft's .NET framework. myWorld handles the low-level system architecture such as network communication and event processing, thereby freeing up developers to focus on game logic and user interface design. Although still in its mid-stages of development, myWorld has shown a lot of flexibility in being able to support a wide array of game types and features.

Palmagotchi (meaning "cute Palm", a play off of the popular *Tomagotchi* toy's name) is the pilot game built on top of the myWorld platform. Using concepts from Darwin's observations of the finches of the Galapagos, Palmagotchi simulates the evolution of birds and flowers based on the effects of mating, feeding, and environmental patterns (Klopfer, 2008). Users are initially given a set of birds and flowers that coexist on an island, and are told that the goal of the game is to keep the birds alive. As time passes, birds age and lose energy, environmental disasters strike, and predators attack, so players have to make important decisions to maximize the probability of survival of their creatures. An example of one such decision in the game is finding an "appropriate" mate because new generations are born with the traits that are genetically passed down from the parents.

In order to emphasize the dependencies that exist among populations in an ecosystem, many actions in the game require a joint decision between two players. For example, birds cannot forage for food on their own island— they must retrieve nectar from flowers on a different island that exists on someone else's device. Mating also requires finding birds with desirable characteristics on other islands. Because of these important interactions, as well as a need to constantly collect data, maintaining a stable infrared and/or wireless client-server connection is essential for the game to run smoothly.

The use of handheld devices has proven to be an effective teaching tool because it creates an active learning environment for the students and provides concrete results for teachers to refer to (Klopfer *et al.*, 2005). With the development of myWorld, a richer set of features can be added into simulations to cover a wider array of concepts. Palmagotchi is an example of a highly

successful game running on myWorld, but there are still two drawbacks that would be helpful to overcome.

First, communication between two devices can be improved. Palmagotchi implements a custom wireless packet design that optimizes the amount of data that is being sent over the Wi-Fi, but the transmission experiences a high rate of packet loss. The performance further decreases if devices are being used across different subnets. Communicating via infrared is an alternative if the devices support the capability, but this mechanism has a relatively short communication range, which imposes a strict distance limitation.

Second, the game is developed for a specific version of an operating system, which makes the game harder to maintain. Tweaks to the code base need to be made whenever there is an upgrade to the OS, such as when the Pocket PC 2003 upgraded to Windows Mobile 5 and then to Windows Mobile 6. However, changing to a different OS would require redeveloping from scratch, which is unfortunate since the Android and iPhone devices are growing in popularity and the mobile phones sphere is still evolving. These drawbacks raise a question of whether a game could be developed such that it allows users to interact over long distances and could be played on any operating system and on any device.

Project Vision

In addition to developing games that are widely accessible and widely supported, there are several other goals and features that are part of the UbiqGames' vision. These principles were derived from previous experience, research, and user feedback of other educational, mobile

games, such as the Participatory Simulations described above. By defining these principles early, game concepts and design ideas were able to be evaluated against the following criteria:

- The games should be interactive and involve multiple players.
- The games should be playable in short bursts and whenever the user has free time. This means that besides the multiplayer activities, there needs to be substantial single-player activities as well (for the times when no one else is online or when there is no Internet connection available).
- The game should be playable both on desktops and on mobile devices.
- The game should be educational, so two main classes of users need to be considered:
    - **Students** should be presented with a concept, and by learning that concept, they can advance in the game. Feedback should constantly be provided so students know what they are doing right and explain how to correct what they are doing wrong.
    - **Teachers** should be able to track how students are interacting with the system and to determine which students are having trouble with which concepts.
- Alerts should be available so players know when they have something critical to do. Since players may only have a short amount of free time, it should not be spent figuring out what actions are available.
- In-game messaging would be a useful feature to have so that players can communicate with each other, even at long distances.

Technology Assessment

In order to develop cross-platform multiplayer games on mobile devices, the following core set of technologies was chosen to help achieve this goal:

1. Browser-based: To support an array of mobile platforms, including iPhones, Androids, and possibly Windows Mobile devices, as well as the traditional laptops and desktops, a common interface to all those devices was needed. This requirement meant writing native code specific for an operating system would not be a viable option. Since the devices all support Internet browsers (albeit different ones), web-based development seemed like a good place to start. By taking advantage of the Internet and HTTP, the ability to "connect" multiple players would then implicitly be available without having to develop a custom network layer.

   Designing a browser-based game has several other implementation implications as well. The majority of game models will be placed on the server-side rather than on the client-side, so finding intelligent and efficient ways to synchronize and report the game state to every client will be an important task. The same goes for the game logic, although the advantage of passing every action to the server, except maybe for some client-side JavaScript-ing, is that managing a central logging repository becomes easier. Delivering global notices and controlling event execution is easier as well, since the modification can be made just once on the server and the change will propagate to clients on their subsequent request.

2. Application framework: After researching several options for developing our project, we chose Ruby on Rails, a feature-rich web application framework, for the following reasons:
    a. Rails has proven by example that it can handle enterprise web applications.
    b. Both Ruby and Rails have mature development communities and extensive documentation, so help on common issues should be readily available.
    c. Web applications can easily and quickly be setup on Rails (as opposed to the more labor intensive Java web servers) and the Ruby syntax is simple to learn. These requirements are favorable to our project dynamic and group structure, since developers routinely rotate over short periods of time.

3. JavaScript (and in particular, AJAX): JavaScript is an important technology that can be used to provide user interface logic, to request and load dynamic content, and to give websites the "feel" of an application. This technology will probably be a limitation of the application as well, since the JavaScript language is not as powerful as native code, and mobile browsers only support a subset of its full capabilities.

4. Database: In theory, any relational database would work, but MySQL was chosen because it has been known to be scalable, is well-supported, and can be easily integrated into a Ruby on Rails project.

Revised Goals

The technology assessment showed that some of the original project goals may not be feasible with the current level of mobile browser support, so revisions to those goals were made. The following list describes some of the requirements were relaxed and the additional assumptions that were made:

- Games do not have to be playable at any time, but instead, only at times when an Internet connection is available. However, in future iterations, incorporating offline browsing technologies, which allow web applications to work without Internet and to resynchronize the next time the user gets back online, may be an advantageous feature.
- As a result of the previous change, players that are using the system are assumed to have a steady Internet connection, although the game should degrade gracefully if the connection gets lost.
- Although ideally every browser on every device will be supported, not all browsers implement the AJAX functionalities well, in particular and surprisingly, Internet Explorer and Opera Mobile. Therefore, only a subset of browsers will be supported, including: Safari on the iPhone, Android's browser, Iris on Windows Mobile, and Firefox on desktops. As browser standards continue to improve, this list is expected to grow.
- In theory, the pace of the game should adapt to the schedule of a player, but for practical reasons, not imposing any time limits in a multiplayer game can cause a lot of frustration to other players who are waiting. Therefore, players are assumed to be able to play at least once a day, and if not, the game will progress as if the player chose to pass his or her actions.

- Although the games should be playable on desktops and mobile devices, the focus will be on the times when the desktop is not available. Traditionally, popular websites such as Digg, Twitter, Amazon, etc., only provide a trimmed down version of their page for their mobile users. However, a different methodology can be used by reversing the model. Our games should be designed to be fully viewable and functional on mobile browsers, and to display the *same* version of the site on desktops, without implementing any additional desktop-specific features. For example, games will only use a small portion of the desktop screen, and will avoid mousedown and mouseover actions like "drag and drop" and tooltips. This idea may seem like a step backwards, since the ability to detect browser and device types exists, but using the same version for both platforms will keep the application's look and feel consistent across devices and will avoid any advantages from using a desktop. Device detection, though, can still be useful in some ways, like to optimize image quality and to adjust font sizes.

With these revised goals and ideas in mind, the UbiqGames project was ready to design and develop its first game.

III. VIRUS

Virus was the first game to be built on the UbiqGames platform, although in many ways it was just a prototype to test different types of mobile and web technologies. Questions that this pilot game was designed to answer were whether mobile browsers were capable of supporting multiplayer games and if so, what are its limits. How much content can webpage hold before loading and rendering times become an issue? And before screen size becomes an issue? Is JavaScript sufficiently supported by the browsers or will the web pages need to rely on basic HTML? Can we deliver real-time data to users so that the application feels like a dynamic game rather than a static website? And since an unfamiliar web framework was being used, figuring out the ins and outs of Ruby on Rails was important as well. Since these technical questions were the main concern, an existing game was chosen to be re-implemented and improved upon instead of completely designing a new game.

In this section, I will discuss the ideas behind the Virus game, the implementation details, and the lessons that were learned from the experience.

A. Game Play

The basis for the Virus game came from one of the earlier PDA simulations, also called Virus, which is a game that challenges players to meet as many people as possible without getting their virtual player sick. Initially, there is a Patient 0 who is infected with a virus, and the virus gets transmitted with some unknown rate when two people meet. There are also people who are designated as immune to the virus and won't ever get sick, although they could possibly be

carriers. The goal is then to try to figure out the transmission pattern of the virus and how to meet the most people without getting sick.

In the new version of Virus, the same concept of having players try and figure out the transmission pattern of a virus was used, but instead of managing a single virtual person, each player manages an entire virtual country. Within each country, there are cities where virus outbreaks can occur. Outbreaks can spread to neighboring cities, and when it reaches the country's borders, it can continue spreading to neighboring countries (but players are not told who their neighbors are).

Instead of spreading just a single virus, multiple strains of the virus appear throughout the world. Each strain has its own transmission rate and degree of virulence, which affects how fast the virus spreads and how many people it can kill, respectively. The virus also has a probability of appearing and disappearing on its own. The game progresses using time ticks, and at every tick, the virus can spread (or die) and a country's population may fall. After some set number of ticks, one year will end and another will begin, and the simulation is repeated over some number of years.

The simulation is not completely passive, however. There are several tasks available to players so that they can prevent their country (and the world) from becoming disease ridden. First, players are allowed to scout their own country by inspecting the cities where there are outbreaks. This tells the player about the virus' transmission rate and virulence degree. The player can also scout other countries and learn about the type of outbreaks that are occurring there (but not the

details of the virus causing the outbreak). However, each country only gets a limited number of scouts per year, so players need to decide how to manage this resource.

Players can use the information they gathered about the viruses to help them choose which strain to create a vaccination for. This is decided through a global voting system held near the end of every year, and the ONE strain that gets the most votes (ties breaking arbitrarily), will disappear from the world starting at the beginning of the next year. Players are then scored based on their vote in the following manner: a simulation is run to calculate what would have happened if no vaccinations were applied and is compared against what actually did happen. A player gets points if their vote matches the strain that would have been globally the deadliest and also based on the number of people in their country that were saved because the group vote vaccination was applied (therefore, part of the score is based on the global effectiveness of the vaccine and part of the score is based on the local effectiveness).

B. Implementation

Virus was developed using Ruby (v1.8.6) on Rails (v2.1.0) backed by a MySQL database (v5.0). Because the project uses the Rails framework, the code structure naturally falls into the Model-View-Controller (MVC) paradigm. Therefore, to understand the system architecture, I will discuss each of the parts separately and highlight the important elements in each layer.

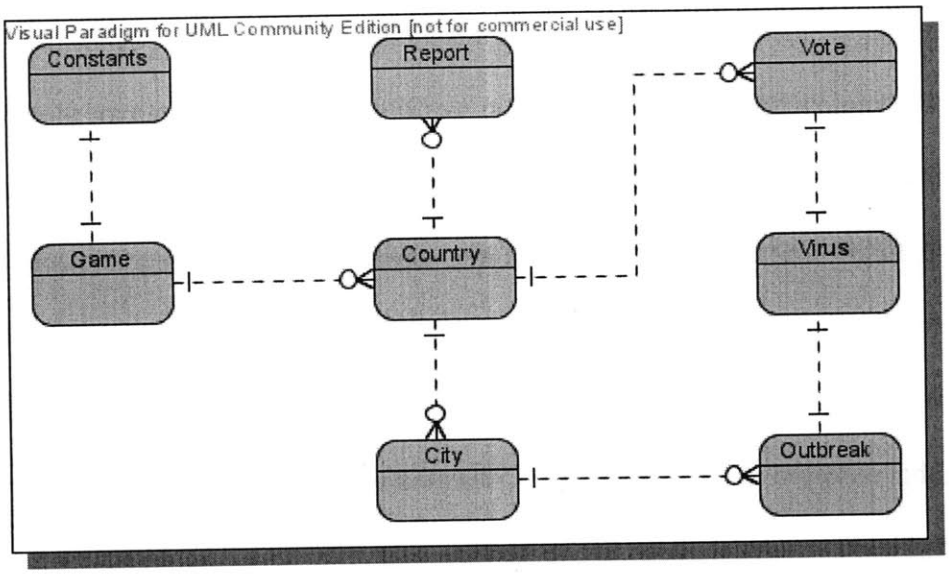


Figure 3.1 Entity relationship diagram for Virus, illustrating the core models (the complete version can be found in Appendix A).

The Models (M)

A Game object in Virus has three important functions. First, it allows players to be associated with a specific game so that multiple game instances can be run on the same database. Second, it is the only model that references the Constants model object, which holds the global game parameters, so the settings can transparently be switched in the middle of the game (possibly to alternate difficulty levels). Third, the Game holds the state indicator that determines whether or not voting is enabled.

The User object represents a person who has signed up to play Virus. However, it only captures the registration information of a player, such as their name, password, and email address. To store the person's game information, a parallel Country object is created, and is related to the User through a foreign key ID. The Country keeps track of the player's score, the number scouts they have remaining, and other game-related information.

A country has many cities, each of which is represented by a City object. Cities have x and y coordinates to denote their position in the game world, and also keeps track of the number of outbreaks that are plaguing the city. The actual outbreak information however, such as the virus strain and the duration of the outbreak, is kept in a separate Outbreak class. This class also stores whether or not the player has scouted the outbreak, since scouted outbreaks should be marked as such, to prevent players from re-scouting the same location. When a city is scouted though, information that is stored in the Virus class is revealed to the player. The data includes the virus' transmission rate, its virulence, its strain, as well as some background information about how the virus gets transmitted.

The last model of interest is the Vote object, which gets created for every player every year. If the current year's vote has not ended yet, then the Vote object is used as a data tracker so that players can review their pick and change it if necessary. Otherwise, the Vote objects are tallied up, and the virus with the most votes gets removed from the game. After the votes get processed, the Vote objects are still kept in the database in order to help calculate scoring at the end of the next year.

The Controllers (C) and Supporting Modules

There are eight main controllers that control the flow of the Virus game, and two supporting modules that help progress the game at constant time intervals. The UserController and SessionController are classes that are generated by running the "restful_authentication" plug-in. They manage the actions related to signing up for a new account and logging into an existing

account. Once the user has been verified, the control is handed off to other controllers until the player logs out again.

The GameController usually takes over right after the player signs in. Its only role is to check whether or not the game has started and redirect the player to the correct next screen. If the game hasn't started, the player is brought to a tutorial screen, but if it has, then the control is handed directly to the CountryController. The main task for the CountryController is to gather all the outbreaks that are occurring in a country and to display the most up to date information to each player. A similar task is performed by the WorldController, which gathers the outbreaks in *all* countries, but then it also needs to decide which subset of that information should be returned to the player.

When a player scouts a city or another country, a request for an outbreak report is handled by the ReportController. To generate a city report, the controller will gather data on the current conditions in the target city, as well as a history of all the outbreaks that has plagued the city in the past. For a world report, the controller returns all the viruses that are currently infecting the target country, but details about a virus are only available if both players have scouted that virus in their own respective countries beforehand. If the details of a virus are available and players wish to view the details, then a request is sent to the VirusController to collect the necessary information.

The two steps of the voting process that are handled by the VoteController are to gather the viruses that are eligible to be voted on and to register the players' choices as they are made.

However, as mentioned before, the votes are not processed until the end of the year, so the picks are just stored as Vote objects in the database. The logic that handles the actual processing lies in timestep_worker.rb, which is a module who's main task is to handle time changes and timed events in the game. In the Virus game, time progresses independently of the activity of the players (i.e., every twenty seconds in real time increments the game time by one unit), which does not quite fit into the request-response model of the Internet. Therefore, in order to get the logic to run outside the HTTP request cycle, the "background_rb" plug-in is used to create a background process on the server that calls methods in timestep_worker.rb. The process can be configured to execute predefined functions at scheduled intervals by invoking system level calls on the server. Therefore, when the timestep_worker.rb functions are invoked and determine that the end of the year has been reached, additional functions are called to help process the Vote objects, remove the virus of choice from the game, update each player's score, and then to reset the yearly parameters.

The Views (V)

Since Virus was created as a prototype game to experiment with mobile technology, the UI was designed foremost to be functional rather than aesthetically pleasing. Therefore, many of the screens have a simple text-based layout, but there are still a few interesting UI elements to examine.

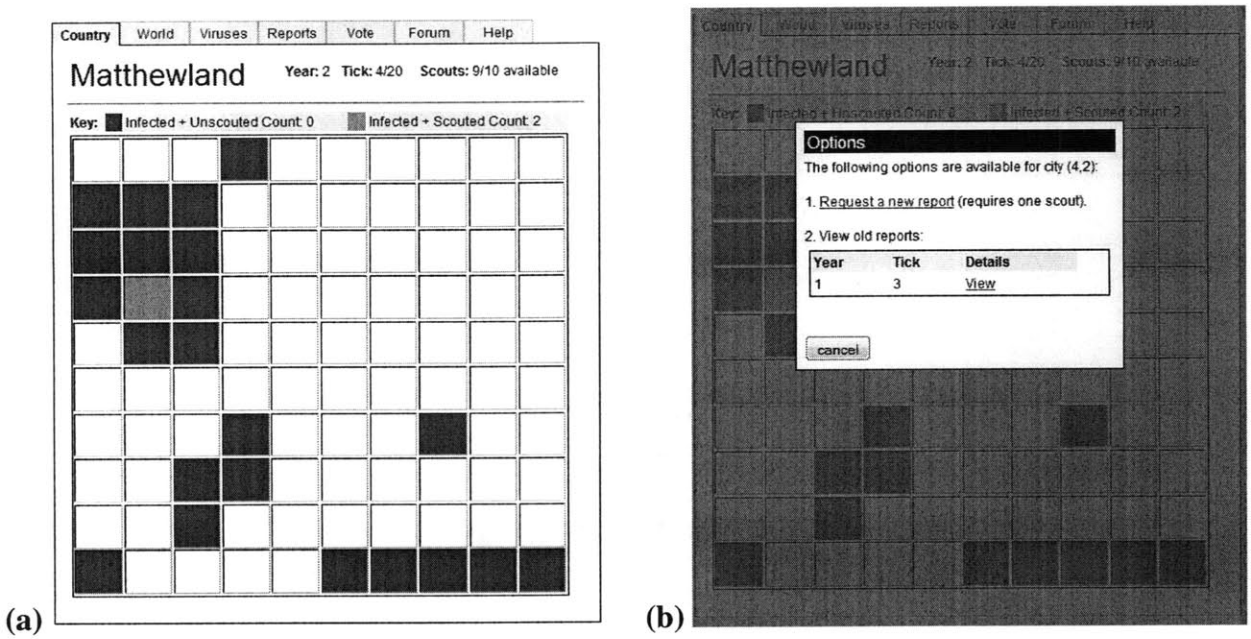


Figure 3.2 (a) The country screen and (b) the modal overlay screen.

As shown in Figure 3.2a, a player's country is depicted as a square grid, where each box in the grid represents a city. If a city has an un-scouted outbreak, the box is shaded bright red, and once the outbreak has been scouted, then the box is shaded gray. Otherwise the box remains white, so by using this color variation, the status of a city becomes easily distinguishable. The map legend is available at the top and contains other useful statistical information, such as a running count of the number of infected cities.

When a player decides to scout a city by pressing on the corresponding box, they are not automatically redirected to the report screen. Instead, a modal dialog is utilized, as shown in Figure 3.2b, to provide additional options, like whether to actually scout the city or instead to view an old report for the city. This option is useful for the instances when a previously scouted city becomes re-infected, in which case players have a choice of finding out new outbreak data, or reviewing virus data for an old outbreak.

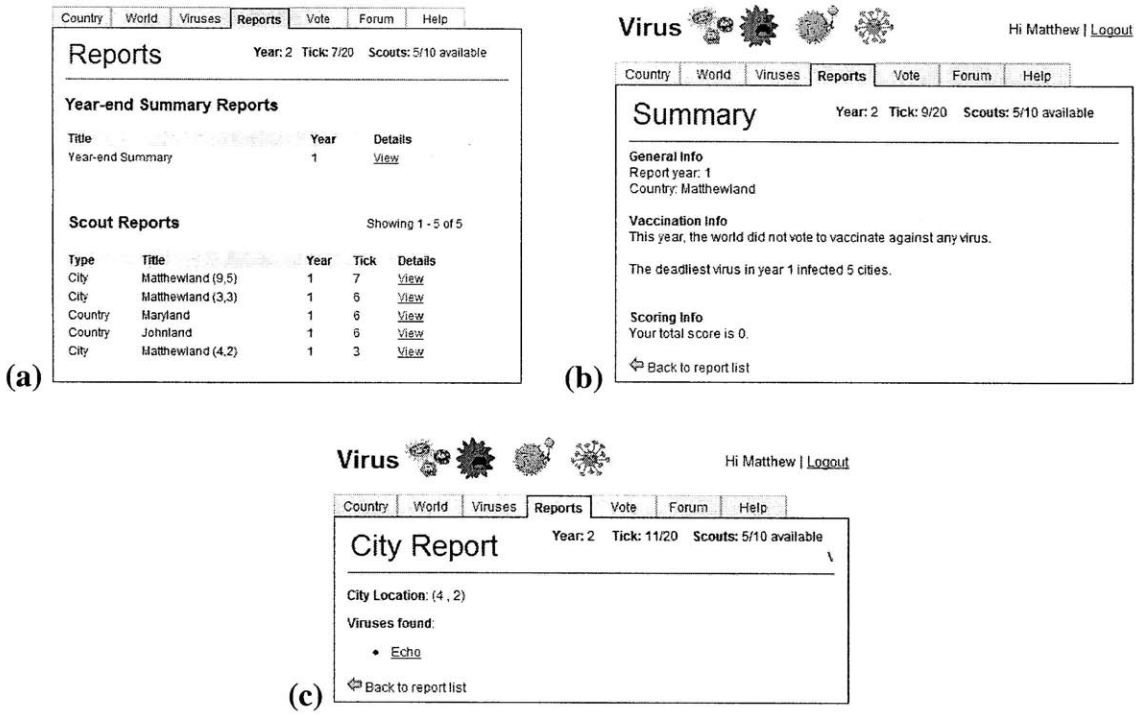


Figure 3.3 Various reports a player can request in Virus: (a) the reports index page, (b) a year-end summary report, and (c) a city outbreak report.

Figure 3.3 shows the different types of report screens that present players with details about game events and provide feedback on their previous actions. Figure 3.3a is the report index screen, which shows a list of all the reports available to review. The reports are grouped by functionality and then sorted by descending date. Clicking on a "Year-end" report brings the user to a screen similar to Figure 3.3b, which summarizes how the vote turned out for a particular year, what was the impact of the previous year's vote, and how many points were scored by the player for that year. Clicking on a "City" report brings up Figure 3.3c, which simply lists the city's location, the time of the report, and the virus outbreaks that are infecting the city.

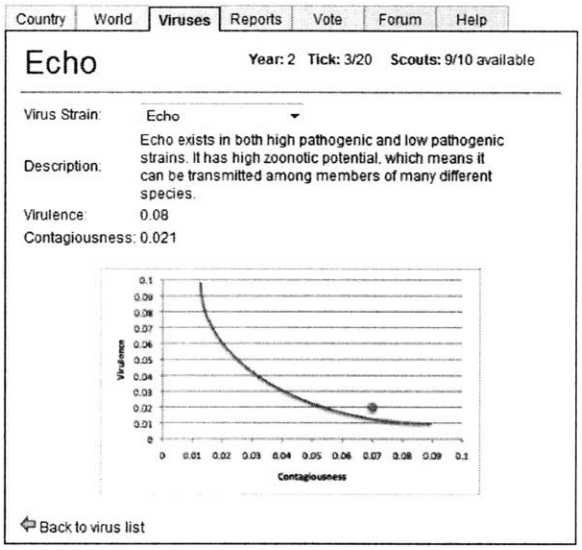


Figure 3.4 Details about a specific strain of a virus

The details of a virus are rendered on its own individual screen, as shown in Figure 3.4. There is vital information displayed in a textual format at the very top, but a graph is provided on the bottom as well, which charts the typical ratio between virulence and contagiousness and where along the curve the current virus' ratio lies. As a navigation convenience, a dropdown menu is implemented to switch between different virus strains.

Lastly, Figure 3.5 shows the different tools and information that an administrator has access to. The top box contains the most general elements: whether or not the game is running, what game parameters are being used, and controls to start, stop, and restart the game. Below that, there is a section which allows administrators to look up aggregate statistics on all players in the game, or by clicking on a particular name, to investigate how that player is doing on an individual level. Then at the very bottom, there is also a feature to generate a comma-delimited file that lists all the outbreaks that have occurred, where they occurred, and how long they lasted.

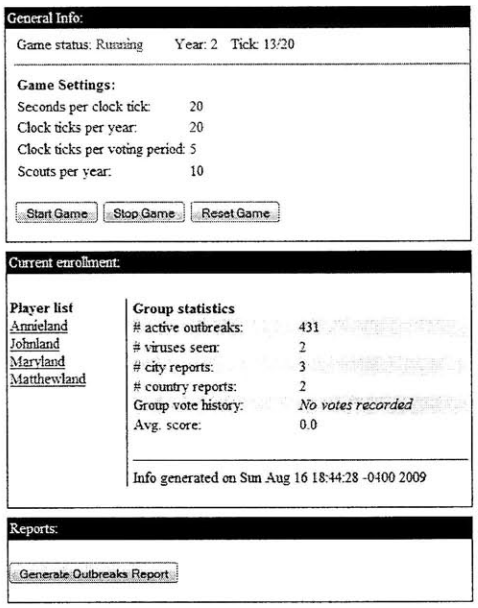


Figure 3.5 The administrator toolbox page

C. Lessons Learned

As a technical prototype, the Virus application should be considered a success because it provided a lot of insight into how and what could be accomplish in a mobile game. A series of informal user tests helped compile the following lists of takeaways from the experience:

- In order for the virus spread to act like a simulation, the country view screen needed to update automatically without the user having to press refresh at every time change. This task was accomplished by using JavaScript's PeriodicExecuter, which proved to be effective on the mobile browsers that UbiqGames supports. This tool provides a powerful functionality that can be used to create dynamic game features like AI characters, randomly occurring events, data streams, and chat rooms.
- The administrator page showed that it was possible to continuously capture the current state of the game and to generate different types of reports in various file formats. The

same procedures can be transported to other games or even extracted into its own external module.

- Running scripts as a background process can be useful, since this is a way to offshoot long running tasks. Coupled with AJAX, users would be able to continue using the site without waiting for the results to complete, providing a smoother user experience.
- Modal pop-ups were useful when more information needed to be displayed, but there was a large probability that users would want to return to the previous page.
- The forum was useful to foster organized in-game communication.
- Advancing the game at set time intervals is one way to move the game forward, but this mechanism places pressure on users to continuously play. If users take a half-day break, many game years would pass and their countries would be infected beyond control, which is an undesirable side-effect.
- Virus showed us some interesting multiplayer dynamics as well. For example, players may want to hide the current state of their country, but find out as much as possible about other countries, so they have to balance information trade. Being deceptive about the state of one's country may be one strategy to sway votes. Another balance players need to maintain is whether to be cooperative and save the world or to focus on saving their own country, since there are scoring incentives for doing each.
- Scouting did not seem engaging enough as a single-player activity and voting did not seem dynamic enough as a multiplayer activity.

Taking these results and experiences from the Virus game, the next project for UbiqGames was ready to be tackled, although this time, more focus was shifted to developing engaging and interesting game mechanics.

IV. WEATHERLINGS

The second application developed under the UbiqGames platform is Weatherlings, a card-dueling game whose educational purpose is to teach users to interpret weather patterns and to nowcast[1] weather conditions. Like many other popular card-dueling games, such as "Magic, The Gathering", "Pokémon Trading Card Game", and "Yu-Gi-Oh", the basic premise of Weatherlings is that players challenge each other to battles using a personalized deck of cards. The twist that Weatherlings brings into the mix is that the cards are strongly dependent on weather conditions, therefore players need to be aware of the current conditions and be constantly forecasting future conditions. Furthermore, the battles take place in arenas, which are modeled after real cities across the U.S., and so the weather patterns that players experience and the weather data that they are given are historically accurate information.

I will begin by discussing the application from the user's perspective: what activities are available and what choices need to be made while performing those activities. Then, I will take a developer's standpoint and explain some of the design decisions that were made and break down the architecture of the system. Next, I will walk through some of the testing and user studies that were performed and describe how the feedback affected the subsequent iteration of changes. Finally, I will give my thoughts on the most important next steps for improving and extending the game.

[1] A short term forecast

A. Game Play

Managing Weatherling cards

Weatherling cards are the primary elements that players get to interact with in the game. Each card has a unique name, an amount of hit points it can endure before being sent to the sideline, an affinity to a specific weather condition, a ranking of how powerful the card is, and a set of moves it can attack with. Figure 4.1 shows a sample listing of Weatherlings with their attributes and moves.

Name	Hit Points	Affinity	Rank	Moves
TriCyclone	50	Wet	Basic	Flash Flood Heal
Desert Hare	80	Dry	Advanced	Caliche Throw Heal
Sorbet Sorcerer	50	Cold	Master	Avalanche Punch
Tropic Sloth	70	Hot	Advanced	Pepper Breath Heal
Ruby Thief	50	Normal	Basic	Bite

Figure 4.1 Attributes of various Weatherlings

Each player starts off with an initial collection of cards and they must decide which cards to take with them into battle. To do this, they designate cards as being part of a deck. Only one deck is allowed per battle and decks have a size limit, so through this mechanism, players are forced to choose a subset of cards that they think will perform the best under different weather conditions. The cards themselves, however, can be part of multiple decks, so players are encouraged to design multiple decks with different combinations of cards. This deck management activity

allows players to explore the cards that they own, as well as to formulate strategies based on how well they think cards will perform in battle and how they think cards will interact with each other. Since decks are not shared between players, deck management is a continuous single-player activity that can be worked on when other players are not online.

Setting up a battle

Once players have created at least one deck, they can head over to the lobby to battle against other players. Battles are set up by configuring the following parameters: the number of rounds the battle will last, the arena that the battle will take place in, and the decks each player will use. The number of rounds is chosen by the host, but the challenger can see this parameter before deciding whether or not to join. By letting this parameter be variable, battles can be flexible to each player's real-world time constraints.

The arena decision process has a somewhat unique mechanism to determine a fair battle location. Out of three random choices, each player simultaneously chooses one arena that they *do not* want, and the last arena is the one that the battle is fought in (in the case where both players choose to remove same arena, a random pick is made between the other two). After the arena is finalized, the players are notified of the location and the month that the battle occurs. Based on this information, they can select their decks, and once this happens, the battle officially begins.

Battling other players

Battling other players is the main task in Weatherlings. It promotes multiplayer game play and requires strategic use of cards and knowledge of weather data. Time is an important aspect of

forecasting, and in battle, the game time progresses independently of real time. That is, the "battle clock" progresses six hours per round regardless of how long it actually takes for the round to complete. The "battle clock" is initialized to some date and time in the past, as predetermined during the setup stages, and is advanced until the desired round limit is hit. Each round consists of two phases: the hand phase is when players draw and play cards from their hand, and the attack phase is when players pick the creatures and moves to attack the opponent with. The phases progress in a synchronous manner, meaning the option to attack is not available until both players have played a card from their hand, and the option to draw a new card only appears after both players' previous attacks have been processed.

A tricky aspect to the battling is system is internalizing how the "battle clock" progresses and when events occur. When a player decides on an attack, he or she is actually picking a move to perform in the *future* because the attack does not get processed for another three game time hours. If the move has a weather requirement, then it will be based on the conditions at that future time, which gives an advantage to good forecasters. After the attack gets processed, players draw and send a card to the field during the same hour, but the ability to choose another attack will not be available for another three game time hours. Given this timescale, attacks from a newly played card will not occur for a total six game time hours (three hours before a move can be chosen and three hours before the chosen move takes effect).

Studying the data

Because of the time delay between when actions are given and when actions are processed, it is important for players to know what will happen in the near future (or at least be able to guess at

it!), so nowcasting is an essential skill to lean. To help players nowcast, a wide variety of weather maps and weather data is given to the players. In particular, they can access hourly temperature, wind, humidity, pressure, and precipitation information from two days before the start of the battle up until the current hour in the battle. They can also view hourly weather maps for the same time range on both a regional and national level. These maps include a combined front and pressure systems map, a temperature range amp, and a satellite imaging map.

Another form of data that is available for players to study is the history of the battle. Each round of battle is associated with a report which includes: the cards that have been played, the attacks that were performed, the damage that was inflicted, and the points that were scored. These details can help players understand the mechanics of the game better, but coupling the data with the resulting weather data, player's may be able to determine what went right or wrong during a previous round and derive alternative strategies that may have worked.

Shopping for new cards

By participating in a battle, players earn cash and experience points, regardless of whether they win, lose, or tie (albeit different amounts). The cash can then be spent at the shop to obtain higher ranking cards (currently, buying cards is the only way to obtain new cards). Card availability is mostly dependent on a player's level, although some cards appear as the game progresses as well, therefore players will want to check back often to see if there is anything they can buy to strengthen their deck. However, to balance the power of higher-ranking cards, they require a certain level of expertise to use and are more expensive as well, so it may take winning multiple battles just to obtain one Master-ranked card.

Leveling Up

The game can quickly become boring if nothing changes to further challenge the players, or if the game is too challenging from the start, then new users may get confused and frustrated easily. Therefore a leveling up system was put in place to progressively add difficulty to the game. As mentioned before, by participating in battles, players earn experience points, and exceeding certain threshold point values will allow players to level up. Levels in Weatherlings are represented as professions— players start off as trainees and then rise to become understudies, scholars, forecasters, and finally gurus. A player's profession is displayed on his or her public profile so other players can check how advanced their opponent is before battling them. But more importantly, advanced professions open up a wider selection of cards that are available at the shop, and unlock tougher arenas where more accurate forecasting is needed.

B. Implementation

Weatherlings was developed using Ruby (v1.8.6) on Rails (v2.1.0) backed by a MySQL database (v5.0), so like Virus, the system architecture follows the MVC paradigm, although being a much more comprehensive game, it make use of library modules to encapsulate a lot of the game logic and setup methods. As with the previous game, I will start by discussing the design of the database models and then talk about how the flow of the system is handled by the controllers and library modules. Next, I will discuss the user interface design and talk about some of the decisions and tradeoffs that were made to accommodate for the mobile device. Lastly, I will mention some of the other important features of the system that do not fit into the MVC structure.

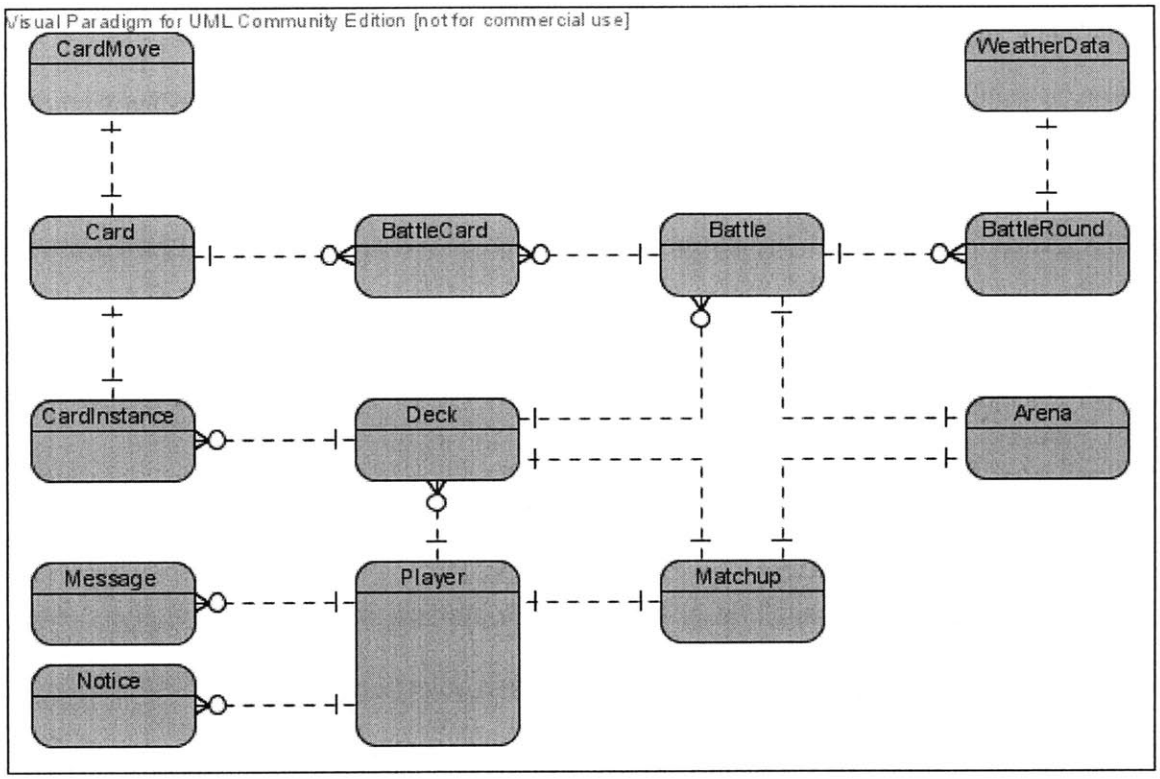


Figure 4.2 Entity relationship of Weatherlings, illustrating the core set of classes (the complete version can be found in Appendix B).

The Models (M)

Figure 4.2 above is an illustration of the underlying database tables that support Weatherlings and the relationship between the tables.

The Game object is very barebones in terms of the information it captures, but its existence allows multiple game instances to be running off of a single database. For example, a teacher may want one game for each class he or she is teaching, so that students can only interact with other students in the same class. By requiring a game code to be entered when players sign up, each student can then be assigned to a game instance and any information delivered to the student will be localized to the specific game they are part of.

The User and Player classes both represent registered users on Weatherlings, but there is a distinct difference between the two— User contains the "real-life" information about a person, such as their first and last name, whereas Player maintains information about the person in a game setting, such as the number of battles he or she has won or the amount of cash they have remaining. Although it may be counterintuitive at first to have two classes represent the same person, the role that each object plays becomes more separable in the context of the application. The User is initially involved in the registration, login, and logout processes, while the Player takes over most of the logic in between.

The Card, CardMove, Arena, and WeatherData classes are basically used for definitional purposes and are unmodified during the game. They represent the data that underlies the core of Weatherlings, and are created through various procedures such as processing XML files, running database migrations, and scraping data off external web sites (in order to obtain real-life weather data and weather maps). The Card class captures the generic information about a card, including its name, affinity, base hit points, and rank, as well as referencing CardMoves to describe the actions it can perform. Each CardMove contains both a verbal description of the move to display to users, as well as a corresponding backend representation so that it can easily be plugged into a damage calculation formula.

Arenas are the locations where battles take place, and currently represent a city in the United States. The model holds information about the city such as its longitude, latitude, and elevation, as well as a code that specifies the weather station that is used to track weather information. The data for each station is then stored in a WeatherData object, which encapsulates the real-world

conditions, like temperature, humidity, wind speed and direction, and mean sea level pressure, for a certain location and a certain hour. Given this representation, the WeatherData table easily grows to be the largest table in the system. Using rough estimates:

(24 hours per day per arena) * (10 arenas) * (365 days) = 87,600 table entries per year

Although databases are known to be able to handle more information than this, queries on this table can potentially hog a lot of system resources if they not carefully constructed and if table indexes are not created.

In order to hold information about the cards that belong to a player and information about how he or she arranges those cards, the Deck class was created. A Deck is basically a collection of cards that a player customizes for the purpose of taking into battle. A player's trunk is internally maintained using a special instance of the Deck class, and is automatically created for the player after registration. However, it cannot be taken into battle and cannot be explicitly modified like a regular deck can, since the trunk is just a reflection of all the cards that a Player owns.

Because players can maintain multiple decks at once, and cards may be part of multiple decks, a separate table, CardInstance, was needed to track how cards were arranged in each deck. CardInstance basically acts as a join table between Deck and Card, but to avoid duplicate rows, it also store the multiplicity of a Card. Figure 4.3 shows a sample diagram of these relations:

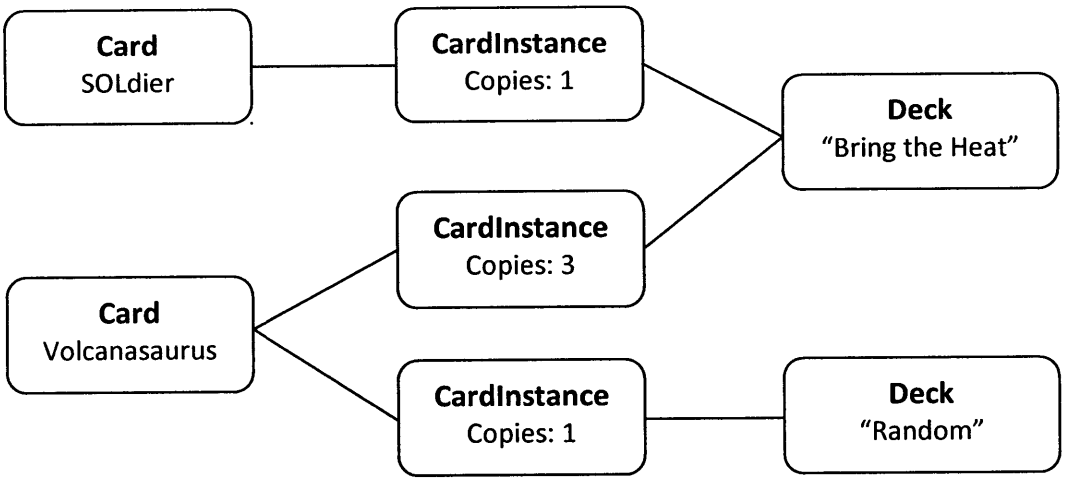


Figure 4.3 A sample relationship diagram between Card, Deck, and DeckInstance

Once two players decide to battle, a temporary Matchup object is created to hold all the information necessary to setup that battle, including the number of rounds to be played, the arenas available to choose from, the decks the players want to use, and the starting date and hour to begin the battle. When all the pre-battle decisions are complete, the data is then copied over to a Battle object and the Matchup is deleted. The design decision to create this transient Matchup class rather than to persist the data in the Battle class was two-fold. First, the setup process is a complex procedure in itself and logically should be considered a separate action from battling. Second, some of the information in Matchup is not needed in a Battle, namely the arena choices that end up being eliminated, so appending these attributes to Battle did not seem like a reasonable solution.

During the data transition from Matchup to Battle, an important collection of objects are generated. Based on the decks chosen by the players, each CardInstance is translated into a BattleCard, which as its name implies, is the card representation when used in a Battle. There are fundamental distinctions between Card, CardInstance, and BattleCard—Card holds the definition

of the Weatherling card, CardInstance specifies Deck that a card belongs to, and BattleCard tracks the card's transient state in battle, such as where the card is located and how much health it has remaining. Although it may seem like this design both duplicates and fragments information, one important advantage of this implementation is that it allows players to use the same deck for multiple battles because in effect, a copy of the deck is generated for each of those battles.

The Battle is a complex and widely integrated class, as can be seen in the entity relationship diagram (Figure 4.2). It holds information about the players that are involved in a battle, the decks that they are using, the arena and time that the battle is taking place, the score for each player, and the current and max rounds to battle for. An important variable that the Battle also maintains is its phase, which is an indicator of the state of the battle. The phase relegates whether players need to draw, play, or attack when the battle is active, but it also records the end result after the battle has finished (i.e., round-limit ending, forfeit ending, admin intervention, etc.).

In order to capture the events that happen during the course of a battle, a helper class, BattleRound, was defined. A BattleRound is created at the start of every round, and is then used to store the cards each player draws and plays, as well as the attackers, targets, and moves that are chosen in the attack phase. It also stores the weather conditions during the hour that the attack takes place, so that the scenario for a specific round can reconstructed with just the corresponding BattleRound object.

Two other classes, Notice and Message, were created in order for the system to provide event announcements to a player and to facilitate interaction between players. Both classes contain a content field and a recipient field. They differ in the fact that a Notice tracks which part of the system the content relates to (for example, "You created a new deck" belongs to the Trunk section), while a Message tracks the player who sent the content.

In addition, an ActionLogger class was generated to provide custom logging features to the system, which was necessary to allow teachers and administrators track and study actions performed by the players. This class is not backed by a database, but instead simply subclasses the built-in Logger class from Rails. The custom class allows control over the log message format and configuration of the output destination. By taking advantage of ActionLogger, developers can dictate what information needs to compile to what file.

The Controllers (C) and Supporting Library Modules

When a request comes in from a client, Ruby on Rails uses a routing module to determine the method that corresponds with the URL and then automatically passes on the request parameters to that action. Depending on the scope and complexity of the request, the controller might hand off the processing to external modules and even spawn long-running tasks. To organize the many different requests, Weatherlings distinguishes between controllers by the types of tasks that it can handle.

Each section within the game has its own controller, although some sections have multiple controllers to break apart the complex action sequences. The shop is managed by a

ShopController, which gathers the cards that are available for the player to buy, as well as handles the buying and crediting process. The DeckController lists the decks that the player has created and processes modification requests such as creating, editing, and deleting a deck. However, to list the cards that are part of a deck, control is handed off to the CardController, whose other role is to display detailed information about each card.

The headquarters is managed by the PlayerController, whose purpose is to deliver a personalized summary to the player. It aggregates the actions taken by the player since joining the game, tracks the battles that the player has been part of, reports win-loss and leveling statistics, and controls the personal messages between players. This controller also handles the profile lookup feature so that logins are searchable.

When players view the lobby screen, the LobbyController takes over. It continuously gathers information about the battles and matchups that the player is part of, as well as the battles that are available to join. The logic is then handed off to different controllers based on the battle he or she wants to view. If a new battle is created or if the player returns to a process of initializing a battle, then the request is sent to the MatchupController. Otherwise, if the player wishes to continue an ongoing battle, then the request is forwarded to the BattleController.

As the name suggests, the MatchupController handles the process of matching two players up so that they can duel. As each request comes in, the controller determines the next action that needs to be taken by the player, and renders the corresponding form or a wait page letting the player know that the opponent still needs to make a choice. On the arena choice page, the controller

gathers the list of available arenas to choose from, and on the deck choice page, it gathers the list of decks that are available. However, if the player wishes to view details about a specific arena or deck, the control is then handed off to the ArenaController or DeckController respectively. By being able to pass this control back and forth, the code for related game processes are localized, but can still be shared between processes without repetition.

Once the battle starts, the flow is handed off to the BattleController. To fit all the game logic into this controller would have made the file very hard to maintain, since there are many views and actions to handle. Therefore, much of the action processing is pushed to a BattleLogic module and the feedback processing is pushed to a BattleStatus module. The code that remains in the controller is used to decide which functions to call from the modules, to interpret the results, and then to choose the appropriate elements to update in the views.

Since much of the battle dynamics is handled in BattleLogic, it makes sense to discuss some of its functionality and structure here. There are six important actions handled by BattleLogic: setting up the battle from a matchup, drawing a card from a player's deck, playing a card from a player's hand, processing an attack, moving the phase and round of battle forward, and ending the battle. Setting up the battle requires copying over the information from a Matchup object, creating BattleCards from the chosen decks, shuffling those cards, and then dealing out the initial hands. Drawing and playing a card basically just changes the location variable of a BattleCard, although there is also a check that the card being move doesn't cause a location to exceed the maximum capacity.

To process an attack, the attackers, targets, and moves are first looked up from a BattleRound object. The weather conditions are then incorporated to determine the damage that needs to be inflicted to each target. Finally, the score for that round gets calculated and creatures with negative health are sent to the sideline. In order to handle switching phases and rounds, a check is performed to determine whether actions were recorded for players during a phase and if both players have acted, then the battle is progressed. Once the maximum round is reached, then BattleLogic is told to handle the ending sequence, which involves determining the winner and updating the players' stats, deleting the BattleCards used for the battle, and then releasing the players from the battle.

To provide direction and feedback for the players, status messages are constructed by another module, BattleStatus. BattleStatus analyzes a BattleRound and constructs a summary of the events that occurred in a readable format. Separating this text from code not only allows us to easily find, update, and add messages as necessary, but also simplifies the process of translating text to adapt to different worldwide locations.

Another important aspect of the battle is its integration with weather data. When users wish to delve into hourly weather statistics or view weather maps in order to make a forecast, a request is sent to the WeatherController to gather information about an arena on a specific date. The data and maps are preprocessed from reputable weather sources and stored on the Weatherlings server beforehand, so information flow is not restrained by capacity or other problems of external servers during game play.

Lastly, there is an AdminController that handles requests from an administrator of the game. These requests involve gathering a summary of the current state of the game (the players that are logged on, the battles that are being fought, etc.), reviewing the historical actions performed by each player (from battle histories, notices, log files, etc.), and possibly generating reports to analyze a player's progress.

The Views (V)

Once the controllers gather the appropriate information from the underlying models, the data is used to create the appropriate views to return to the client's browser.

General Guidelines

The first major aspect that was considered was the amount of screen real estate to use. Although many modern mobile browsers have the capability to resize websites to fit on the screen, Virus showed that the process of zooming in and out of specific sections can be quite cumbersome. It forces users to perform multiple clicks just to navigate a single page and users may get lost if they are zoomed in too far. As a result, after testing on iPhone and Android devices, Weatherlings was chosen to be constrained to 300 pixels in width to avoid unnecessary zooming. The height of the page is unconstrained, although by practice, most pages were designed with the vital information fitting on the initial screen and the full content taking up no more than three screens.

Speed is always an issue of concern when developing applications, since long wait times can severely degrade the user experience and cause frustration. By constraining the amount of

information that was displayed on each page, the amount of data that needed to be transferred over the Internet and rendered by the browser was also reduced. As a result, the total time between when a request is made and when the next action can take place is minimized, and several play tests have shown that the current game pace is tolerable. Weatherlings makes heavy use of the AJAX enabled browsers to retrieve information and to perform updates *only* on sections of the page that change, which further reduces the data that needs to be transferred.

Although the incorporation of AJAX and other JavaScript functionality into web applications helped relieve some of the speed and processing issues, they also added a few complications of their own. Mobile browsers only support a subset of the JavaScript available to desktop browsers, and different browsers have different levels of support, so in order to accommodate as many of them as possible, a lot of the scripting needed to be kept to the basic levels. Notably, animation and textual effects resulted in rendering and timing issues, so many of the comprehensive third-party libraries that add stunning visual effects could not be used in the application. As a concrete example, an early idea was to animate the attack sequence by moving Weatherling images around the screen, but because of timing and positioning problems on mobile devices, the idea was later dropped. These compatibility issues extend beyond JavaScript— some CSS-stylings of DOM elements are disabled or rendered differently on mobile browsers, and animated GIFs have been known to be dysfunctional at times.

Screen Design

Using our experience from Virus and keeping screen size, speed, and compatibility in mind, the Weatherlings game was designed in the following way:

Figure 4.4 Weatherlings' introductory screens: (a) the login screen and (b) the homepage

Figure 4.4a shows the initial screen that is displayed when users arrive at the Weatherlings site. The minimalistic interface serves two purposes: to allow registered users to log in and new users to sign up for an account. Once the user is identified by the system, he or she is redirected to the screen shown in Figure 4.4b. This is the player's homepage, which serves as a portal to different sections of the site. It has a very simple layout as well, drawing the user's attention to four large, color-coded buttons for navigation. The idea was to emulate the "main menu" or "world map" screen found in many battling and RPG type games.

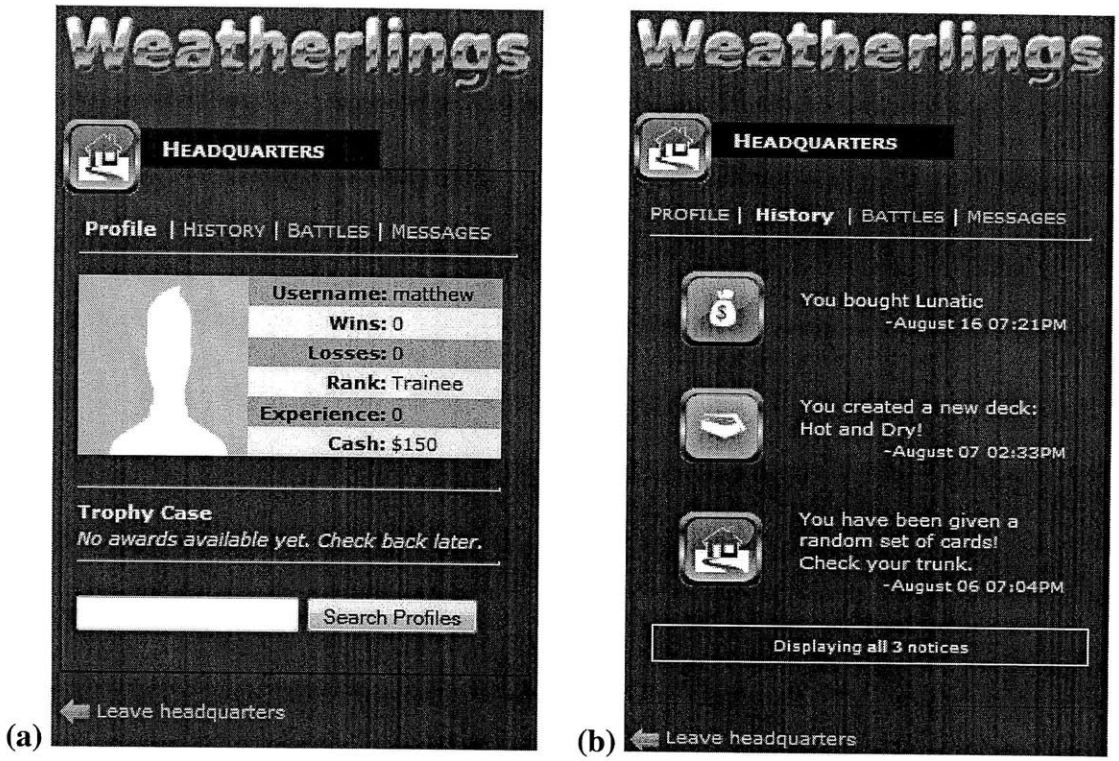


Figure 4.5 Pages a player has access to in their headquarters: (a) public profile and (b) notice list.

Clicking on the Headquarters button brings players to several panels where they can learn about their status in the game. The first panel shows in Figure 4.5a is where players can find their public profile, which is information that can be viewed by anyone in the game. The data includes basic win-loss and rank statistics, an avatar image, as well as a "trophy case" that display awards and medals won for performing special tasks or winning tournaments in the game. To lookup other profiles, players can search for a login using the search bar at the bottom, which includes a "suggested completion" feature.

In addition to the public profile panel, there are several other useful panels as well. One panel, shown in Figure 4.5b, contains historical information about important actions taken by the player, such as creating a deck or buying a card. Another panel provides a summary of all the

battles that the player has taken part of, including the final scores and the date that they took place. The final messaging panel, which utilizes an email-like interface with inbox, outbox, and compose tabs, was created to allow players to communicate with each other privately when they are not face-to-face. One possible use case of this messaging system would be players directly challenging other players to a match and setting a time when they would both be online.

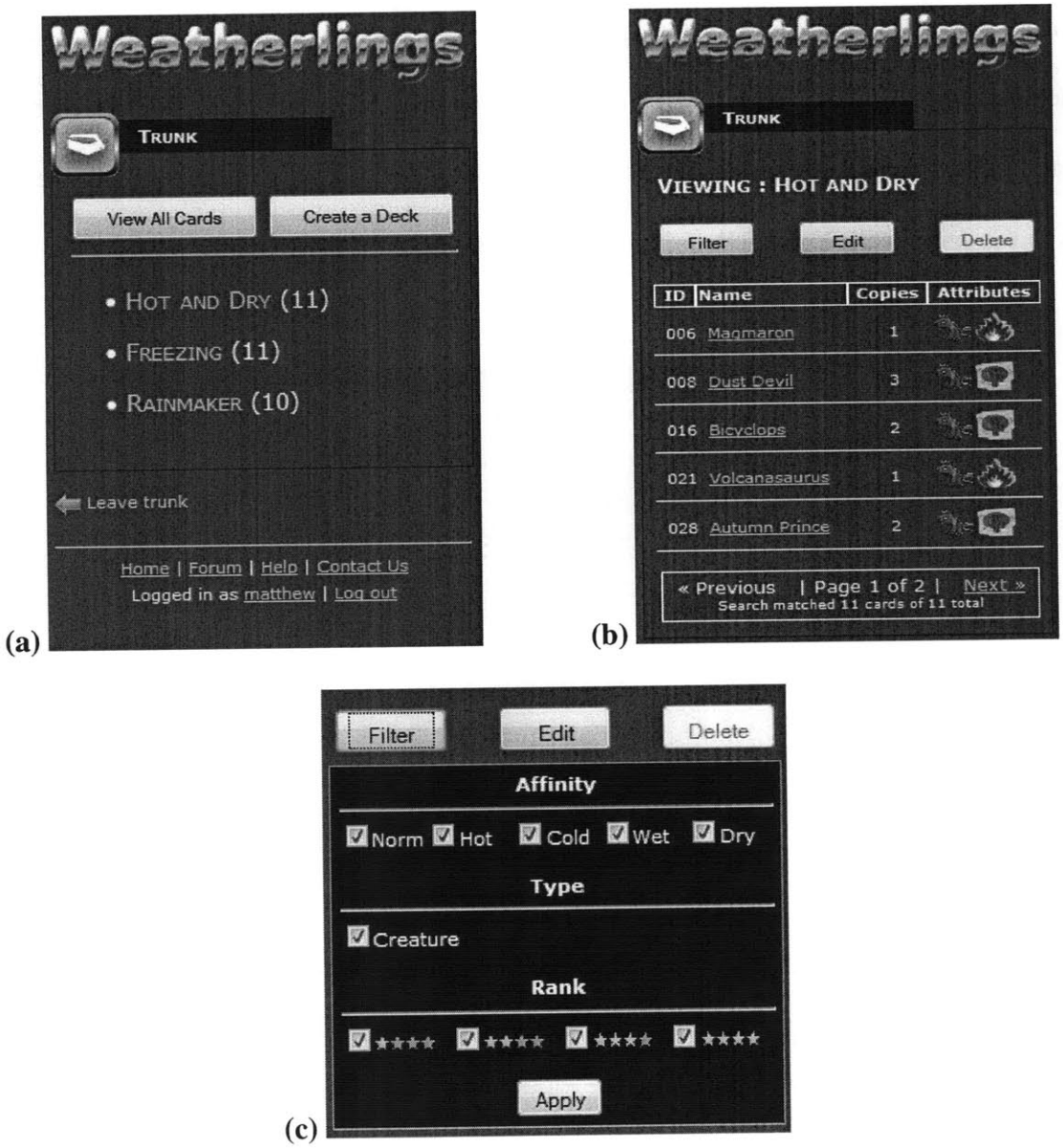


Figure 4.6 The trunk activities that a player can access: (a) the deck listings, (b) the card listings, and (c) the filter menu.

When users navigate to the Trunk, they are initially shown a screen similar to Figure 4.6a. All the decks that they have created are listed, as well as a button to view all cards that belong to them and a button to create a new deck. Choosing to view a deck or to view all cards brings them to a card listing page shown in Figure 4.6b. Important information about each card is displayed on this page, making use of icons rather than text to save space. An initially hidden filter menu is available too (shown in Figure 4.6c), which allows users to browse a subset of cards, like those with a specific affinity or rank. Choosing to create a new deck or modify an existing deck brings up the deck editing form, which looks similar to the card listing page, except there are press-able plus and minus symbols to add and remove cards from a deck. A tally of the total number of cards in the deck is also included at the bottom of the page.

Figure 4.7 An example of a Weatherling card

To gather more detailed information about a card, players can click on a card name, which brings up a screen shown in Figure 4.7. Rather than just showing an image of the creature and its stats

in a table-like format, the page was designed specifically to look like a real-life playing card in order to support the metaphor that players are managing a collection of cards. The information displayed on a card includes the starting HP of the Weatherling and the moves (along with their damage conditions) that the Weatherling can perform. The card screen is rendered as an overlay, since after viewing a card's information, players usually want to return to what they were doing. So instead of re-rendering the previous page, the visibility of the overlay is just set to be hidden when the player hits close, which offers a significant speedup from sending another HTTP request.

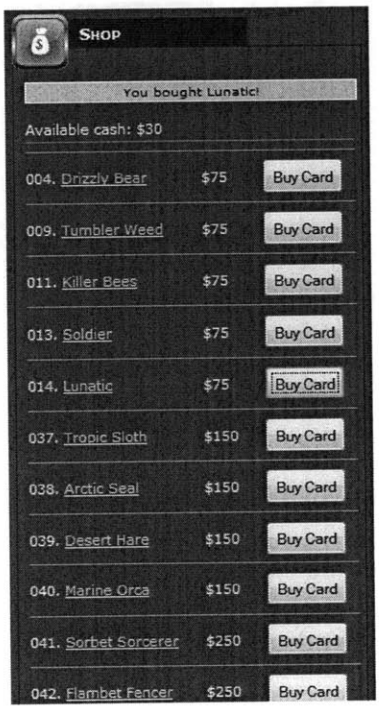


Figure 4.8 A list of card available to purchase at the shop.

The screen shown in Figure 4.8 is an example of what a player might see when they visit the Shop. There is a quick buy button next to every card for expert users who know the cards by name, but like the deck view, clicking on a card's name will bring up the detailed card view

overlay. The buy option is added to the overlay to avoid having to search for the card from the shop list again. When a purchase is made, a notice is displayed across the top of the shop and the player's available cash is updated.

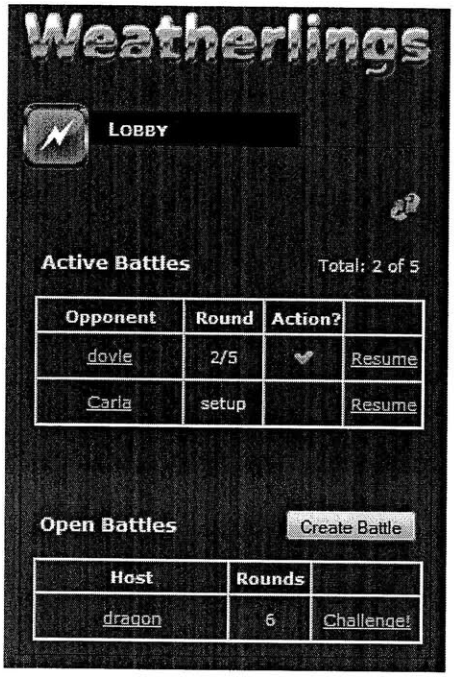


Figure 4.9 A list of battles a player is part of or can join.

The lobby screen in Figure 4.9 is a dynamic page that automatically refreshes as new battles are created and states of existing battles are changed. The page is divided into two major sections. The top section lists all the battles that the player is part of, including the ones that are currently being set up. An important feature of this section is that each battle indicates whether or not there is an action available to the player, and how far the battle has progressed. When part of multiple battles, this page can be very useful to players since it summarizes all their battles and directs them to the ones where their input is needed. The bottom section lists all the players that waiting for challengers and the number of rounds that they want to battle for. If players are looking for

games, they can either pick one out from the list or create a new battle by pressing the "Create Battle" button. The current system allows players to be part of five battles at one time, and once that limit is reached, the options to join or to create a battle are disabled.

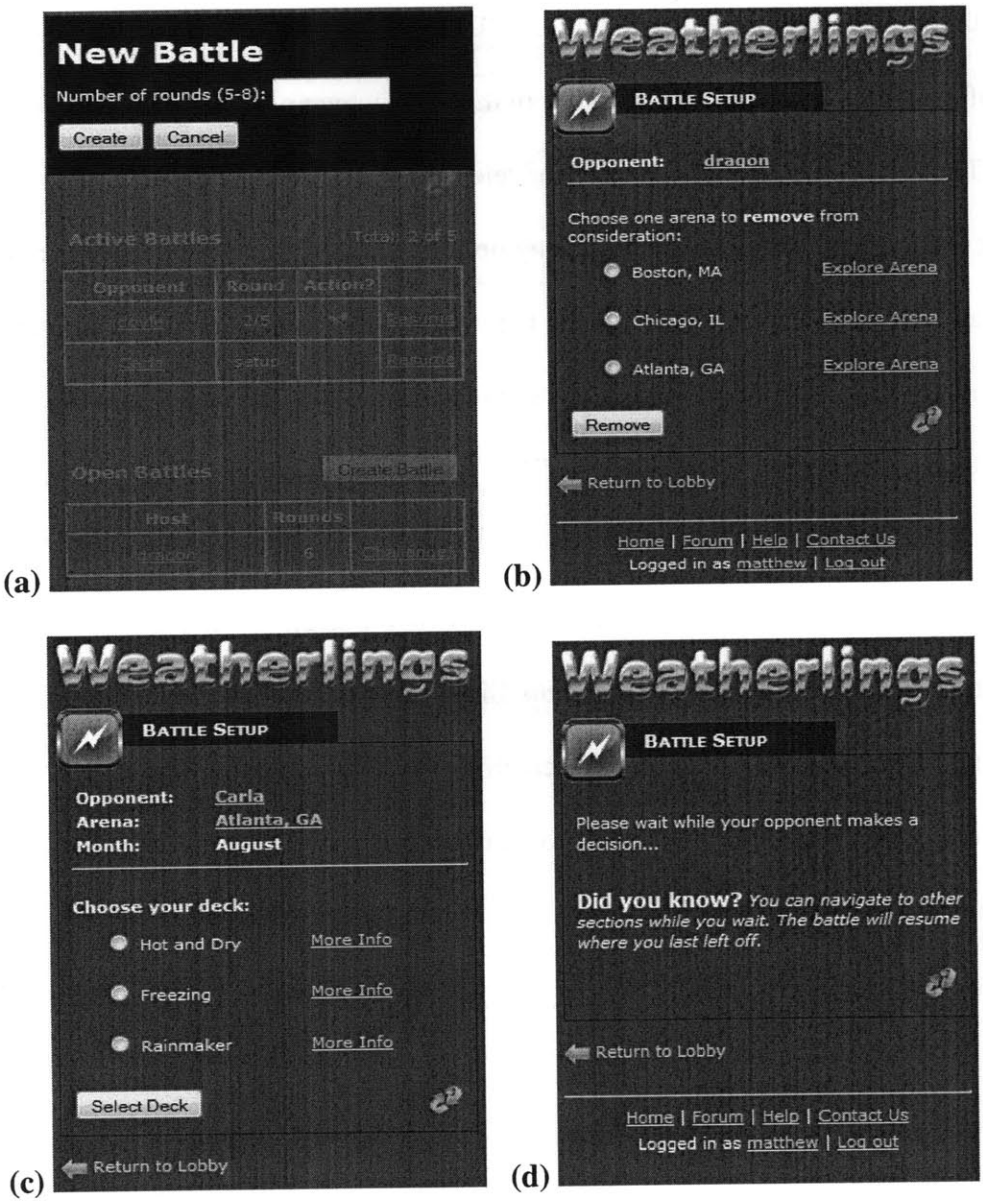


Figure 4.10 The sequence of steps to set up a battle: (a) select the number of rounds, (b) choose an arena to eliminate, (c) choose a deck, (d) wait for an opponent to choose.

In order to set up a battle, players are led through a sequence of screens shown in Figure 4.10. The first screen appears as a popup and asks the host to input the number of rounds to battle for. Once the battle is created, the host is returned to the lobby page to wait for a challenger. When another player accepts the challenge, the setup can advance to the next phase, which is to choose the arena that they would prefer *not* to battle in. Clicking on the arena name will show annual climate information for a particular location in the U.S., which players can use to help make their decision. The final screen shows the page for selecting the deck for the battle, and similar to other pages, clicking the deck name will bring up information about that deck. Across the top of the arena and deck choice screens is a panel that reviews the choices made thus far and can be helpful since selections from previous steps may factor into the decision that needs to be made at the current step.

Figure 4.10d shows a placeholder screen that is shown to players while they wait for their opponent to make a decision about a matchup. Since there is nothing for the player to do while waiting, one feature that was added to the screen was a "Did you know?" message, which randomly displays a useful tidbit, such as how the battling system works or how to interpret a weather map.

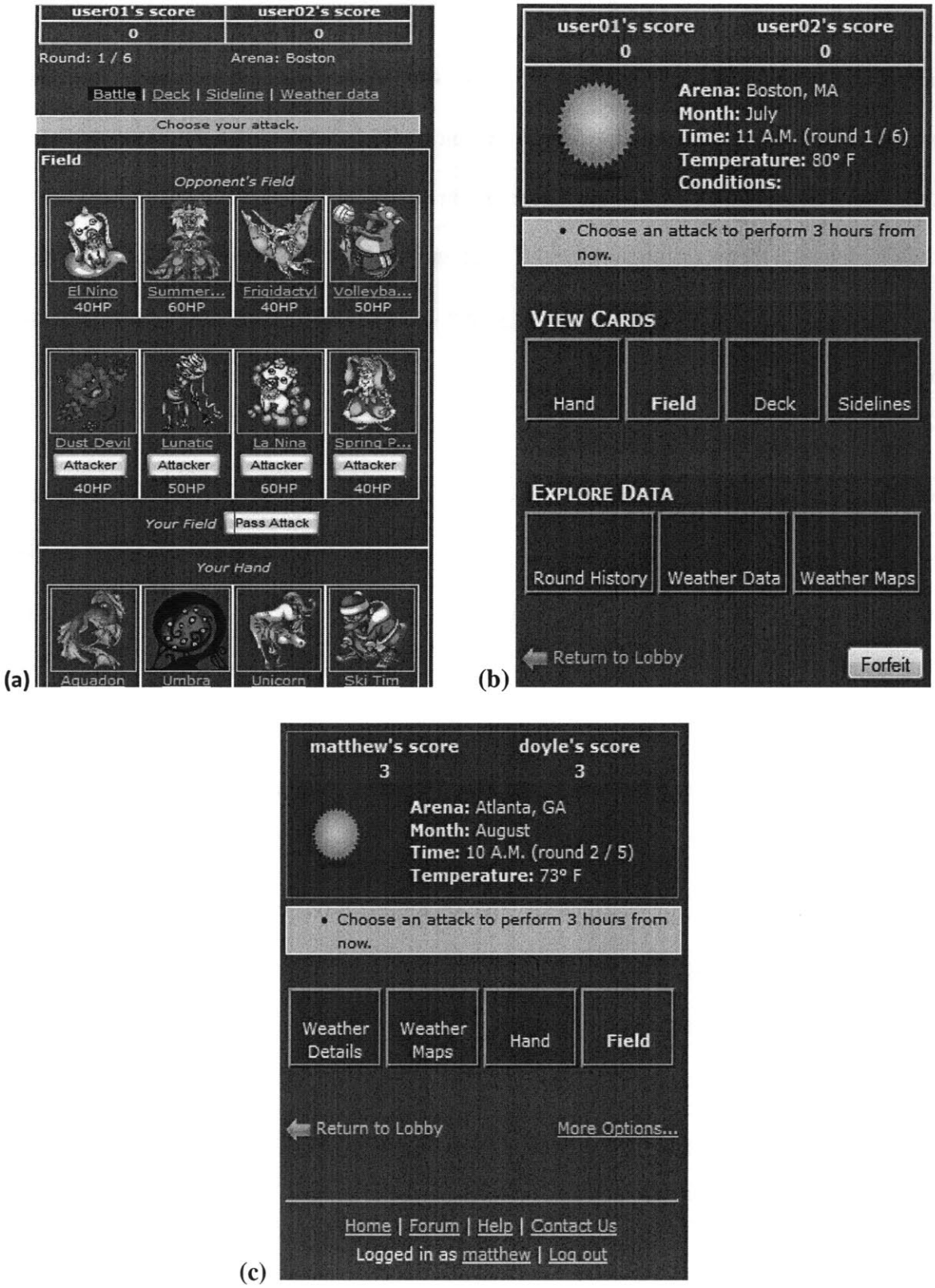


Figure 4.10 Iterative redesigns of the battle screen: (a) full view, (b) menu screen, and (c) hidden options.

A lot of time was spent developing, testing, and revising the battle interface since battling is the primary main activity in Weatherlings and it is the section that players are expected to use the most. Figure 4.10a was the initial design of the battle screen, where the hand, field, and menu options were all displayed on a single screen. Figure 4.10b was the second design we came up with, which uses a menu system to hide the hand and field until called upon. Figure 4.10c shows the current design, which also uses a menu system, but moves some of the options to a secondary screen instead.

There are merits to each of the designs and tradeoffs were made when deciding to switch from one version to the next. The initial version reduced the number of links that needed to be clicked since the hand and field cards were always in plain sight. Actions such as checking the field to decide what to play from your hand only required a short vertical scroll, whereas in the menu version, it would require alternating visits to the field, menu, and hand screens. However, there were two main reasons why the switch was made. First, the amount of information being displayed at one time can be overwhelming for the user, which makes knowing which section to focus on next a hard task. This is especially true on mobile browsers since the summary information is up at the top, the attack buttons are in the middle, and the play and draw buttons are down at the bottom. Users would need to constantly scroll up and down to check where the next action takes place. Second, this type of view detracts from the importance of the weather information, since the busy nature of the page and the card graphics tend to overshadow the text-based submenu link to the weather maps and weather data.

To remedy the drawbacks of the first design, the second design was created to emphasize the actions that were available to the player as well as the resources that can be used to make well-informed decisions. The score box was expanded to include summary information to provide a good overview of the current state of the battle, but the hand and field screen were hidden from view until an action required the user to view them, such as drawing or playing a card from the hand. In order to minimize the time needed to swap between views, the menu, hand, and field screens are only rendered once, and then are always being updated in the background. Clicking a link only toggles the visibility of the desired section and hides the other sections, so flipping to these screens usually feels instantaneous to the user. Also, in order to direct the user to the appropriate screen where an action can be performed, the button corresponding to the action's location is highlighted on the main menu.

The major changes between the second and third versions of the UI basically deal with how the menu buttons are arranged. First, instead of separating buttons by their function (i.e. "view cards", "explore data"), they are grouped by how often they are used. Checking the weather data, the hand cards, and the field cards are necessary actions for every round so they are displayed up front, but exploring the history of the battle, and checking the deck and sideline cards are less used features. These features were moved to a secondary menu page that can be accessed via a link from the main menu. Second, in order to further promote the importance of using the weather maps and data to forecast, the weather buttons were moved to the very front of the menu.

There is also one hidden, but important change to the battle mechanism in the third version: an additional timestamp for the client-side information. Since the client is constantly polling the server for information at short intervals while viewable updates may not occur for a long time (i.e. the opponent is busy attending class and doesn't make a move for an hour), repeatedly transferring and rendering the same content would be useless. Instead, the client's request is sent with an additional timestamp parameter, so that the server does not respond unless the timestamp on the server has been updated.

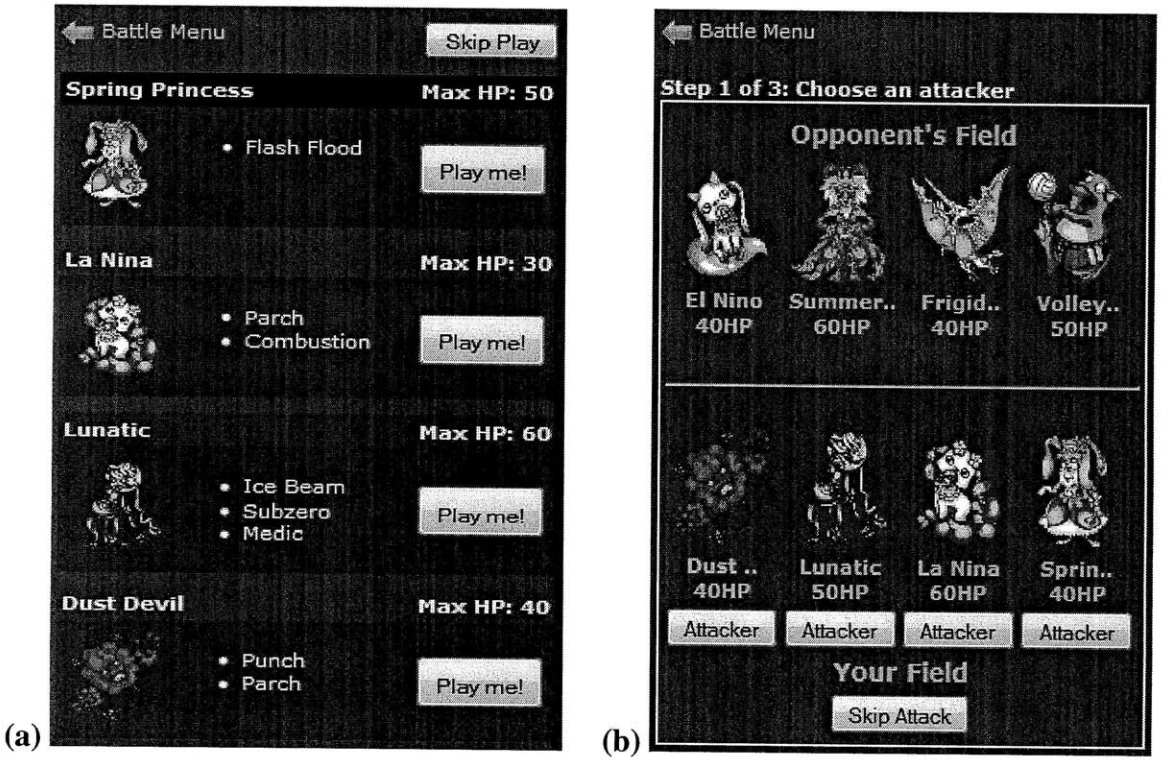


Figure 4.12 Important battle screens: (a) the hand and (b) the field.

The hand and field screens are shown in Figures 4.12a and 4.12b respectively, and have changed a bit since the original design. For the hand screen, instead of just depicting cards as square images aligned in a row, details such as its maximum hit points and its move list were added,

since these are major factors that players consider when playing a card. Although a bit more subtle, the color of the headings also signifies the affinity of the creature. For the field screen, borders were removed from the creature sprites to make them look livelier and it reduced the clutter on the screen. The field that the Weatherlings are standing on was also spruced up with a bit of color to make it seem as if the creatures were battling on a dirt turf.

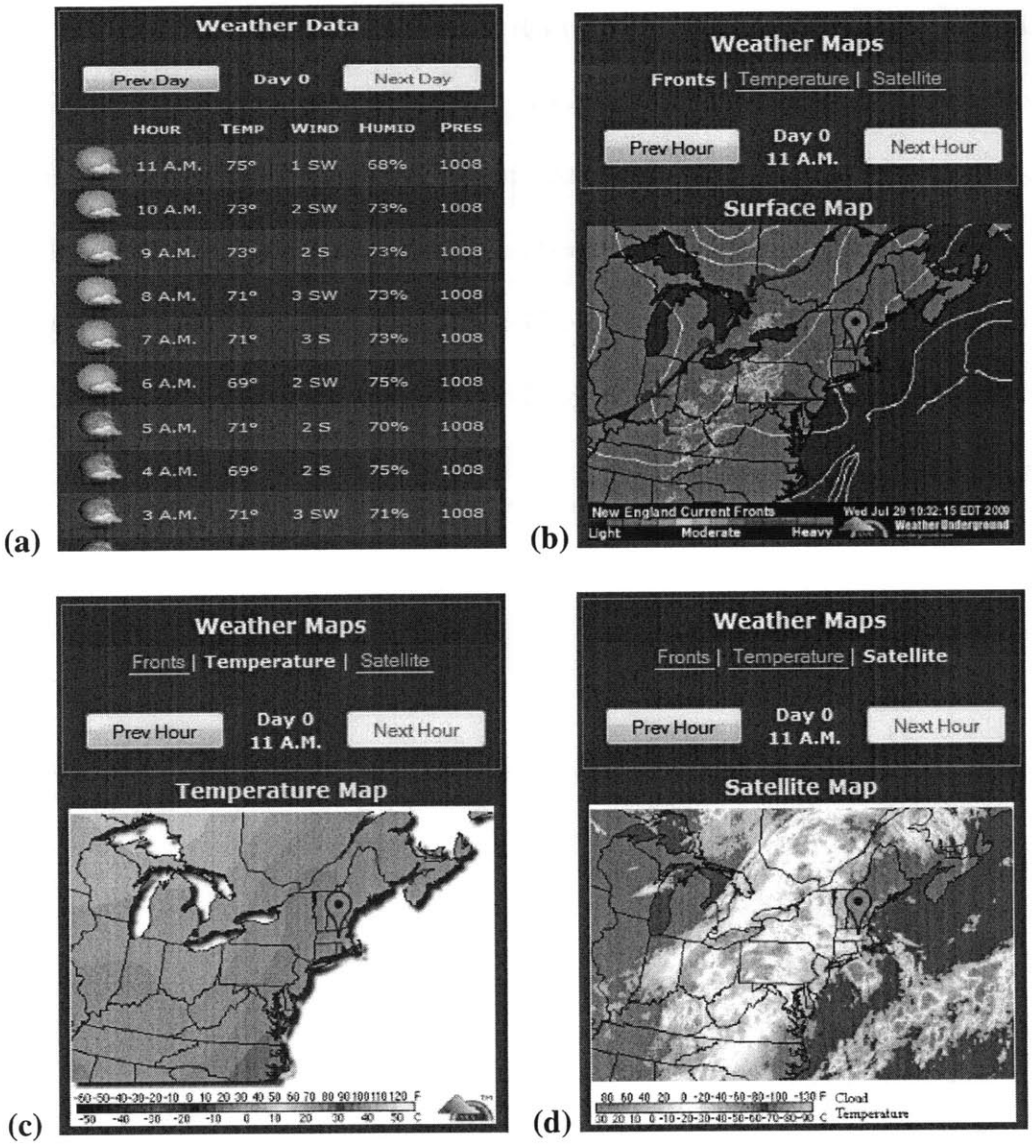


Figure 4.13 Various weather data available during a battle: (a) station data, (b) fronts and pressure map, (c) temperature map, and (d) satellite map

Another user interface challenge was the presentation of weather data and weather maps, as shown in Figure 4.13. On the map view, players view a regional map of the arena they are battling in and can choose between a fronts map, a temperature map, and a satellite map, as well as an hour of day to view (up to two days before the battle starts). The chart types and time periods are all synchronized, so switching between map types keeps the hour constant while switching between hours keeps the type constant. The current design of the weather data page lists the hourly numbers for an entire day, which breaks the interface goal of reducing the amount of vertical scrolling. An alternative design would have further paginated the table into groups of x hours (i.e. 8 hours), but user testing has shown that finding trends is an easier task when both the before and after data point are on the same screen, so breaking apart the data is actually disruptive to the forecasting process. Daily listings seem to be a natural breakpoint in the data and twenty-four entries seems to be a good balance between ease of use and excessive scrolling.

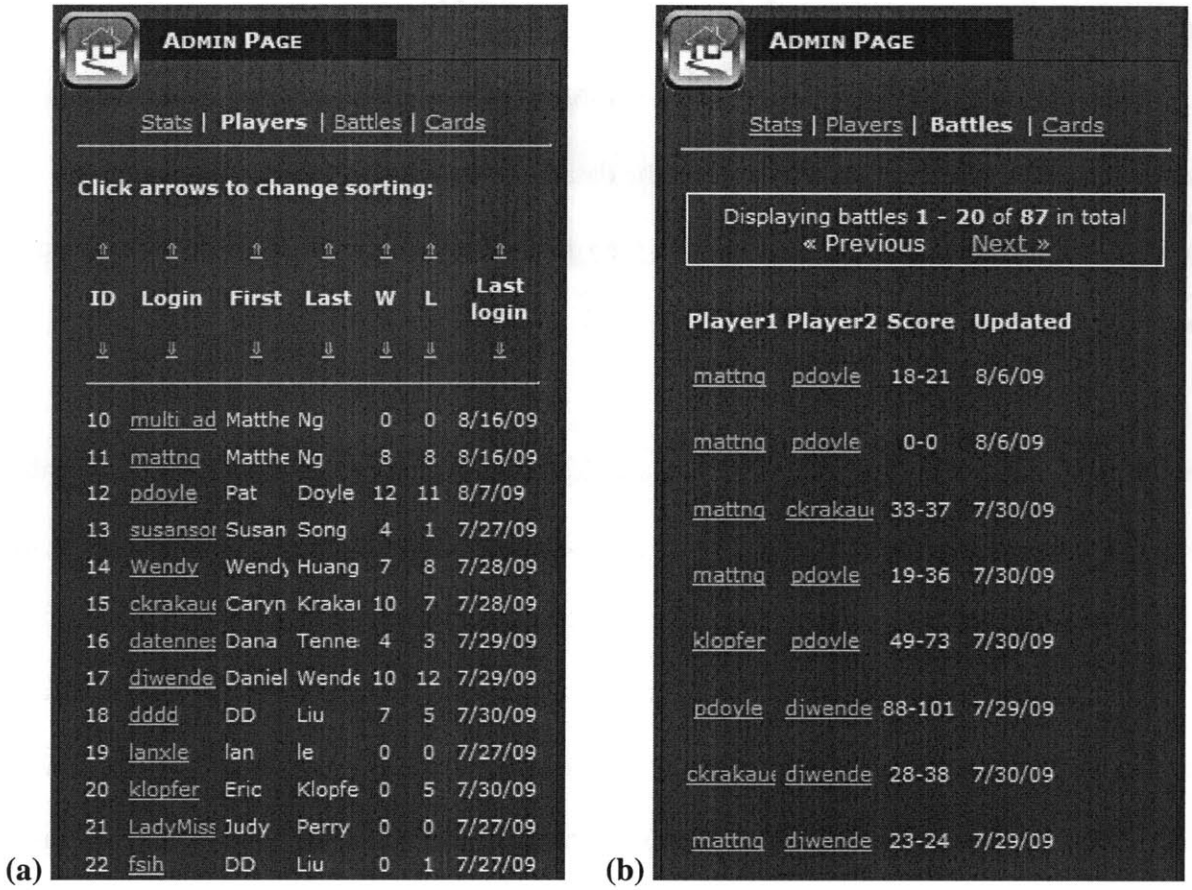


Figure 4.14 The administrator toolbox for Weatherlings

The administrator interface works a bit differently than the other screens in Weatherlings, in that it is designed to be used on a *desktop* rather than a mobile device. It is still viewable on mobile browsers, but the idea is that if administrators are generating reports and summaries to follow along in the game, they would most likely be working at a desk than on the go. Looking up information can still be performed on the mobile device, but creating Excel spreadsheets and charts would be confined to desktop usage. Figure 4.14 is an example of what an administrator might see when they view details about players and results of various battles.

Other Modules

In order to run Weatherlings smoothly, several other Ruby scripts were written, mostly with the task of generating the static data to store in the database. This data comes from a variety of sources, from parsing XML files to scraping web data to querying archives of historical map images.

The format which we chose to store Card, CardMove, and Arena data is in an XML (extensible markup language) file since they are easy to generate and configure for non-technical users, easy to parse (if well-formatted), and as the name suggests, easy to extend in case these models need to be changed in the future. Instead of writing the parser from scratch, since there were already many existing XML parsers written in Ruby, a well supported third-party library, Hpricot, was chosen. The Hpricot library facilitates the task of navigating through tags and reading the tag's attributes and values. The main work left then derives from transforming the user language text in the XML files into system language encodings that can be stored in the database. As a simple example, in the XML file, users should be able to define a card's moves by writing "Punch" and "Kick". In the database, however, these moves need to be stored by their primary ids, so an additional lookup needs to be performed. To handle these various types of preprocessing tasks, the basic design pattern involves defining a hash where the keys of the hash are the tag names and their values are the *procedures* (to be precise, a Ruby Proc object). The procedures handle evaluating the tag's values, processing them, and then returning the transformed value. Examples of the XML snippets are available in Appendix B.

Another script that was written involves downloading images of weather maps from an online archive and storing it on the Weatherlings' server. Although capturing an image is a straightforward procedure, additional image processing techniques needed to be applied so that the image would be suitable to be displayed on a mobile device. In order to accomplish this, a well supported Ruby library, RMagick, was used. RMagick provides an API to crop, trim, resize, and convert images to other formats. By using this tool and pixel calculations, hundreds of North America weather maps were transformed into regional maps that focused in on particular Arenas.

To capture raw historical information from weather stations across the U.S., a script was written to scrape data off an external website that provides the information for free, but was not in a format that was usable out of the box. The script makes heavy use of the Hpricot library again, since the library can parse both HTML and XML files, but this time instead of reading in tags, it relied on the consistent structure of the webpage to assign the data values to their corresponding fields. Since maps and data need to be synchronized in the game (that is, if players are able to view the raw station data, then they should be able to see a visualization as a map as well), both of these scripts were combined into a parent script, process_weather_info.rb. This script takes a starting date and duration as parameters, and downloads the data and maps for all the arenas that are currently in the database. An optional type parameter is accepted as well ('M' for map and 'D' for data), in which case only one of the scripts is run.

Ruby modules were also used to encapsulate game constants and file paths so that the parameters are centrally located and easily changeable. This design allows the parameters to be accessed in any model, view, or controller without having to access the database. The GameConstants

module stores items like the initial cash that each player receives, the amount of experience necessary to level up, and the number of points you score for different actions in a battle. The module ImageResolver does something similar in that it stores the directory path to a particular image resource, but it also takes an object as a parameter in order to fully resolve the filename.

C. User Testing

To date, Weatherlings was put through four major play tests, each of which provided useful feedback on the game features, offered suggestions for future improvements to the game, and uncovered bugs that needed to be fixed in order for the game to work properly. In this section, I will talk about the specific goals associated with each play test and the major findings and changes to the system that were made based on the feedback that was received.

The first play test was held relatively early in the development cycle of Weatherlings. Once a functional deck management and battle system was set up, along with a small sample set of weather data, an informal play test was executed in a classroom setting. The goals of this play test were to see how the Weatherlings system responded to having multiple players logged in simultaneously, and to get feedback on the playability of the game, such as the battle flow and the user interface. The testers were undergraduate and graduate students at MIT who were taking a class on game development, so they had a relatively high level understanding of technology. The students were first given a brief overview on how card dueling games worked in general, but not specifically on how Weatherlings worked. Then they were given a worksheet (see Appendix C) with questions to fill in by the end of the play test. Next, they were asked to sign up for an account and play the game on desktop browsers, since the mobile aspects were not fully

functional back then. After about an hour, the testers were given some time to fill in worksheet given to them earlier. For the last half hour, a group discussion was held to talk about any additional feedback or suggestions they had.

The play test ran relatively smoothly, as players were able to create decks and run through a couple of battles, which provided a positive indication that Weatherlings was headed on the right track. Two minor bugs were uncovered by the play test, but those issues were resolved soon thereafter. The major takeaways from the feedback forms and group discussion include the need for the ability to participate in multiple battles, since a single battle did not provide enough action, and the need for more feedback and direction (tutorials and practice battles were explicitly suggested).

After incorporating a lot of the feedback from the first play test and fleshing out the game features a bit more, a second play test was held about two months later. The goals of this play test were to continue to assess the usability of the system, and more importantly, to get feedback on the experience of Weatherlings when played on a mobile device. The testers were researchers from the TEP lab at MIT, who have had experience designing and creating educational games for kids, but have various levels of expertise working with mobile devices. Similar to the last play test, the participants were only introduced to the idea of a card dueling game, and then were asked to play the game for about an hour. Testers were using a variety of mobile devices including iPhones, an iTouch, Androids, and HP Travel Companions (with Windows Mobile 5 installed), as well as a few desktops and laptops too. During game play, questions and comments

were noted down and responded to, but the major feedback came from the group discussion at the end of the hour.

The feedback from this session was especially important, since this was the first beta version of Weatherlings that was tested by non-group members (the version for the previous play test was more of a prototype, similar to what Virus provided). The following lists some of the major feedback and suggestions that were generated by the play test:

- Defining where arenas are located on a map is essential for forecasting
- Mapping out sequences of actions for the user will make processes such as attacking easier to understand. A possibility is to create a step-by-step wizard to guide the user.
- Accounting for different time zones was a bug that needed to be fixed, since the weather data was given in UTC time rather than local time. This discrepancy caused unusual weather patterns such as "cooler days" and "hotter nights".
- Balancing cards by adjusting HP values and the moves list, since some of the more powerful cards renders other cards useless. Another possibility is to make powerful cards much harder to obtain.
- Providing a last round warning message will make the battle ending less abrupt and less surprising, since in the current system, round changes are not apparent enough.

After the play test, the entire feedback list was evaluated and was used to prioritize the next features that needed to be developed.

Approximately two weeks later, another lab wide play test was held, but this time, the main purpose was to capture how people played the game over a period of four days. Most of the

participants were part of the previous play test, so they had some familiarity with the system, but several changes had been made since then, most notably the overhaul of the battle screen UI (as described in the previous section). Testers were told to use the system however they see fit and to battle during free minutes of their day. Thirteen players signed up for the play test and were given the same variety of mobile devices as the previous play test, but they were also allowed to play on any other platforms that were available to them. Questions and comments were handled on an individual basis during the four day period and a group debrief session was held on the last day of the play test.

Although somewhat unconventional, a few changes were made to the system in the middle of the play test based on the continuous feedback that was received (the majority of the application, though, was kept consistent over the time period to reduce the probability of introducing bugs). For example, the scoring system had originally only factored in the number of creatures that were sidelined, but this method resulted in a lot of tied games. Players complained about this type of scoring because it made battles less exciting and each action seem less important. Therefore, between the first and second day, the system was modified to take into account the damage inflicted by every move, which added minor point values to the score. Battles were now won and lost by smaller margins, but this increased the importance and thrill of selecting the best move in every round.

The debrief session provided many more comments, suggestions, and feature requests. A summary of the major takeaways are listed here:

- The effect of character affinities on damage calculation should be explicitly defined for the player.
- The current daily weather maps are insufficient to forecast precipitation. The raw data is a decent indicator for rainfall, but predicting the exact hour is a coin toss.
- A weather tutorial is needed to describe how to interpret maps and how to use maps and data to forecast conditions.
- Report results should be in the user's language. That is, numerical formulas are cryptic whereas verbal descriptions are preferred.
- A feature that indicates which players are online will be useful, since direct messaging can then be used to setup a new battle

The play test turned out to be very successful, producing many useful suggestions and criticisms, as well as its fair share of praises for the Weatherlings game. Half of the players were able to achieve the highest "Master" rank and over sixty battles were fought on the system, which showed a successful backend implementation. Players considered weather conditions to be a key aspect of battles and knew they needed to forecast to do well, which showed success as an educational design, and thought the game was fun and addicting to play, which showed success on the design of the game mechanics. Following these results, development on Weatherlings forged ahead to prepare for the next play test.

With the application becoming more mature, user-focused studies were held more frequently and the incremental changes to the system became more closely tied in with the feedback from these sessions. About a week after the multiple day play test, another play test was held, but this time

the participants were more aligned with the game's target audience – kids from ranging from 10 to 12 years old. Ten kids participated in this study and they were all given Android devices to use. Background information for this play test was a bit more detailed than previous studies, as the kids were given a 10-15 minute onscreen walkthrough of Weatherlings before they started playing. They were then asked to play for about 45 minutes, before briefly pausing for questions, and then resumed for another 30 minutes. After that, they individually filled out a post-play survey before recombining for a group feedback session. The table below summarizes the numerical results from the pre- and post- surveys that were given out (full documents are included in Appendix C).

	A lot	A little	None
Like videogames	9	1	0
Like computers	9	1	0
Played card-dueling games	1	3	6
Like battle games	1	6	3
Like Weatherlings	9	1	0
Like character design	4	6	0
Fronts map use	1	5	2
Temperature use	1	9	0
Precipitation map use	2	7	1
Station data use	6	0	2

Number of testers: 10 (7 boys and 3 girls)

Figure 4.15 Pre- and post- survey results from the kid play test

The play test with the younger audience brought out unexpected positives and negatives with the system's usability. Adapting to the interface of a mobile device is usually the first step for new

users, but this task proved especially difficult for some of the kids in the group. One theory attributes this difficulty to a general lack of patience. In a particular example, one user mistakenly brought up the bookmarks screen on the Android browser, but instead of examining the screen for a close button, he tried to revert to the previous screen by tapping on random points on the page. These actions brought up more alerts and screens, and eventually he stated that he did not know what to do next and needed help. Another instance when this caused problems was when one user pressed a link on the lobby page, but the game screen did not change immediately. A progress bar did appear at the bottom of the page, but instead of checking for feedback, the user decided to rapidly press the button again and again. These subsequent presses were less precise and because of this, eventually one of the presses ended up triggering links to other pages, which invalidated the original action. One possible solution, although not perfect, is to give a device usage demonstration to go along with the game introduction, in hopes that the tutorial will teach users about the different feedback mechanisms of the device and how to resolve common errors.

One the other end of the spectrum, an unexpected positive feature was the participants' interest in using the direct messaging system. In the post-survey, kids wrote that they liked messaging because "you could talk to people" and "you can set up times to battle your friends". Making this feature more prominent may be an easy way to make the game more fun, since many users did not know that the system existed until it was mentioned in the break between play sessions. Additional feature requests proposed in the group discussion include:

- A way to visualize an attack
- A system overview of all the game's features

- A storyline to the game
- Better card graphics to distinguish between creatures
- A virtual world to explore
- Hidden items and un-lockable items add a purpose to the game
- Equip-able cards such as swords and shields
- A buddy list to quickly message and keep track of friends

Common threads that can be seen from these requests are that kids want to explore a bigger world outside of the battle system where they can interact with their friends and that they want to see more visual effects. Although these are interesting features to add, they might lie outside the purpose of an educational game.

D. Improvements

At the time of writing this paper, Weatherlings is still undergoing development, so features are added daily and the UI is constantly being improved upon. The current application has a solid foundation and has proved to be playable in its current form, but there are still many ways to extend the game. In following list, I will describe some of the changes and improvements that I think are the most important to allow the system to grow and to enhance the user experience (ordered by decreasing levels of urgency).

1. Supply weather tutorials – Education is a major goal in every game in the UbiqGames project, so making sure that players come out of the game knowing more about the subject matter than when they first started is important. In Weatherlings, players may initially start off by guessing future weather just based on the current time point. Keen players may then make use of the given historical data and maps in order to perform

some interpolation and pattern matching. However, learning to understand weather systems and the cause and effects of different weather attributes is a difficult task without further instruction, so this is the role that the weather tutorials should try to fill.

However, simply including a tutorial as a "mini textbook" does not integrate well in the spirit of a game, and can even deter users from exploring this section. Instead, finding ways to incorporate the education content as a game mechanic is a much more effective idea. For example, one thought is to make a background story for a particular Weatherling who is in need of a correct forecasting. By learning to read and use weather maps, players can help the Weatherling accomplish some important task and earn some type of reward for doing so, such as unlocking the creature card at the shop, or earning additional money, or maybe receiving a medal to display at the headquarters. In any case, the learning aspects and game play experience should be indistinguishable to the player.

2. Balance/Create cards, moves, and types – The current game supports roughly forty cards, fifteen moves, and only one card type, creatures. In order to keep the game interesting over a multiple day period, more cards and moves need to be available so there can be more deck permutations and more cards to discover. Adding more card types, such as equipment cards to enhance the abilities of creatures and spell cards to modify the flow of a battle can make attack combinations more dynamic and interesting. Who doesn't like to see a last round comeback from a strategically used chain reaction of spell, creature, and equipment cards?

3. Fill in data gaps – Content that is not pertinent to the functionality of the game has been continually put off, but the game is at a state right now where details should be filled in to give users the full experience. Some examples of data gaps include descriptions and fun facts about U.S. cities in order to give personality to an arena, a chart of how affinities affect damage calculations, a glossary for weather terminology (such as the definition of a "subterranean continental" climate), and a help screen for basic guidelines to using the application.

4. Test, test, and more tests – Because of the rapid pace that code was being pushed out, there are areas in the code base that do not have test cases associated with them. One important example is the BattleLogic module, which controls the entire flow of battle. Since several next steps need to modify how the battle sequence works, unit tests should be put in place to ensure that old code does not break as the system continues to grow. User tests should continue to be held as well, with a particular focus on how well kids understand and use the system, and also how well kids understand and enjoy using the weather tutorials.

5. Specify reporting parameters – Another aspect important to educational games is the ability for administrators to follow along how players are progressing and to generate reports to document and evaluate this progress. Currently, Weatherlings does support writing to a custom log file and will display up to date information about players and battles, but it lacks any reporting and evaluation tools that aggregates the information. One major reason for this is that not much is known about how teachers want to use the

gathered information and what type of results teachers want to know. If these specifications can be gathered, then detailed reports can be generated, or if it is found that the stored data is insufficient to generate such reports, then more actions can be logged.

6. Create alerts via email and/or SMS – A major logistical difficulty that the Weatherlings' battle system faces is to get players to act when an action is available, since idleness may occur because the player is busy or because the player forgets to check the game. In order to remedy the second event, an alert system should be put in place to serve as a reminder, but flexible enough so that it doesn't become a nuisance. Opt-in email notification is a possibility, although this may not be the optimal medium for preteen kids, since they may check the Weatherlings site more often than their email account. Text messages that appear directly on the mobile screen (and possibly vibrate the device) would be much more useful in this situation.

7. Evaluate performance and implement smarter caching – During the play tests, one question that was continually asked was whether or not loading time was an issue, and luckily the answer was consistently no. However, determining actual timing numbers for each task is important to figure out where potential bottlenecks might be when more data and more users are added in the future. This task can be done on the server end with the Rails' built in performance measurer, or with any of the widely supported plug-ins. On the client end, running tests on Mozilla Firefox's Firebug and YSlow add-ons is one way to go about it, although these would be times relative to a desktop browser.

If necessary, different server-side caching techniques may be ways to improve performance, although none are used currently. For example, Ruby allows SQL queries and results to be saved, so database lookups may be short circuited if the results are known to be the same for multiple users or the same over a period of time. In addition, if the generated HTML for a GET request or AJAX request is static, then the result may be cached as well and future requests can avoid any processing. If this approach is used, then every resource should consider adding an expiration header in order to avoid returning stale data.

Although this list contains a lot of work that can be done, and it is only a fraction of the full task list that the group maintains, Weatherlings has already shown that there is a large potential in creating educational, dynamic, multiplayer browser-based games for mobile devices. In fact, I believe that the numerous amounts of feature requests speak positively of the extensibility and potential of the system, rather than diminishing the progress that has been made. Even with the relatively limited initial card, move, and arena data set, there is an addicting quality to the game that draws players to want to explore the game. It remains to be seen how receptive kids are to an entirely separate tutorial module though. But by carefully integrating the educational aspects into the flow of the game, then players may be motivated to learn and apply their weather forecasting skills.

V. FRAMEWORK

From the very onset, the mission of UbiqGames was not to build one or two ubiquitous games, but to build an entire *suite* of them. With the experience from developing two such applications, I will pull together the design decisions, guidelines, and features into a framework that I believe will be most effective in helping ease future development of multiplayer mobile games. The framework can be broken into four main categories: System Architecture, Reusable Modules, Frontend Design, and Third-Party Utilities. I will discuss each of these categories in turn, and then conclude with possible ways to extend and improve upon this framework.

Architecture	**Useful Modules**	**Web Design**	**Third-party code**
Stateless client	XML Parser	Screen width	Restful_authentic-ation
Timestamp data	Custom Logging	Link areas	Will_paginate
Database model as state machine	Web Resource Grabber	Consolidate refreshable content	Hpricot
Separate game and control logic	Messaging	Modal overlays	RMajick
Separate users' real and game data		Toggle visibility	Background_rb
Game codes		Client-side validation	FasterCSV

Figure 5.1 A summary of the elements in the UbiqGames framework

System Architecture

System architecture refers the design of how different components in an application interact with each other.

A critical idea used by Weatherlings and Virus was to assume the client was completely stateless (except for knowing its own identity) and to design the server to store and deliver the most up to date information. By storing the state on the server, there is no need to worry about resolving version conflicts since the server is always right and the clients just need to synchronize with the sever state. Even if clients fall in and out of connection, their next request will generate all the data they need to continue playing the game.

However, generating all the data at every request can be expensive though, since a lot of information may need to be re-sent and re-rendered. There are a few ways to decrease the impact of this transfer. In Virus, the entire outbreak state of each country did need to be refreshed, since the server was updating itself on a timed basis. However, in Weatherlings, the battle state stays the same until users trigger an update, therefore by adding a timestamp to each request, the server can decide to not respond unless it knows the client data is stale. In addition, both games made heavy use of AJAX requests to only update the specific sections of the page that required changing.

If the game has a systematic action sequence, then one design to consider is to use a database object as a state machine by storing a phase variable that indicates the correct action to render. Weatherlings uses the Battle and Matchup objects in this manner, and adds the condition that the phase does not get updated until each player's action is recorded. Virus uses the Game object to indicate whether or not players are allowed to vote.

While designing the game, careful emphasis was placed on separating game logic from control logic, for readability and practical purposes. In Weatherlings, by separating out the damage, turn, and card playing logic out of the BattleController, the controller file size was reduced by more than 50%, which conforms to the "thin controller" paradigm. In Virus, by isolating the spreading and voting code into their own module, the functions were easily invoked and run in background scripts. Furthermore, separating game constants and file paths to their own modules made tweaking the system and dynamically loading game parameters easier.

Some other design decisions that proved to be practical were separating the User model from the Player model and making use of game codes. Not only do game codes help keep unwanted users from registering, but the practical standpoint is that they allow running multiple games instances without needing to create a new database for each one. This design does require being careful to remember to check that the information retrieved from the database is for the appropriate game, which for the most part, can be accomplished by adding convenience methods to the Game object.

Reusable Modules

These modules refer to pieces of code that can be mixed in with applications when a specific functionality is needed.

XML is a popular file type used to specify data in a formatted way because it allows the use of custom tags, which can be made to be understandable by non-technical users. Weatherlings uses these data files to populate Card and CardMove data so it is easy for any users to create new

cards and moves. As described in the Weatherlings section, the xml_to_db.rb script parses the XML files by using an external library, Hpricot, so the additional work that needs to be done to reuse the script is to define a hash of procedures that relates to each tags, which defines how to translate the tag value into a database value.

CustomLogger is a short and simple piece of code that allows the developer to log custom formatted messages to a new file. By adding status messages in key places within the system, it becomes easy to track important actions taken by users and display it in a log to administrators. Further text parsing scripts may be necessary to create useful reports from these logs.

There are also a few modules that need to be tweaked a bit in order to be reused, but development is currently be done to extract the game specific code out of those modules. This includes an image grabber and a site scraper that outlines an easy procedure to retrieve and customize resources from the web, and also various messaging mechanisms, such as instant chat, private messaging, and forums.

Front End Design

These web design tips are not hard and fast rules, but general guidelines that helped start the thinking about how to format each screen in the previous two games.

- The width of each screen was kept to 300 pixels, although there is some buffer because the browsers that UbiqGames support have resize and zoom features.
- The link function should be placed in a padded element around text so that there is a press-able *area*.

- Group items that can be refreshed together so that entire sections can be replaced
- Modal overlays are useful for displaying "dead-end pages", which are screens where users would usually press the back button after viewing. This mechanism saves the reloading time to retrieve the previous page.
- CSS and JavaScript make the process of toggling visibility of DOM objects easy. A handy trick to reduce the amount of viewable content on a page is to render everything at once, but be selective of the sections that are shown and the sections that are hidden (Weatherlings applies this concept on the battle screen).
- Validating forms on the client end can reduce the need for reloading pages on error and having to re-enter information, which is a problematic task on mobile devices. Validating the parameters again on the back end is a good secondary defense against corrupting data in the models.

Third-Party Utilities

A lot of useful third-part utilities for Ruby on Rails are available, and the ones listed here have been already proven useful in the development of Virus and Weatherlings.

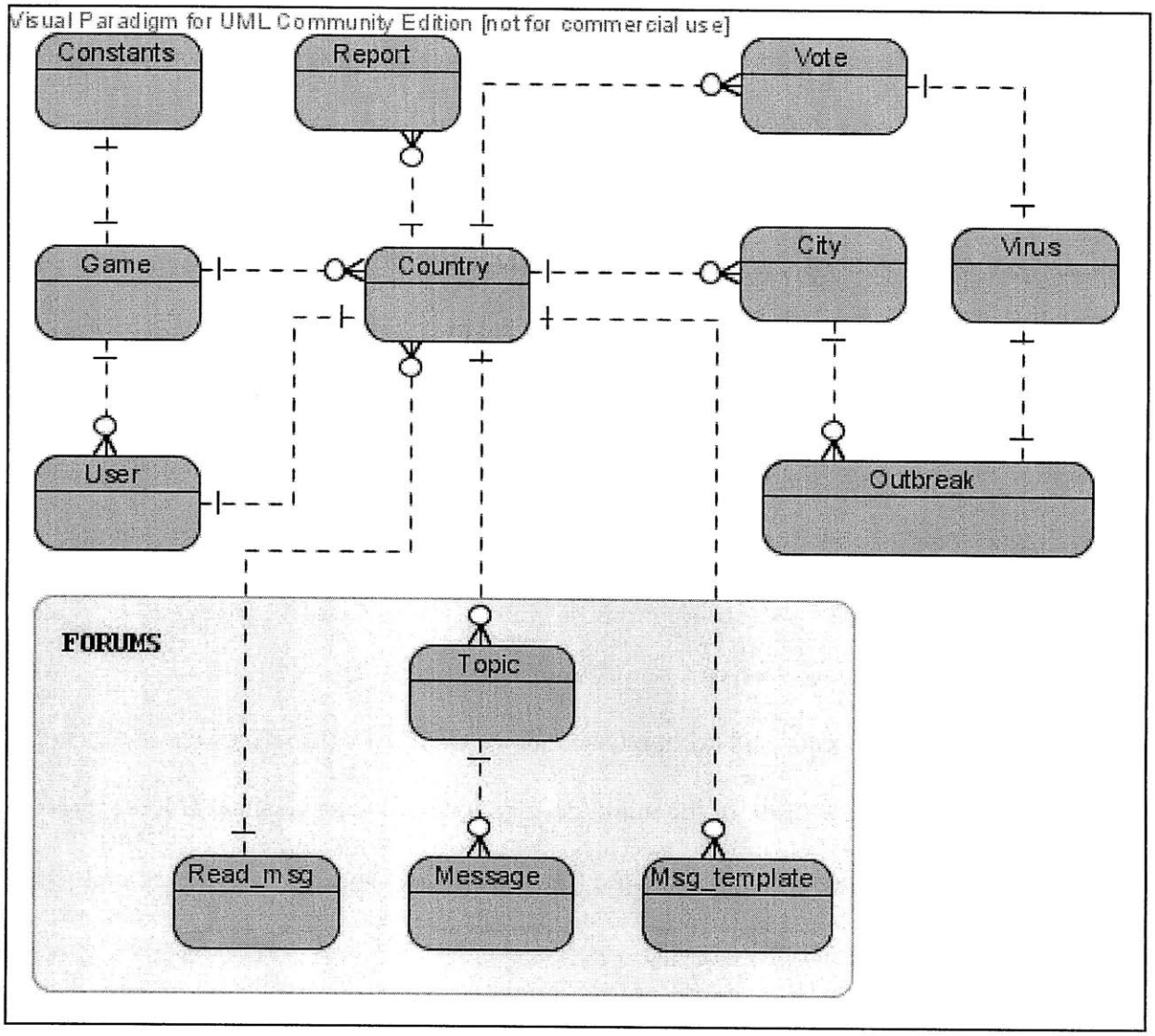
- Restful_authentication – maintains user registration and user accounts
- Will_paginate – separates lists of data into pages
- Hpricot – parses HTML, XML, and CSS files
- RMagick – contains useful image processing functions
- Background_rb – allows scripts to be run in a background process
- FasterCSV – generates comma-delimited files

Reflection

The Virus and Weatherling applications have made strong headway into developing casual, educational, multiplayer games for mobile devices, but the process is far from complete. Besides the additional features that need to be implemented for each individual game, the UbiqGames framework should continue to be extended as well, so that a wider variety of games can be supported. For example, testing how well an offline support library, such as Google Gears, incorporates with the goals of our games is essential for removing the assumptions that were made about users having a consistent Internet connection. Also packaging the scripts and modules described above into self-containing plug-ins or rake tasks is a way to make the framework more mature because it allows features to be easily "installed" and mixed into new games.

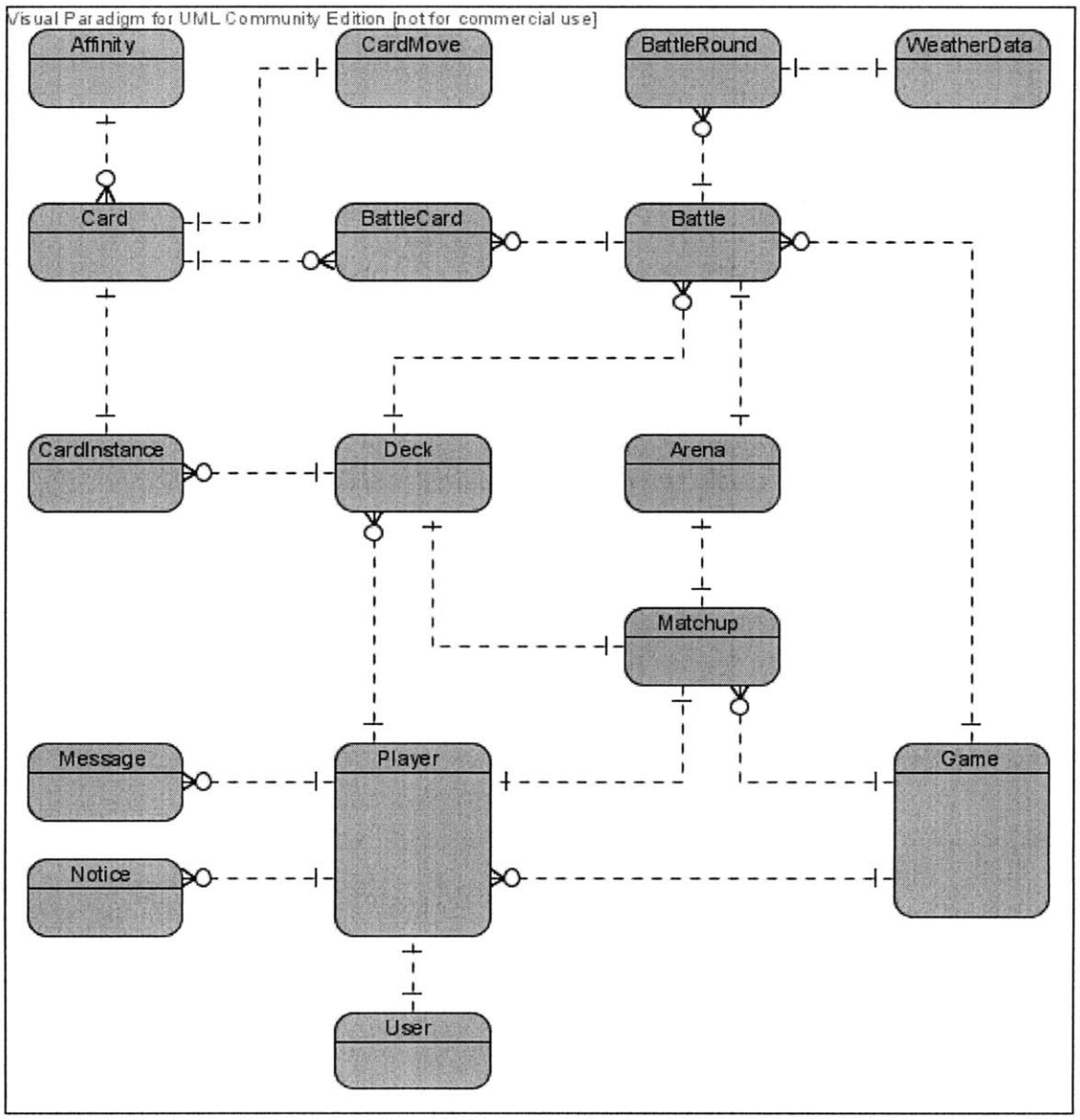
Virus and Weatherlings successfully accomplished the goals that our project set out to achieve, and were essential building blocks for the initial development of the framework. But because mobile browsers are still rapidly improving, the full potential of ubiquitous gaming is yet to be seen.

APPENDIX A: VIRUS



Complete Virus Entity Relationship Diagram

APPENDIX B: WEATHERLINGS



Complete Weatherlings Entity Relationship Diagram

Card.xml Snippet

```
<card>
     <name>BiCyclops</name>
     <description>A deadly Weatherling</description>
     <affinity>dry</affinity>
     <category>creature</category>
     <hit_points>50</hit_points>
     <rank>basic</rank>
     <cost>25</cost>
     <move1>parch</move1>
     <move2>heal</move2>
     <move3></move3>
</card>
<card>
     <name>Drizzly Bear</name>
     <description>A deadly Weatherling</description>
     <affinity>wet</affinity>
     <category>creature</category>
     <hit_points>70</hit_points>
     <rank>intermediate</rank>
     <cost>75</cost>
     <move1>bite</move1>
     <move2>flash flood</move2>
     <move3>medic</move3>
</card>
<card>
     <name>Autumn Prince</name>
     <description>A deadly Weatherling</description>
     <affinity>dry</affinity>
     <category>creature</category>
     <hit_points>50</hit_points>
     <rank>basic</rank>
     <cost>25</cost>
     <move1>combustion</move1>
     <move2></move2>
     <move3></move3>
</card>
```

APPENDIX C: PLAYTEST SURVEYS

(Note: The following survey that was handed out during the Weatherlings classroom play test was prepared by Eric Klopfer, the director of the Teacher Education Program at MIT)

User Study – Weatherlings

Battles – Were the battles intuitive? Fun? Did they take the right amount time? Did you feel that your inputs and skill affected the outputs?

Information – Were you able to access the information that you needed to make informed choices in the game? Was the information presented in a way that made sense and was useful? What additional information would you like? How else could information be presented? Was there any extraneous information?

Feedback – Was there sufficient feedback in the game to know how you were doing? Or how you could improve? Did the history record the right information for you? What else would you like recorded? How else could feedback be presented to make it more powerful?

Content – This was a prototype with canned data. Looking towards the final product, what kind of data would you like to see included in the game? How should it be presented? What skills do you think you could build interacting with that data?

(Note: The following two surveys that were handed out during a Weatherlings kid play test were prepared by Judy Perry, a project manager at the Teacher Education Program at MIT)

Weatherlings Playtest : PRE-Survey

Hi! Thanks for participating in our playtest today! Before we get started, please tell us a little about yourself. (Note: all information will be kept anonymous.)

1. What's your name? _____
2. How old are you? _____ _____
3. Are you a (*circle one*) BOY GIRL

Please CIRCLE ONE phrase (IN ALL CAPS) per question

4. I enjoy **playing computer games**	A LOT	A LITTLE	NOT AT ALL
5. I enjoy **using computers** in general	A LOT	A LITTLE	NOT AT ALL
6. I have played **Pokemon or Yu-Gi-Oh!** style trading card games...	A LOT	A LITTLE	NOT AT ALL
7. I like **playing card/battle games**...	A LOT	A LITTLE	NOT AT ALL
8. I **collect Pokemon/Yu-Gi-Oh! cards**...	A LOT	A LITTLE	NOT AT ALL

9. *Have you used any of the following?*

___iPod TOUCH

___iPhone

___another handheld computer *(what kind?)* _____

That's it. Thanks!

Weatherlings Playtest: POST-Survey

Thanks for helping us today! Please tell us what you think about Weatherlings...

1. What's your name? _____

2. Which device did you use to play Weatherlings today? *(circle one)* iPHONE ANDROID

What did you think about Weatherlings? Circle one phrase (IN ALL CAPS) per question.

3. Did you like playing **Weatherlings?** A LOT A LITTLE NOT AT ALL

4. Did you like the **characters/pictures?** A LOT A LITTLE NOT AT ALL

Please write 1 or 2 sentences for each question (Q5-Q7).

5. What was the **best** thing about Weatherlings? Why?

6. What was the **worst** thing about Weatherlings? Why?

7. What was the **most confusing part** of Weatherlings? Why?

Did you use any "weather information" during your games? (circle one)

8. **Temperature Map** A LOT A LITTLE NOT AT ALL DON'T KNOW

9. **Precipitation Map** A LOT A LITTLE NOT AT ALL DON'T KNOW

10. **Fronts Map** A LOT A LITTLE NOT AT ALL DON'T KNOW

11. **Hourly Weather Data** A LOT A LITTLE NOT AT ALL DON'T KNOW

12. Anything else you want to tell us about Weatherlings? *(you can write on the back of this page...)*

That's it. Thanks!

REFERENCES

Colella, V. (2000). Participatory simulations: Building collaborative understanding through immersive dynamic modeling. *The Journal of the Learning Sciences* 9:471-500.

Klopfer., E. (2008). *Handheld Simulation Games Augmenting Learning and Reality*, MIT Press, Cambridge, MA.

Klopfer E., and Woodruff, E. (2002). *The Impact of Distributed and Ubiquitous Computational Devices on The Collaborative Learning Environment.* In *Proceedings of the Annual CSCL Conference*, Boulder , CO, 702.

Klopfer, E., Yoon, S., and Perry, J. (2005). Using Palm Technology in Participatory Simulations of Complex Systems: A New Take on Ubiquitous and Accessible Mobile Computing. *Journal of Science Education and Technology* 14(3): 287-295.

Roschelle, J, (2003). Unlocking the learning value of wireless mobile devices. *Journal of Computer Assisted Learning* 19: 260-272.

Vahey, P. and Crawford, V. (2002) *Palm Education Pioneers Program: Final Evaluation Report*, SRI International, Menlo Park, CA.

Woodruff, E., Klopfer, E., Yoon, S., and Young, L. (2003). *Platform for Participatory Simulations: Generating Discourse with Wearable and Handheld Computers.* In *Proceedings of the Annual CSCL Conference*, Bergen, Norway.

Zurita, G., and Nussbaum, M. (2004). Computer supported collaborative learning using wirelessly interconnected handheld computers. *Computers and Education* 42: 289-314.

My World- Next Generation Wireless Ubiquitous Simulation Games. (2008). <http://education.mit.edu/drupal/myworld> (May 16, 2008).

PDA Participatory Simulations @ MIT. (2005). <http://education.mit.edu/pda> (May 16, 2008).