

Gregory J. Werner

CSCI 188/297

December 14, 2009

Summary of Scrabble Project (Final Project)

1. Introduction

I wrote an application for handheld devices which allows the user to play Scrabble against a computer agent. In particular, my design was for the Clamshell style phone provided with the Sun emulator. There were several benchmarks this program was trying to meet. A major feature was to use JSR-172 to perform dictionary lookups. Secondly, the computer player was to play a fairly sophisticated game. Thirdly, I was attempting to demonstrate some sort of internationalization. Finally, I was trying to demonstrate overall knowledge of J2ME programming.

2. JSR-172

Because I studied JSR-172 for my presentation (and also I used some of it in Lab 5), my dictionary lookup functionality was one of my earliest successes. I wrote a simple web service to run on my local machine and then generated the necessary stubs for to be included in the designed code for the handheld device. These stubs were Java files and not class files in the event I had to make minor tweaks to the files. Each time the host of the web service changes, the client files must be regenerated unless if we want to edit by hand because the web address I coded into the generated stubs.

I found out a couple issues were discovered after presentation of JSR-172. First of all, what happens if the web service is unreachable during game play. The local dictionary can not be switched on because the web dictionary may have been totally different script! Therefore, we allow the user to restart the application (not presented during class demonstration). The second issue is that we can overwhelm the web service. This is a problem because a call to a web service is blocking. The sensible solution would be to implement a timeout, however J2ME is bare bones and does not provide such a feature readily as does Apache's implementation of the Stub class. Because a timeout solution would be hard to implement in my current memory and speed challenged implementation, the best immediate step would be to reduce the trips made to

the round service. Since the presentation, I have moved to looking up 50 words at one time when the computer is playing.

The web service provided 4 functions which were used within the handheld code. The first of these was a lookup to see what dictionaries the web service offered. Secondly, a function was provided to determine if a word was in a specific dictionary (2 argument function). Third, I provided a function to determine how many different kinds of tiles are needed for that version of Scrabble (generally number of characters in the alphabet plus one for a blank tile which is universal). Included in this are methods to see what the tile distributions and scores for each tiles are. Finally, I provided a lookup for a specific type of tile given an index.

Each of the dictionaries had to be created by hand, even though the two English dictionaries contained over 100,000 words each. This required a great deal of my time. In general, a major problem with creating a dictionary for a given language is verb conjugation. For Scrabble, we would need to explicitly state every possible conjugation, which is not always provided in a standard dictionary. Other changes in form such as plurals, noun/adjective agreement, feminine/masculine/neuter endings, declensions and other such constructs must all be enumerated.

In general, I would classify my usage of JSR-172 as a major success. I would in the future need to provide a more reliable backup though in case the web service was not reachable. I created a local dictionary 1/3 the size of the original TWL dictionary, but that still gave me major memory headaches. A number of times, the app will run out of memory with the local dictionary, so I would in the future have to improve this section of my code. Due to memory constraints of the program, I would still have to recommend JSR-172 enabled in order to get the full playability effect.

3. Sophistication of Computer Opponent

One of the major tests of how good a computer game is how the computer compares with the human player. If a game is too easy or too unforgivably difficult, the user will likely put down the application pretty quickly.

For the game of Scrabble, there are 3 major components which contribute to the success of a player. First, we have chance which we have no control over (except that there is no such thing as a random number generator of course!) Second, and most important, is the number of words

a player knows. Finally, there is a strategic elements as to which is actually the best word to play.

On the second point, a computer player could theoretically have perfect play. On a standard desktop computer, this is quite easy to achieve. On a handheld device with limited memory, the computer is not able to maintain perfection. The major difficulty lies in the blank tiles which can be used as a substitute for any other valid tile (for most rule sets anyway). I found that even having a single blank could cause the program to run out of memory. Having two blanks therefore need not be considered for even a second. There are many strategies for handling the existence of a blank. Programmatically, I attempted to allow the computer to try to use one blank, but I can not guarantee reliable performance when the computer does draw a blank. Beyond the blank issue, I tied in my dictionary lookup with my strategy algorithm. This means that I tie my lookup of words to the existence of certain tiles on the board at the current time. This forces me to use a cut-off (I chose 3 words) at which point dictionary lookup stops. It is quite possible therefore that the computer will never determine the existence of the words it holds on its rack. I therefore avoid the issue of the computer player being too hard. The question is really does the computer play well enough? The answer may well lie in varying the cut-off value.

On the third point, choosing the best word to play, I note that the best play is not necessarily the highest scoring play. This is where the rule-based approach fails us and really should be replaced by something of a neural net or genetic algorithm approach (if we were not on a handheld device). It is quite natural for a scrabble player to set themselves up for a bingo on the next turn (that is using all 7 tiles to get a 50 point bonus) by playing a lesser word on the current turn. The long term benefit of such an approach is a higher score at the end. This is however a combination of instinct and probability, the former of which a computer is not very good at. This makes are cut-off even more reasonable because choosing the best play could lead to poor play on future racks and is not always a good approach.

Currently I provide just one level of difficulty. In the future, it might be beneficial to provide multiple skill levels to increase the audience I am intending to reach. In general I find I can beat the computer. My current rating on ISC (a popular online Scrabble site) is 1317 which makes me a fairly strong player. The rating of my program would certainly be under 1000, but it would take actual games to try to pinpoint an actual number.

4 Internationalization

I made a fairly concerted effort to provide a mechanism to play Scrabble in other languages. The algorithms themselves would not need to change and would provide fairly consistent play across all such languages. Allowing international play is very closely tied to other aspects of computer internationalization. Given a specific language, it is necessary to determine how that language is encoded. After that, it is necessary to have the input data files on the web service in the right format, have the design development set up correctly, have Java setup correctly, and possibly worry about phone specifics.

I tried to provide play in Slovenian. Slovenian contains 25 characters (and has 1 blank like all other languages). 22 of these letters are identical to English. The other three characters are Č, Š and Ž. I had no problem reading in the latter two characters, but the Č kept coming across as a question mark symbol (integer value 63). I was able to put together a small dictionary from what I could piece together on the web, but without a representation for all 3 of these tiles, I would be in big trouble.

I also tried to provide French. All the diacritical marks are removed in French Scrabble which makes the tiles exactly the same (except that tile distribution and points for each tile is different). To repeat, dictionary creation is a major effort and beyond the scope of this project. Therefore, the user could play French Scrabble, but with a limited dictionary which would make the game not so fun. In order to distribute this game, we would have to spend serious time on creating a playable dictionary.

If I were to strictly use images for all of my tiles and I were to change my blank substitution form a bit, I might be able to overcome this problem. I could use a program such as Word to generate these characters and then make images out of them. Certainly this requires more effort, but if I were seriously thinking of distributing this program that might be the better long-term approach.

5 J2ME Analysis

I have two separate canvases in my ui design with the rest relying on forms and alerts. The first canvas is used for the opening screen/sequence. The other canvas is my Scrabble canvas which displays the board, the user rack and the score. I did not need to use GameCanvas because the board is easily realized using an array of filled rectangles (squares to be exact). Some of the forms are a bit ugly and I considered using canvases for them, but in the end a form was the correct functional approach. I have forms for the configuration page, the exchange tiles page and the associate your blank with a letter page.

The application consists of a main thread which navigates us through the entire sequence of the application. It handles the introduction sequence and playing of opening song. Once I reach opening game play, a while loop is entered with a series of wait/notify constructs which pause at the right time. One example of a wait is when it is the player's turn and they must select which tiles to play.

A parallel structure is the CanvasState which is an attribute of the ScrabbleCanvas. It contains various game state features which must be known by the canvas in order to refresh correctly. State of the board, whose turn it is and score are examples of things the canvas must know about.

Both of these objects coordinate with the Play class which contains the various tasks which must happen on a human player's turn and on a computer player's turn. The Play class connects with the AI classes that the computer uses to determine what the best play is.

The Board object itself was done with a singleton design. Only 1 Board should ever exist, and knowledge of the board is required in several areas of the program, therefore it seems quite natural to reference it by a getInstance method. The board holds information about the tiles available, the current dictionary, the tile distribution, and what is currently on the board (excluding the current user's turn). A separate Move class is created to hold these tiles which can be taken off the board before a move is finalized.

The introduction screen references two properties which must be set before the game begins. The first is which dictionary will be used. This information must be passed on to the Board before any tiles have been drawn. Secondly, I require the human's name to be set before the program. This is not mandatory, but it is enforced nevertheless. In the future, I would probably store these two items in the RecordStore because they are probably settings which only need to be configured the first time the user plays.

The Anagram algorithm used is quite memory efficient using only a primitive single-dimension array. For other algorithms we use classes from the java.util package included in the 1.3 distribution of SE. I went through the Java provided code and created an appropriate subset for use on a handheld device. This is something that I would reuse across projects since it is such a tedious process.

I initially designed this program as an SE program. This really simplified working out the AI portions of the program. I created a Display interface which at the time was implemented with a console display. I took for granted a few of the nice features I had with the console based application, so I ran into a harder than expected conversion approach. The isFree methods in particular took a little while to get correct. The difficult task is if you try to play across tiles already on the board. As an example, say the word "on" is on the board. The user tries to play "alone" by placing an a, an l, and an e on the board. The method has to recognize that the a, l, and e tiles are being played, yet it is the alone that must be both scored and looked up in the dictionary. This process is harder than it sounds, but I finally did make the successful transition from console to ui.

6. Conclusion

I feel the program was fairly successful for what I hoped to undertake. There were some peripheral issues which took me longer than I had hoped. The J2ME aspect was very easy and very intuitive the whole way. Despite the limitations, I was able to make proper decisions pretty quickly because I have seen other such applications on limited resource platforms. The computer intelligence and the internationalization were the hardest issues, but neither really for reasons specific to handheld devices. I did bump up against memory limits, but even in an SE environment one has to deal with such concerns when undertaking game analysis. The domain itself possesses this special challenge. I really enjoyed this process, but it will translate to little real world success due to copyright and trademark issues involved with the game. I would prefer a more flexible ui in the future and touch-screen improvements may help me out in further endeavors. This app certainly does give the user what they would want though, a fun way to pass the time when the situation calls for a little diversion.