The ITB Journal

Volume 11 | Issue 1 Article 6

2010

Mobile Phone Game Localisation

Leonie Troy

Matt Smith

Richard Gallery

Mobile Phone Game Localisation

Leonie Troy, Matt Smith, Richard Gallery
Institute of Technology Blanchardstown
Dublin, Ireland

Abstract

Often, mobile phone games are developed over a short time span. Because of the additional work and complexity that localisation requires, such games are frequently produced without localisation in mind. In recent years automation and standardisation of localisation has been developed and promoted by the Localisation Industry Standard Association (LISA) and Oasis. Mobile phone game localisation involves various types of language transfer on a small scale, which challenges the localisation process carried out on a game. Our work investigated the workflow for the localisation of a mobile phone game into Spanish and German using a LISA Standard TMX (Term Base Memory Exchange) and the Oasis standard XLIFF (XML Localisation Interchange File Format). Using Unicode the game was also localised into one Altaic language (Korean) and one Semitic language (Arabic). The localisation results have been compared and contrasted using software and statistical analysis carried out on a range of methods.

1. Introduction

Many games are produced without localisation in mind due to cost, time to market and competition from other companies. This paper describes our work to help overcome the problems that may be associated with localising a mobile phone game. We aim to find a low cost, low risk approach to achieve this goal, using localisation industry standards and technologies. We will develop a workflow to localise a legacy mobile phone game in four different languages - Spanish, German, Korean and Arabic.

2. Review of Literature

This section of the paper will review localisation in general, internationalisation and localisation for mobile phone games. It will also give a brief outline of the mobile phone game Monster Madness.

2.1 Localisation

Two terms that are used when looking at localisation are: “Localisation” and “Internationalisation”. The terms Internationalisation and Localisation are similar in meaning and need to be differentiated. Sam [22] states “Internationalization involves writing and designing an application so that it can be used with different languages, date, time, currency and other values without software modification”. Chandler [5] states “Localisation is the translation and adaptation of a software or web product which includes the software application itself and all related product documentation”. Hoft [9] also states that localisation is: “the process of creating or adapting an information product for use in a specific target market”. Localisation is the adaptation of a system for a particular locale (country) which usually takes place at the level of program design and document development. It is also the process of translating a software product into other languages. It involves taking a product and making it linguistically and culturally appropriate to the target locale (country, region, language) where it will be sold. A well-localised product is one which enables users to interact with a software product in their native language. They should be able to read all interface components such as error messages or on screen text in their own language

and enter information with all accented characters i.e. characters that are distinct to their own language for example the '˜n' in Spanish.

The above are important factors to consider for localisation. When many people think about localisation they only think about the translation of one language into another. However, this may not be the case as localisation issues can also arise between countries that have a common language. For example in two English speaking countries America and Ireland, the date format in America is mm/dd/yy and in Ireland it is dd/mm/yy; the currencies and time formats are also different as well as paper sizes. When creating software projects for both countries these are some of the problematic areas that need to be addressed.

2.2 Mobile Phone Game Localisation

The localisation of games is more complex than localisation of other software or merchandise [13]. This is because games not only have text but also animations, voice over, sounds, graphics etc. that need to be localized. According to Quan [19] localisation-friendly code is code that is developed with localisation in mind. In order to achieve this there are many parameters that need to be looked at when localising a mobile phone game. The literature (for example see [5] and [19]) yields a number of different sets of parameters that need to be considered. According to Chandler [5] some of the features that need to be looked at when localisaing a software product are:

- Game code
- Display text
- Device being used

Other areas that are important for the localisation of mobile phone games are:

1. Displaying of international characters

If the displaying of international characters was not taken into consideration then special characters outside the source language (the language being translated) such as the '˜n' in Spanish would have problems displaying correctly or may not display at all.

2. Text in user Interaction

Concatenation is used to connect or link words or phrases in a series or chain. In Computer Science the term is used to arrange strings of characters into a chained list. When displaying text concatenation should be avoided as it may lead to text displaying incorrectly e.g. running off the screen. Concatenation may be created in a game by pulling two separate text strings from the game code and displaying them as a single sentence in the game. This can cause a lot of problems in localisation with verbs, tenses and genders. However, they may display grammatically correctly in the English language but their translation may mean something completely different. [5]. An example of concatenation would be two separate words such as 'Shalom' and 'Ofam'. When they are pulled together/concatenated they would produce the string 'Shalom Ofam'.

This can be seen in Figure 1.

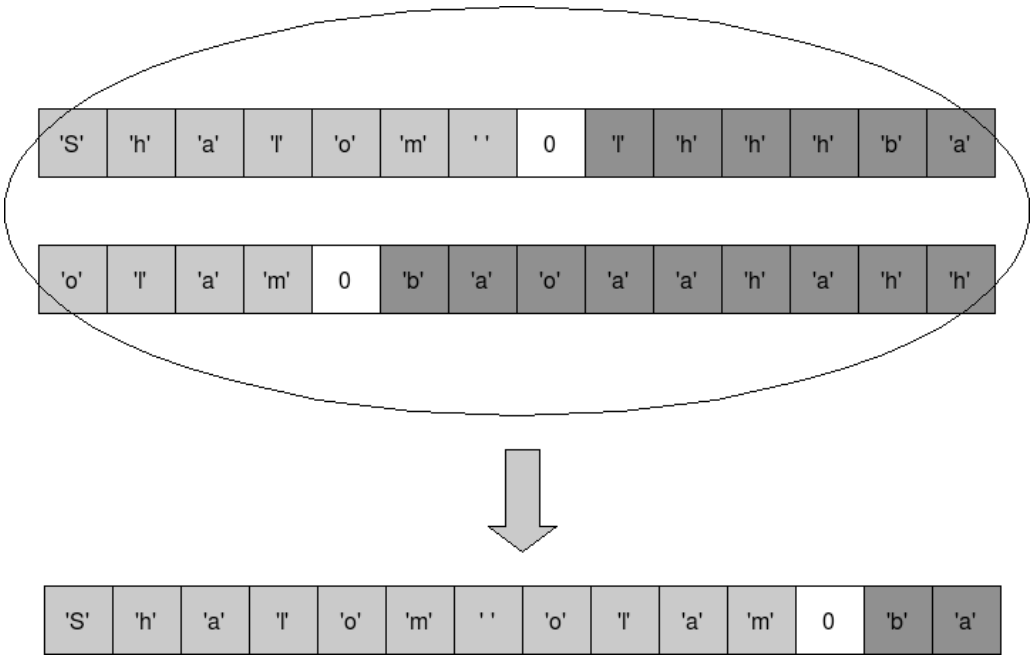

Figure 1: Example of Concatenation [25]

3. Font Properties

The type of font selected is very important in localisation as text and graphics are the main interactive elements between the user and the game. Texts that are too large or too small may cause the text to overlap or be difficult to read. "Translated words are usually 25% to 30% larger than their English spelling" [5], [10], meaning that the English word is small compared to the translated word. This is due to the fact that English is a compact language compared to other languages.

The font where circumstances allow should be scaled down to accommodate the increase without making the text difficult to read. If the translated word was transferred back to English then the text would shrink by the same amount (about 25% to 30%). Some languages need a larger font nthan others. Take Japanese and English fonts for example. If the font size was small in Japanese it would make it almost impossible to read as each Japanese character is very detailed but the English version would be legible and easy to read.

4. User Interface (UI) Design

There are many factors to consider when designing a localisation-friendly UI. One of the main factors is to design the UI to cope with the localised text. If a button used to display the word "exit" has to be translated into German then "ausgang" would be displayed on the button which has three characters more than the English text. It can clearly be seen that the German word is longer and will cause problems when trying to place it on a button that contains the English text. Where possible it would be a lot more efficient to use buttons where the UI's can be scalable, such as drop-down boxes, text boxes and other elements that can be modified up or down depending on the information it is holding. Using icons where possible is a lot more programmer-friendly

than translating text on a box as described above. Icons cut down on the amount of localised text to be used and reduce the amount of linguistic bugs such as truncation and concatenation. An example of this is using a silhouette of a single player to indicate a single player and a silhouette of two players can be used to indicate a two player game. Alternatively numbers could be used in much the same way, to depict for example the number of players. Cluttered UI screens should always be avoided. In the case of a mobile phone game this can be a difficult concept to overcome as not all mobile phone screens are the same size [5].

The international date and currency formats need to be supported as the dates are displayed as dd/mm/yy in Europe and mm/dd/yy in America. The currency is Dollars in America and Euro in Europe. Special characters must be available on the input screen for mobile devices. Sometimes there is a UI screen depicting a keyboard to save the highest score. To allow the user to input data that keyboard must have special characters and numbers in the localised version.

5. Audio and Video Graphics that contain text

The decision to localise audio and video graphics is another important factor to consider when localising a mobile phone game. For example if the game was developed in England and it contained audio that many English or European people were familiar with (e.g. chart music) this may have to be changed in order to facilitate localisation in another country. If the game was going to be localised for the Korean market then the programmer should change the audio to music,that would be recognised by that country. The use of video graphics is another issue that has to be taken into consideration when localisation a game. For example the Monster Madness logo that can be seen in Figure 2. The game has a title screen which depicts the game logo in English. If it were to be localised into Korean, Spanish etc. the programmer or company has to decide if the logo has to be localised into the relevant language or does it stay as it is as a way of recognising the game in any country worldwide.

Figure 2: Monster Madness Title Screen [24]

When games are localised not only has the text to be taken into consideration but in some cases the graphics, depending on the country [6]. For example in Germany strict censoring laws ban games that depict scenes containing blood. When localising the

game the graphics are changed to a green substance instead of blood. Also in some cases were roadsigns or directions are used, these graphics would have to be localised in order for the end user to be able to engage with the game. This concept also applies to mobile phone games where the graphics would have to be tailored to suit a certain locale.

3. Monster Madness Game Localisation

In this research the mobile phone game that was used is called Monster Madness. This is a legacy game from Eirplay Games [8]. It is a simple shooting game and in order to get an idea of the game a screen shot of it can be seen in Figure 3.

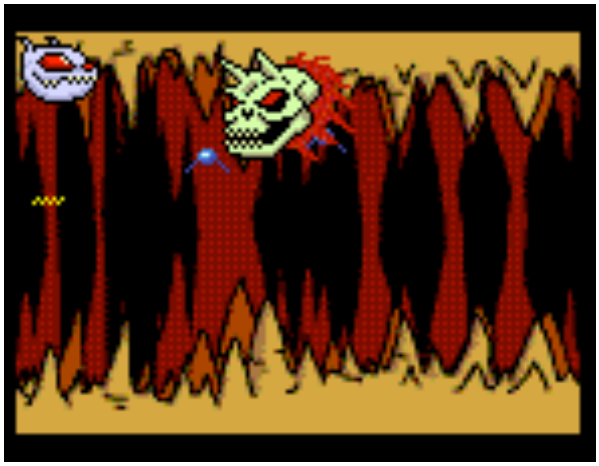

Figure 3: Monster Madness Screen Shot

When localising the Monster Madness Game the main factor that will need to be taken into consideration is the displaying of international characters. A lot of localisation factors that have been discussed do not apply to this game because user interaction takes place through text and not audio or graphics. Examples of these are the graphics (no icons need to be changed) and audio (there is no audio in this game). However, when designing all of these issues would have to be considered from the early development stage.

Localisation should be considered from the onset when creating a game [4]. There are two main types of situations for localising a game. The first is the localisation of a legacy game and the second is the localisation of a new game which is been designed with localisation in mind from the start. One of the most important aspects in localisation in a new game is the separation of text and audio files etc. From the software source code, as this makes it much easier to localise. If this was not considered (as in the case of a legacy game) then the text and audio are embedded in the source code files. If the text were scattered in different source code files it would be difficult to locate and translate. The programmer would then have to go through the code line by line, in which case it could be thousands of lines, and extract all of the translatable text and place them into a separate source code file in order to facilitate localisation. On big projects this could take up to a number of weeks. It is much more efficient to place the translatable text into a separate folder away from the source code.

When developing a game the life cycle of the game has to be taken into consideration. Today many games are created from start to finish in about three months. When localisation is involved this may take longer. [13]

4. Methodology

The Monster Madness game was written in J2ME (Java Platform Micro Edition). J2ME is a Java platform aimed at developing software for small applications such as mobile devices [3]. The IDE (Integrated Development Environment) that was used to develop the software was Netbeans 5.5.1 as it is one of the IDE's that contain a mobile phone emulator to test the code [1].

4.1 Background to the Monster Madness Game

The Monster Madness mobile game was designed by Eirplay Games [8]. This game was used in the SECASE (Software Engineering CASE studies) [2] project by the Institute of Technology Blanchardstown (ITB).

The Monster Madness game is a legacy game and was built without localisation in mind. Due to this all of the strings that contain the text or languages are found throughout all the classes in the code. This influenced a workflow for the localisation of the game.

4.2 Approach Adopted

To create a workflow to localise the game, the first step was to decide what content of the game needed to be localised. The following is a list of the components of the game that required localisation:

- text strings appearing on line during the game play
- text for control menus
- images in the game that contain text
- sound (none)
- video (none)
- game play content (any violence or cultural issues that could be offensive, illegal or inappropriate in other locales)

After the localisation aspects of the game were defined the next stage was to look at the source code of the game. The components of the game that had to be localised had to be located in the Java source code and multimedia content of the game. The following were the aspects of the source code that needed to be localised:

- 13 Java classes in the source code contained text.
- Only the title screen contained text and a decision was made to keep it in English (as with most branded games).
- 5 menu screens contained text.

The next phase was to compose a list of options to modify the source code of the game in order to localise it. Some of these options included:

- Structure 1 - do a search and replace all of the hard coded strings, leaving the code unchanged
- Structure 2 - extract all UI text to a separate class [11]
- Structure 3 - at run time detect the locale (the language it is running in) and make the game read in the translation for the current locale from an external file

- Structure 4 - at run time detect the locale and choose the correct string embedded in the Utils.java class

The option to extract all UI text to a separate class was chosen to localise the game as it was the efficient way to handle multiple languages in one class. The next stage was to design the workflow for localisation. The workflow consisted of the following prototypes and versions of the game:

1. Prototype 1 - Version 1 - Original version of the game with unstructured code and a single language. The English text was hardcoded throughout different classes in the source code.

2. Prototype 2 - Version 2 - This version of the game had structured code and a single language. All of the UI text (English words and phrases) from the original source code was put into a new class called Utils.java.

3. Prototype 3 - Version 3 - This version of the game had structured code and multiple languages. A multi-dimensional array and index were created in the Utils.java class (hand coded) allowing for more then one language to be held in the code. The indexes allowed the user choose what language they wanted the game to run in. This version contained English, German and Spanish hardcoded into the code.

4. Prototype 4 - Version 3 - This was a variation of Version 3 using a XLIFF file (see Section 3.3).

A code generator was designed to read from the XLIFF file and output the source and target languages into a Utils.java class. This reduced the file size and download speeds of the game. If the XLIFF file was read into the game code and not a code generator the file to be downloaded to a device would increase in size and run a lot slower. The Utils.java class would have almost doubled the file size, but by keeping them in a separate class this problem was avoided. The XLIFF file was created by the use of Swordfish Translation Editor software [15]. This took in a XML document, asked the user to translate the text and outputted the XLIFF document. The XLIFF document was also hand coded, but this led to relevant information and "id" tags being omitted, therefore the use of Swordfish produced higher quality and validated documentation that the programmer could not achieve by hand coding.

5. Prototype 5 - Version 2 - This was a variation of Version 2 using a TMX file [20], [23], (see Section 3.4). A code generator and the Swordfish Translation Editor software was used in the same manner as Prototype 4.

6. Prototype 6 - Version 2 - This was a variation of Version 3 using Unicode (hand coded) to translate the game into Korean and Arabic. The operating system that was on the PC was Windows 2000.

This platform does not facilitate the input of such languages. The only solution to this problem was to install a Linux operating system (as this recognises Arabic and Korean languages) or to use Unicode. This version of the Monster Madness game used Unicode. In order to translate from English to Arabic or Korean, translate.google.com

was used. To translate from Arabic or Korean to Unicode Babelpad software was used. This encrypted each line or symbol in the target languages into a unique Unicode hexadeciamel number.
An example of Korean and Arabic phrases and their Unicode equivalent can be seen in Figure 4.

Figure 4: Example of Unicode displaying the Korean phrase "Exit" and the Arabic phrase "OK"

4.3 XLIFF (XML Localisation Interchange File Format)

XLIFF is a localisation technology standardised by Oasis in 2002. It is used as a format to exchange localisation data between many people who are involved in a localisation project such as translators and localisation Engineers [17]. XLIFF is an XML based format that allows translators to concentrate on the translation of text and for localisation Engineers to be able to intemperate that text and use it in a software project. The XLIFF document consists of two sections - a header and a body.

The header section contains relevant data such as - contact information, project phases, pointers to reference material, and information on the skeleton file. When a file is converted to XLIFF, the structural formatting is extracted and stored in a skeleton file. The skeleton file indicates where the text from any given trans-unit should be placed.

The format of this file is not defined by the XLIFF specification, so conversion tools can use any format they choose. The conversion tools should be able to recover the original source file, given the skeleton file and the XLIFF file. [17]. The body section contains trans-unit elements. These are the main elements in an XLIFF file that contain the source and target languages i.e. the localizable text and its translations. The trans-unit elements contain source, target, alt-trans, and a handful of other elements. The example of my work below shows a simple Xliff programme:

```
<?xml version = ‘ ‘1.0’ ’ encoding = ‘ ‘UTF□8”?>
<xliff version = ‘ ‘1.2’ ’ >
<? encoding UTF□8?>
<file datatype = ‘ ‘xml’ ’ original =C: Documents
and Settings n b00000652 nDesktop n Swordfish
Projects n ES Swordfish n Spanish
Translations.xml’ ’
source□language = ‘ ‘en□GB’ ’
target□language = ‘ ‘es□ES”>
<header>
<skl>
<external□file href =
‘ ‘C: n Program Files n Swordfish n skl n Spanish
Translations.xml.61214.skl’ ’/ >
</skl>
</header>
<body>
<trans□unit>
<source xml : lang = ‘ ‘en□GB’ ’>Get ready for
next level </source>
<target xml : lang = ‘ ‘es□ES’ ’>Consiga listo
para el Nivel siguiente </target>
</trans□unit>
<trans□unit>
<source xml : lang = ‘ ‘en□GB’ ’>
Game not functioning correctly. </source>
<target xml : lang = ‘ ‘es□ES’ ’>Juego que
no funciona correctamente. </target>
</trans□unit>
</body>
</file>
</xliff>
```

In the code above the trans-unit element contains an id attribute which is used determine where the segment goes in the original document. The trans-unit element contains the source and target elements. The target element holds the translated source element after linguistic review has taken place. Alt-trans elements are used to hold attributes associated with the XLIFF file such as match-quality and the tool used to produce the file. According to Raya [21] Figure 5 depicts the translation process from the original document to the XLIFF document.

The steps involved in this process are:
1. Text extraction: Separation of translatable text from layout data.
2. Pre-translation: Addition of existing translation to the XLIFF file generated in the previous step.
3. Translation: Performed by a professional translator.
4. Reverse conversion: Generation of a translated document from the translated XLIFF file.
5. Translation memory improvement: Storage of new translations in a translation memory (TM) database for later reuse.

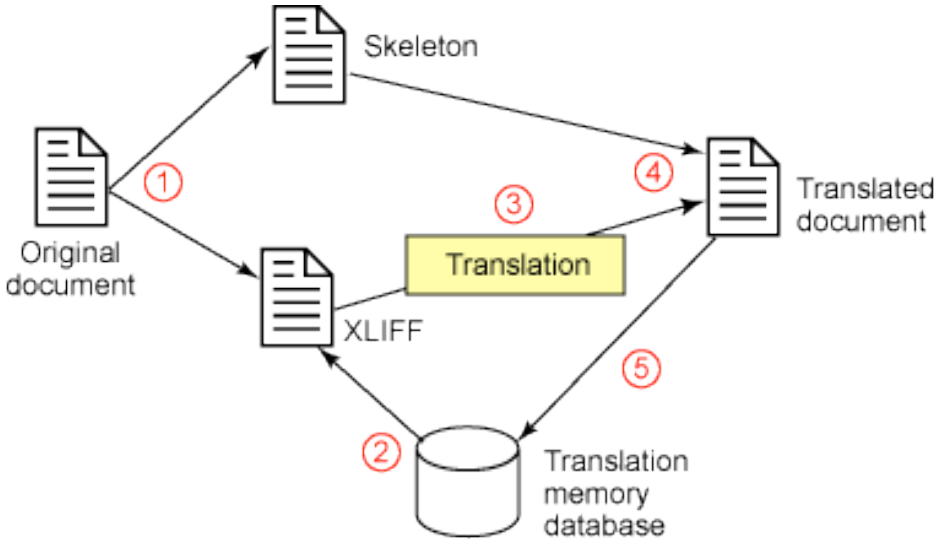

Figure 5: XLIFF Process [21]

Maxprograms Swordfish Translation Editor software was used to produce the XLIFF files. An XML document was written that contained the English (source) text to be translated. The XML document was opened in Swordfish to be converted to XLIFF. During conversion from XML to XLIFF a translator is required to translate the source language. These translations are then stored in a database in Swordfish and can be used again in the future, reducing translation times on further projects.

Some of the benefits of using XLIFF are that translators and Engineers do not have to learn many different programming languages to translate text and use these translations in various different projects. XLIFF allows for the text to be translated, extracted and used in any programming style that the user requires. This leads to a document that is structurally formatted and supports a localisation processes in almost any project or task.

4.4 TMX (Term Base eXchange)

TMX has existed since 1998 and is a certified standard format. TMX is developed and maintained by OSCAR (Open Standards for Container/Content Allowing Re-use), a LISA (Localisation Industry Standard Association) Special Interest Group. TMX is the open, XML-based standard for exchanging structured terminological data. A formal definition of TMX from the LISA website states:

"TMX (Translation Memory eXchange) is the vendor-neutral open XML standard for the exchange of Translation Memory (TM) data created by Computer Aided Translation (CAT) and localization tools. The purpose of TMX is to allow easier exchange of translation memory data between tools and/or translation vendors with little or no loss of critical data during the process. " [12]

Like XLIFF, a TMX document is divided into two sections - a header and a body. The information about the document is described in the header and the main context is described in the body. In the body there are translation unit elements <tu> that contain a collection of translations. Each translation unit contains text in one or more languages in translation unit variant elements <tuv>. The text of a translation unit variant is enclosed in a <seg> element. [12]

An example of a TMX file that was converted from XML to TMX through the use of the Swordfish Translation Editor can be seen below.

```
<?xml v e r s i o n = ‘ ‘1.0 ’ ’? >
<tmx v e r s i o n = ‘ ‘1.4 ’ ’ >
<h e a d e r c r e a t i o n t o o l v e r s i o n = ‘ ‘ 1 . 0 . 0 ’ ’
d a t a t y p e = ‘ ‘ winr e s ’ ’ s e g t y p e = ‘ ‘ s e n t e n c e ’ ’
adminl ang = ‘ ‘EN□US’ ’ s r c l a n g = ‘ ‘EN□US’ ’
o□tmf = ‘ ‘ abc ’ ’ c r e a t i o n t o o l = ‘ ‘XYZTool ’ ’ >
</ he ade r>
<body>
<tu>
<t u v xml : l a n g = ‘ ‘ en ”><seg>Winner
</ seg></ tuv>
<t u v xml : l a n g = ‘ ‘ es□es ”><seg>Gandor
</ seg></ tuv>
</ tu>
<tu>
<t u v xml : l a n g = ‘ ‘ en ”><seg>Auto F i r e
</ seg></ tuv>
<t u v xml : l a n g = ‘ ‘ es□es ”><seg>Fuego Auto
</ seg></ tuv>
</ tu>
<tu>
<t u v xml : l a n g = ‘ ‘ en ”><seg>New Game
</ seg></ tuv>
<t u v xml : l a n g = ‘ ‘ es□es ”><seg>Neuvo Juego
<seg></ tuv>
</ tu>
</body>
</tmx>
```

Some of the advantages of using TMX are that it gives the user control over the translation of the language that it is representing. It keeps consistency and quality by allowing the user to control the terminology which leads to the localized text being more likely to represent what the user wants it to. This also improves the quality of the text. It reduces the time spent on localising a product. By using TMX it allows for flexibility of the tool being used.

An example of the game running in English, German and Spanish through the use of XLIFF and TMX can be seen below.

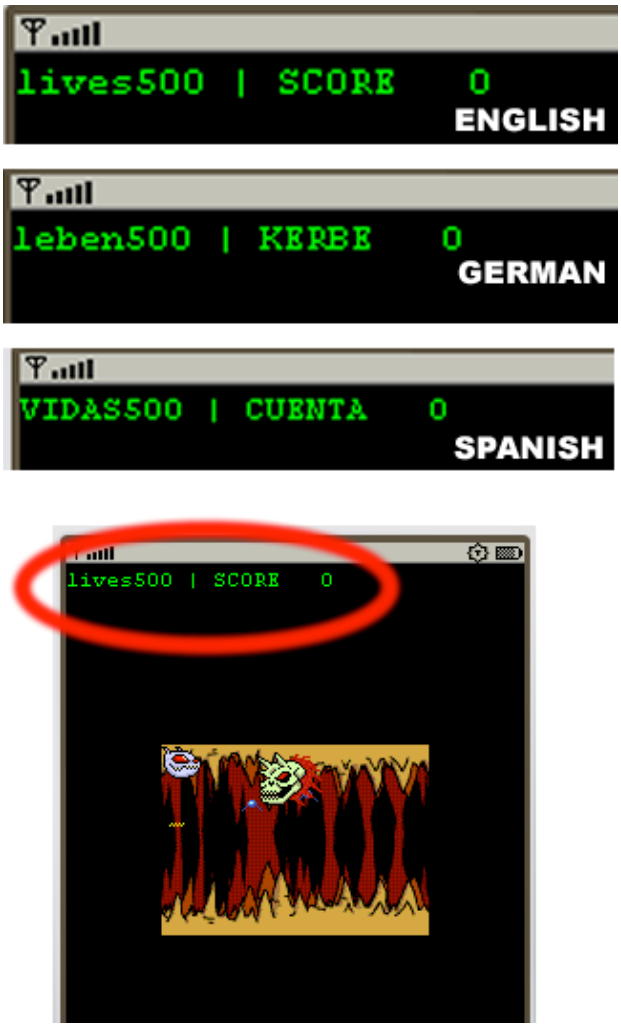

Figure 6: Monster Madness Screen Shot in English, Spanish and German

4.5 Unicode

As has been stated before, one very important factor in localisation is the displaying of international characters. If this was not taken into consideration then special characters outside the source language would have problems displaying correctly or may not display at all. Languages such as Korean and Arabic, are not recognised by Windows operating systems. To overcome this problem, one option was to use a Linux operating system (which recognises these languages) or the other is to use Unicode.

Unicode is the standard for representing text characters. Unicode provides a unique number for every character of a keyboard so for example the letter ‘a’ corresponds to the number 97 in the decimal system and 0061 in the hexadecimal system. The advantage of this system is that no matter what the platform, program, or language being used, the device will recognise the Unicode characters and be able to translate them into the required language. This is due to the fact that each language has its own Unicode.

It also enables a software product or a website to be targeted across multiple platforms, languages and countries without re-engineering. [18]. Unicode is a system that uses two bytes (16 bits) and provides a unique number for every character regardless of the programming language etc. This gives the game code the capability to display more than sixty five thousand characters in many languages including Spanish, German, Arabic and Korean. Some languages such as Arabic are referred to as bidirectional text. This text is read from right to left (RTL) instead of left to right (LTR), therefore, the device must support the capability to display text in the correct manner and accept text inputs in both directions. A screen shot of the game running in Arabic and Korean TMX can be seen in Figure 7.

Figure 7: Monster Madness Screen Shot in Arabic and Korean

5. Results

The research into the results of the localisation of the game is still ongoing. However, we have primilary results on the following:

1. Software Testing - Junit/JMUint
2. Game Testing
3. Comparison of file size and download time.

5.1 Software Testing

After the game was localised into four different languages, the first test that was applied to it was unit testing. This ensured that the code still functioned correctly after the game was localised successfully. With unit testing every class that is written by a programmer should have a corresponding unit test to test the functionality of the code [7], [14]. JUnit testing is a toolkit used for performing unit testing on Java programs. With JUnit testing there are two types of tests that can be written. One is to test a source file (a pre-written class) and the other is to write a class test for a class that has not been written (a post-written class) [16]. In the Netbeans IDE 5.5.1 JUnit examines the source file and file and generates sample test code for each method in a class. For applications, Netbeans has the capability to allow the programmer to automatically create a JUnit test for any method or class. When the test is created the programmer can modify the parameters and values to test the piece of code. For example if a calculator class was written to take in two numbers to be added together, the programmer would have create a test stub in JUnit, input two numbers into the test, run the test and check to see if it passed or failed. However, this is not the case for mobile applications.

Using JUnit for mobile applications requires a different approach. The testing principals are the same but the mobile phone application is run on an emulator and will need to be tested on various mobile phones. JUnit testing on mobile applications is conducted through JMUnit. This allows for fully automated regression testing of mobile application on an emulator and on a mobile phone. [7].

The Monster Madness game was not designed for testing with JMUnit. If every method and class were to be tested in the game, the code would have to be rewritten to facilitate the testing, which is time consuming. Due to this only one class was tested - the SmallEnemyMonsterSprite class which contains fourteen methods. The test stubs are created in the same way as Junit but a testing approach has to be decided upon for each method. This was achieved by testing each valid input to ensure that it does not throw an exception. Then the inputs were changed to force it to throw an exception. This proved that the code would run correctly under the correct conditions. In order to run JMUnit, the main programme needs to be ran and then the test-suite needs to be activated. This allows the tests to be run.

The test for the SmallEnemyMonsterSprite class can be seen below. It shows that the outcome of the test suites when all of the methods have passed successfully and also when they have failed. To rectify the fail, the JUnit test result window in the IDE shows exactly where in the code the failure occurred. This failure in the test could be due to one of the following reasons [7]:

1. An incorrect test specification
2. An incorrect design of code implement the test specification
3. A coding error in the code under test
4. A failure of other classes or methods
5. Some other reason

Once the problem is fixed the testing code should then pass successfully.

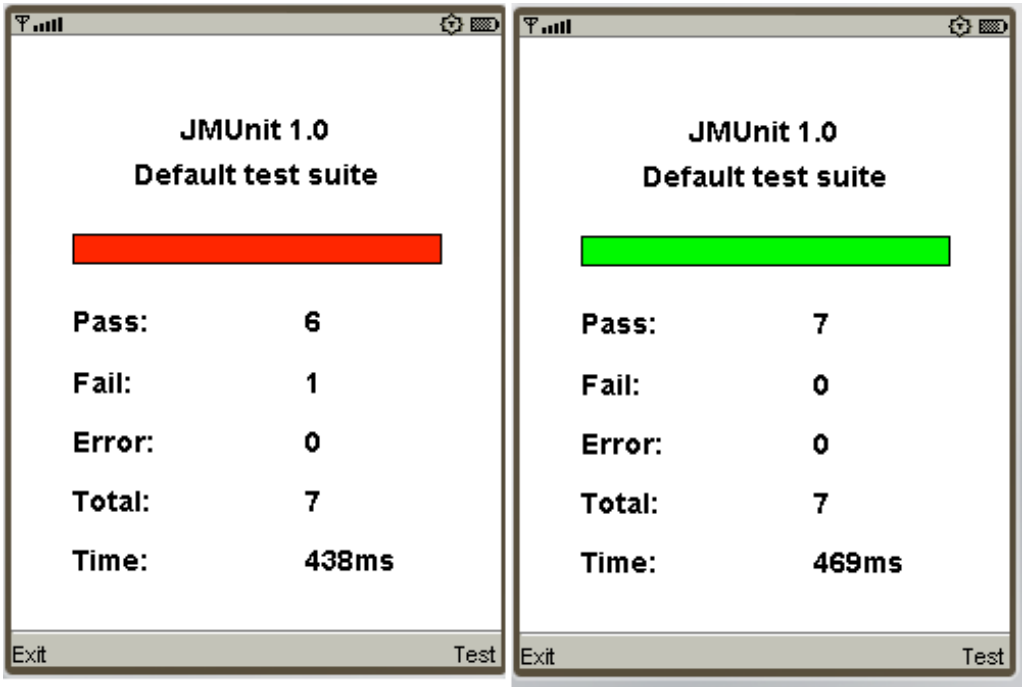

Figure 8: SmallEnemyMonsterSprite class test - Pass and Fail

5.2 Monster Madness Game Play Testing

Game Play Testing was the second test carried out on the localised versions of the Monster Madness game. This stage of testing is usually carried out by a native speaker of the language that the game is running in. Game Testing ensures that the game has been played and ran through every possible scenario that could take place in the game. The Monster Madness game was tested on the emulator and on two mobile devices - Palm Treo and Sony Ericcson W5801. All of the bugs and errors that were encountered during game play are then logged and sent to the programmer. The following conditions outline the tests that the game went through at this stage.

1. Run all version of the game to the very end.
2. Terminate all versions of the game after each level is completed.
3. Terminate all versions of the game during each level.
4. Check all sub menus of the game. E.g. Settings, High Score, About, Settings. game play tests were carried out for the German and Spanish version of the game.

Some of the problems that were encountered in the Spanish and German games are listed below. The majority of errors were untranslated text, therefore, when running the game in Spanish or German, some of the text was still in English.

1. “Back” in English
2. “New Score” in English
3. “Winner” in English
4. Monster MIDP 2 - Spelling Error
5. Help and About menus were still in English

There were also problems with the text running off the screen (concatenation). This problem was resolved by changing the phrase or text into a shorter one. Another problem that was encountered when running the game on a mobile phone and not on

the emulator was during the deploying stage. When the Spanish version of the game was deployed and ran it functioned correctly. When the German version was then deployed and ran, the High Score menu was still in Spanish. This problem was solved by changing the file name to be deployed and changing some of the code.

5.3 Files sizes and speeds

In order to deploy the game onto a mobile phone, the .jar and .jad files are sent to the device via bluetooth, infra-red, OTA (Over the Air) or USB cable. Therefore, for efficient download speeds and execution time of the game, the .jar and .jad files need to be small. One way of achieving this was to keep the XLIFF and TMX files out of the game code so when the game was compiled and ran in Netbeans these files did not get encapsulated into the .jar file, reducing it in size. This stage has not been completed and is the next phase to be tested.

6. Conclusions

This paper looked at localisation and in particular, localisation of a mobile phone game. It describes our work to help overcome the problems that may be associated with localising a mobile phone game. We found a low cost, low risk approach to achieve this goal, using localisation industry standards and technologies. We created a workflow that was developed to localise a legacy mobile phone game into four different languages - Spanish, German, Korean and Arabic.

The methodology section gave a clear and concise description of the steps that were taken to localise a mobile phone game. It also described the two industry technologies and standards that were used to achieve this - TMX and XLIFF and showed example code for both. This section also displayed the strengths associated with the use of Unicode to localise a mobile phone game into an Altaic language (Korean) and a Semitic language (Arabic). The results section of this paper outlined the software testing that was carried out on the game and the main problems that were encountered when test playing the game. It also provided solutions to fix any bugs or errors that were found in different versions of the game.

6.1 Future Work

Our aim is to expand on the work that we have done to date by increasing the capabilities of the game. This will be achieved by adding audio and graphics. The graphics will cut down on the amount of localised text to be used and reduce the amount of linguistic bugs. An arbitrary example of this would be to use a silhouette of a single player to indicate a single player and a silhouette of two players can be used to indicate a two player game. As part of the research, one of the aims is to develop a project lifecycle that mobile phone game companies can follow when developing games for localisation. This will be achieved by going back to industry (Eirplay Games) and getting feedback and recommendations on the prototypes that have been developed. This will also be achieved by evaluating the workflow of that was used for this project.

References

[1] Netbeans website. Online at www.netbeans.org last visited on 26th January 2009, 2009.
[2] Secase website. Online at www.secase.eu/course/view.php?id=4 on 26th January 2009.
[3] Sun java website. Online at java.sun.com last visited on 26th January 2009, 2009.

[4] D. M. Carthy. The Complete Guide to Game Development Art and Design. ILEX Press, United Kingdom, 2005.
[5] H. Chandler. The Game Localisation Handbook. Charles River Media, Inc, Massachusets, 2005.
[6] P. Fitzpatrick. Interview with Peter Fitzpatrick, Senior Program Manager at Microsoft Game Studios. Online at www.gamedevelopers.ie/features/viewfeature.php?article=141 last visited on 20th Februrary 2008, 2008.
[7] R. Gallery. Testing and Junit, Online at www.secase.eu, 2008. 2008.
[8] E. Games. Online at www.eirplaygames.com last visited on 26th January, 2009.
[9] N. Hoft. International Technical Communication. John Wiley and Sons, 1995.
[10] R. Jacobsen. Don't Get Lost in Translation. This article originally appeared in the January 2006 (Vol. 29, No. 1) issue of The Editorial Eye, 2006.
[11] J. Lam. J2ME amd Gaming. Online at www.jasonlam604.com/books.php, last visited on 26th January 2009, 2007.
[12] L.I.S.A. Localisation Industry Standard Association. Article on Translation Memory eXchange (TMX). Online at www.lisa.org/Translation-Memory-e.34.0.html, 2009.
[13] P. Lynch. Interview with Peter Lynch CEO of play Games. Online at www.secase.eu last visited on December 2007, November 2007.
[14] V. Massol. JUnit in Action. Manning, 2004.
[15] Maxprograms. Swordfish Translation Editor Software. Online at www.maxprograms.com/products/swordfish.html last visited on 19 February 2008, 2009.
[16] A. Myatt. ProNetbeans IDE 5.5 Enterprise Edition. APress, America, 2007.
[17] S. D. Network. Sun Development NetworkArticle: XLIFF: An Aid to Localisation, Online at www.developers.sun.com. 2009.
[18] B. Nolan. Masters in computing Notes. Institute of Technology Blanchardstown, Dublin, Ireland, 2006.
[19] B. K. Quan Zhan. The Localisation of Digital Games:A Case Study in China, Proceedings of DiGRA 2005 Conference. 2005.
[20] R. M. Raya. Article: XML in localisation: Reuse Translations with TM and TMX, Online at www.maxprograms.com. 2009.
[21] R. M. Raya. Article: XML in localisation: Use XLIFF to translate documents, Online at www.maxprograms.com/articles/xliff.html. 2009.
[22] J. Sam. Derived from A question posed by Nitin Agarwal. Online at www.jguru.com, last visited on 19 February 2008, 2001.
[23] J. Schinharl. A Terminology Checking Tool to ensure Translation Consistency. ITB Library, 2007.
[24] M. Smith. Monster Madness Title Screen. Online at www.secase.eu, 2007.
[25] Thedsadude. C language shalom olam string concatenation.png. Online at images.google.ie last visited on 12th May 2008, 2007.