Lappeenranta University of Technology
School of Engineering Science
Software Engineering
Master's Programme in Software Engineering and Digital Transformation

Joel Salminen

DESIGN OF LOCALIZATION WEB ENVIRONMENTS

Examiners: Professor Ajantha Dahanayake
M.Sc. Eng. Paula Laaksonen, GE Healthcare Finland Oy

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma
Master's Programme in Software Engineering and Digital Transformation

Joel Salminen

Lokalisointiin tarkoitettujen verkkoympäristöjen suunnittelu

Diplomityö

2019

87 sivua, 19 kuvaa, 1 taulukko, 3 liitettä

Työn tarkastajat: Professori Ajantha Dahanayake
DI Paula Laaksonen, GE Healthcare Finland Oy

Hakusanat: web-kehittäminen, lokalisointi, käytettävyys, ylläpidettävyys
Keywords: web-development, localization, usability, maintainability

Lokalisointi parantaa ohjelmistotuotteiden menestymistä kansainvälisillä markkinoilla. Tämän tutkimuksen tavoitteena oli vähentää lokalisointiin kuluvaa aikaa ja kustannuksia suunnittelemalla parempia ohjelmistotyökaluja. Tämä tutkimus tarjoaa toteutussuunnitelman uudelle verkkopohjaiselle ohjelmistoympäristölle, jota voidaan käyttää lokalisointiprosessin eri vaiheissa. Verkkoympäristöön kuuluu työkaluja käyttöliittymätekstien syöttämiseen ja kääntämiseen sekä dokumenttien kuten käyttöohjeiden käännösten tarkistamiseen. Suunnitelman kelvollisuutta arvioitiin käytettävyystestin avulla. Tulosten mukaan verrattuna nykyisiin ratkaisuihin käyttäjät suosivat uutta ja että uusi käyttöliittymä on riittävän intuitiivinen kokemattomille ja satunnaisille käyttäjille.

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Software Engineering
Master's Programme in Software Engineering and Digital Transformation

Joel Salminen

Design of localization web environments

Master’s Thesis

87 pages, 19 figures, 1 table, 3 appendices

Examiners: Professori Ajantha Dahanayake
M.Sc. Eng. Paula Laaksonen, GE Healthcare Finland Oy

Keywords: web-development, localization, usability, maintainability

Localizing software products positively impacts its success in international markets. The goal of this study was to decrease time and cost required to do localization by designing software tools. This study offers design for a web-based environment for different parts of a localization process. The web environment contains tools for inputting and translating user interface texts and reviewing translated documents such as user manuals. The design was evaluated with a usability test. The results indicate that compared to currently existing solutions the new user interface is preferred by the users and that the new features are intuitive enough for inexperienced and infrequent users to learn.

TABLE OF CONTENTS

FIGURES

TABLES

APPENDICES

LIST OF SYMBOLS AND ABBREVIATIONS

ETSI	European Telecommunications Standards Institute
DOM	Document Object Model
FURPS +	Functionality, Usability, Reliability, Performance, Security +
GILT	Globalization, Internationalization, Localization, Translation
GUI	Graphical User Interface
HCI	Human Computer Interaction
IDD	Identifier Dictionary
IFML	Interaction Flow Modeling Language
LISA	Localization Industry Standards Association
LTO	Long-term Orientation
MCR	Modern Code Review
MT	Machine Translation
PD	Power Distance
RQ	Research question
RTS	Regression Test Selection
TM	Translation Memory
UA	Uncertainty Avoidance
UI	User Interface
WebLocKit	Web Localization ToolKit

1 INTRODUCTION

Imported hardware and devices often need to be localized because of local laws. User Interfaces (UI), user manuals, labels and other parts of related equipment need to be fully translated into the language of the target country before they can be exported to the country. As the amount of languages to translate increases, the management of localization becomes a complicated, time-consuming process that requires communication between multiple entities that usually reside in different countries. (Esselink, 2000).

The localization process of GE Healthcare Finland consists of separate software- and user manual translation processes. In the past the most time-consuming part of the software translation process has been preparation of texts before transferring them in batch process between software developers, localization coordinators and translators. Nowadays an in-house developed platform, Web Localization ToolKit (WebLocKit), has provided the software developers a web interface where they can input the software texts directly into the localization database. This has streamlined data transfer between developers and the localization team since localization coordinators can now begin the pre-translation process as soon as new texts arrive (Tolvanen, 2003).

However, there are still opportunities for improvement in other sections of the localization system. For example, transferring English software text to translators is still done one batch, which results in delays in localization projects (Tolvanen, 2003). The goal of this research is to explore if other parts of the localization system can be implemented like the web environment described by Tolvanen. This research offers a design solution for a web-based localization platform that fixes issues of the current localization system. The design has been created with usability and maintainability in mind, as these were identified as the most important quality attributes for the platform.

1.1 Research questions

This study attempts to answer the following Research Questions (RQ):

- RQ1. What should be considered to achieve ease of use for users of an UI

- RQ2. What should be considered when developing platforms that are planned to be expanded by maintenance developers?
- RQ3. What are the changes that the web interface of the current system requires?
- RQ4. With the existing core system in mind, how should a modern web environment be developed to enable more user parties to gain access to a more diverse range of services?

1.2 Research methodology

This thesis begins with a literature study to gain a deeper understanding on the selected topics. Usability was selected as a topic because GE Healthcare wants to keep training needs in minimum and provide ease of use applications to end users, always aiming to improve productivity. Maintainability was selected because of the strategy to keep maintenance resources in economical level. Finally, international interfaces are explored to understand why and how software products can be expanded to international audiences.

The rest of the research will be completed using the design science research method proposed by Peffers et al. (2007). The first phase of this research is problem identification and motivation in which the research problem is specified and reasons for why the solutions would be valuable are provided. These two topics are covered to provide motivation to the researcher and the audience, as why this research is important. The problem definition provides knowledge about the state of the problem which can later be utilized in the design phase.

In the second phase objectives for a solution are defined. Quantitative or qualitative objectives are provided, explaining how the new artifact should perform better compared to the old solution. (Peffers et al., 2007).

The third phase, designing the architecture, follows a typical software development project, including requirements gathering, analyzing the requirements and then designing the new architecture that could be used to create the new artifact (Peffers, 2007).

In the fourth phase demonstration is provided about how the new design solves the problem defined in the problem identification phase (Peffers, 2007). This is achieved by running a usability test with a functional prototype.

Finally, in the evaluation phase the designed artifact will be measured to determine how well the established research problem is solved (Peffers, 2007). This can be achieved by comparing objectives of the research to the actual measured results.

1.3 Limitations

This thesis will only provide a design for a new web environment. Full implementation of the platform will be left out, as it was estimated that a design accompanied by usability studies should be enough to evaluate the validity of the system. Usability literature review only focus issues relevant to computer users. Usability of mobile devices, tablets or other platforms are not considered, because the web platform is designed primarily for desk top computers and laptops in mind. Web development in general is not discussed.

1.4 Structure of the thesis

This thesis is divided into seven chapters. Chapter 2 contains a literature review on the topics of usability, international interfaces and maintainability. Chapter 3 introduces the localization processes of GE Healthcare Finland. Chapter 4 covers requirements gathering methods and design decisions that affected the architecture of the new localization web platform. Chapter 5 describes how the design was evaluated and results of the usability study. Chapter 6 presents discussion, conclusions and next steps for this research. Finally, in chapter 7 this thesis is summarized.

2 LITERATURE STUDY

This chapter consists of literature studies on the topic of usability, maintainability and international interfaces to better understand how the localization web platform should be designed.

2.1 Usability

ISO 9241 standard (1996) defines usability as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use". In literature usability is not considered to be a one-dimensional property of a UI, but instead consisting of multiple subcategories (Nielsen, 1993; Quesenbery, 2001). The goal of this literature review chapter is to find out how usability can be taken into consideration during interface design. This is achieved by reviewing literature recommendations from the point of view of Nielsen's (1993) usability attributes which the author lists as efficiency, satisfaction, errors, learnability and memorability.

2.1.1 Efficiency

Efficiency is the speed in which users can complete a task accurately. It is measured in number of key presses or clicks required for task completion. (Quesenbery, 2001). In a well-designed interface the number of steps required to complete tasks is minimized. (Lane et al., 2005).

Keyboard shortcuts can improve the effectiveness of user interfaces (Cockburn et al., 2014; Quesenbery, 2001; Scarr et al., 2011). Even though the productivity loss that results from individual actions is small, they can accumulate over time resulting in massive amounts of wasted time (Cockburn et al., 2014). In many tasks the hands already rest on the keyboard, and thus, shortcuts eliminate the need for removing hands from the keyboard, selecting a pointing device and finally clicking on graphical icons on the screen. Also, keyboard shortcuts mitigate the need to navigate in menu hierarchies by allowing a great number of commands to be executed using a single keyboard shortcut. (Giannisakis et al., 2017; Malacria et al., 2013). However, if keyboard shortcuts are the only interaction method available, they can also slow down users who are unfamiliar with them or only use the system

infrequently (Quesenbery, 2001). Alternatives to keyboard shortcuts have been proposed (Fitzmaurice et al., 2003; McLoone et al., 2003), but currently they are not widely used as they are situational and required special peripheral devices that are not usually found in office environments.

A well-designed structure can also improve the efficiency of an interface. Needs of the users should to be considered, as hierarchical structure that targets infrequent and novice users can be frustratingly slow for experienced users (Quesenbery, 2001). Flat command structures can make items more accessible, therefore positively impacting users' task completion efficiency (Scarr et al., 2012). Alongside with predictable methods such as spatial consistency, flat hierarchies allow selecting commands with less actions, which makes the goal of reaching high performance levels easier. (Cockburn et al., 2014). Spatial stability helps users to figure out the location of interface elements faster rather than having to rely on relatively slow visual searching (Cockburn et al., 2007).

Compromises between ease of use and efficiency may have to be considered. Interfaces that have been designed for novice users, such as Graphical User Interfaces (GUIs) and wizard interfaces, are easy to learn and memorize for inexperienced users, but expert users don't usually prefer them due to their time-consuming nature (Hwang & Yu, 2011). Moreover, GUIs tend to trap users into beginner mode, where they never learn to use more efficient methods (Cockburn et al., 2014; Lane et al., 2005), thus plateauing at a level of mediocrity. Making an interface method highly visible also needs to be considered carefully. On the positive side, initial performance of novice users is often improved. However, making novice methods more prominent has a risk of impairing adoption of advanced methods such as keyboard shortcuts, which can harm the long-term performance of users. (Cockburn et al., 2014).

Interfaces frequently have opportunities to assist users with automatic tools, for example by correcting user inputted text or by snapping dragged objects to neighboring ones in image editing tools. Surprisingly, most users seem to prefer automatic tools over manually doing everything by themselves even if automatic tools sometimes take actions that require correction by the user. (Cockburn & Quinn, 2016). Other studies have reported similar

results (Koester & Levine, 1994; Koester & Levine; 1996; Quinn & Zhai, 2016), which suggests that ease of use and effortlessness may be more valuable to most users compared to speed of task completion and efficiency.

2.1.2 Satisfaction

Interfaces are engaging if using them is pleasant and satisfying (Quesenbery, 2001). Images, colors, multimedia elements, readability of tables and texts, visual design and visual complexity all influence the user's satisfaction (McLoone et al., 2003; Quesenbery, 2001). Time delays can be frustrating to the users, unless they are predictable, and an accurate time estimation about the duration of the delay is shown to the users (Ceaparu et al., 2004).

Readability of tables can be improved in many ways. Decoration in tables should only be kept to minimum, as it can distract readers from the contents. Table decorations include borders and frames that surround the table, background colors and gridlines. Instead of gridlines, whitespace should be utilized to separate columns from each other. Vertical lines should only be used if the space between columns is so narrow that reading mistakes occur without them. Moreover, stretched columns or columns with even width are best avoided, as this reduces the readability of the contents. Instead, the width of the column should depend entirely on the width of the text within each column. (Rutter, 2017).

Repetition in tables should be avoided by including the repeated information in the headings of columns or by simply removing it. Headings should be aligned with the contents of the columns, text should be aligned to the left side of the table and numerals to the right or to the decimal point. Numerals within a column should use consistent precision and use tabular lining that homogenizes the width of the numeral characters, making them more pleasant and faster to read. If the tables are to be read using small screen devices, the table columns should scroll horizontally, or the users should be allowed to customize the visibility of columns. (Rutter, 2017). Following these rules greatly improves the readability of tables while not distracting the reader with unnecessary decorations.

High readability of texts can reduce users' frustration and shorten task completion time (Miniukovich et al., 2017). Miniukovich et al. (2017) developed a set of 12 core guidelines

to improve readability of texts in web interfaces. The guidelines include recommendation for both average and dyslexic readers. Recommendations for average readers are as follows:

- use short and simple sentences and paragraphs
- avoid complex language or jargon
- reduce the need to scroll by limiting the content on a page
- use off-white background color, for example light gray for background and dark gray for text instead of black.

Together with visual design, visual complexity of an interface can greatly affect the first impression of users (Tuch et al., 2009). GUIs that with low visual complexity often result in in higher user satisfaction due higher perceived accessibility and aesthetics. (Miniukovich et al., 2018). In turn, high visual complexity decreases interaction pleasure and applications cluttered with information can increase the cognitive load of the user (Harper et al., 2009; Tuch et al., 2009).

2.1.3 Errors

In practice errors are unavoidable regardless of how well a software system is designed as systems will always have imperfections and the users of the system make mistakes (Quesenbery, 2001). Time lost to frustrating experiences with computer systems can be as high as 50% (Ceaparu et al., 2004; Lazar et al., 2006), stemming from lacking knowledge or unwillingness to take tutorials or to read instructions (Ceaparu et al., 2004). Errors are often perceived frustrating regardless of whether the source of the error is a mistake made by the user, a design flaw, or an implementation bug. (Ceaparu et al., 2004).

Error tolerant interfaces reduce the chance of user from causing errors when interacting with the system. They help users to recover from mistakes by providing backtracking features, avoiding dead end screens and asking for confirmation whenever necessary. Taking incorrect actions should be made difficult by designing interactable elements to be distinctive and using clear language that the target user group can understand. (Quesenbery, 2001).

Error messages should be used provided to the users when something eventually goes wrong, as users tend to get lost if they make mistakes and the system doesn't give any feedback. Error messages should be precise and human-readable rather than written in obscure code or abbreviations and they provide information to the user about what to do next. (Nielsen, 2001). Visibility and high noticeability are also important. This applies to the error message itself as well as to the location of the error. For example, if a form is filled incorrectly, the system should clearly inform the user which fields require fixing rather than returning the same format without any visible indicators about what went wrong. (Nielsen, 2001; Travis, n.d.). In these cases, as much of the work of the user should be preserved instead of having to begin filling the entire form from the beginning (Nielsen, 2001). Furthermore, polite phrasing is also advised in error messages (Nielsen, 2001; Travis, n.d.).

Sometimes benevolent deception is behavior that deceives users in a way that is beneficial to them. It can be used to hide or recover from errors in a way that potentially increases the usability of a system compared to simply showing an error message. (Adar et al., 2013). For example, under normal circumstances Netflix provides their users personalized recommendations based on their movie watching history. If the recommendation system is offline, the quality of recommendations is degraded by changing the source of recommendations to popular titles instead. In these cases, error messages are not shown to the users and as a result, users get to continue using the system as if everything was working as intended (Ciancutti, 2010).

2.1.4 Learnability and memorability

Interfaces with high learnability allow users to build on their existing knowledge without exerting conscious effort. Users can predict how an unfamiliar system works by using the patterns they've learnt from using other software systems. To minimize the time required to learn a new interface, internal consistency within the interface as well as being consistent to other software systems are important. In consistent interfaces controls are in their expected locations, functios behaves in a predictable way and terminology remains the same throughout the entire interface. (Quesenbery, 2001). Many aspects that make an interface easy to learn also aid with memorability. Memorability is important for infrequent users and those that have not used the interface in a long time. (Nielsen, 1993).

2.1.4.1 Transitioning from a novice to expert user

GUIs have become popular interface types, in part because of their support for novice users (Scarr et al., 2011). In a GUI environment the user can visually scan the interface for features that they are interested in, without having to remember line commands or hotkeys (Shneiderman, 1987). As mentioned earlier, the downside of graphical interfaces is that they tend to have a significantly lower maximum performance when comparing to interfaces designed primarily for experts (Cockburn et al., 2014).

Users of graphical interfaces are easily trapped in the beginner mode, where they use inefficient tools and never improve (Scarr et al., 2011). Carroll and Rosson (1987) coined the phrase "the paradox of the active user" when describing the phenomenon where users plateau at a level of mediocrity, thus never reaching their maximum efficiency. This is said to result from two biases that users of computer systems tend to have. Firstly, production bias refers to people using familiar methods instead of finding improved ways to complete tasks. Secondly, assimilation bias makes people solve new problems by applying their existing knowledge. Because of these biases, both novice and more experienced users continuing with methods that they know to work instead of looking for improvement opportunities. (Carroll & Rosson, 1987). As shown in Figure 1, switching from novice friendly methods to expert ones often results in a temporary dip in productivity that can discourage users learning new, more efficient methods (Scarr et al., 2011). Even if users know or suspect that more efficient ways exist, they are often too focused on their tasks to look for alternative methods or strategies (Carroll & Rosson, 1987).

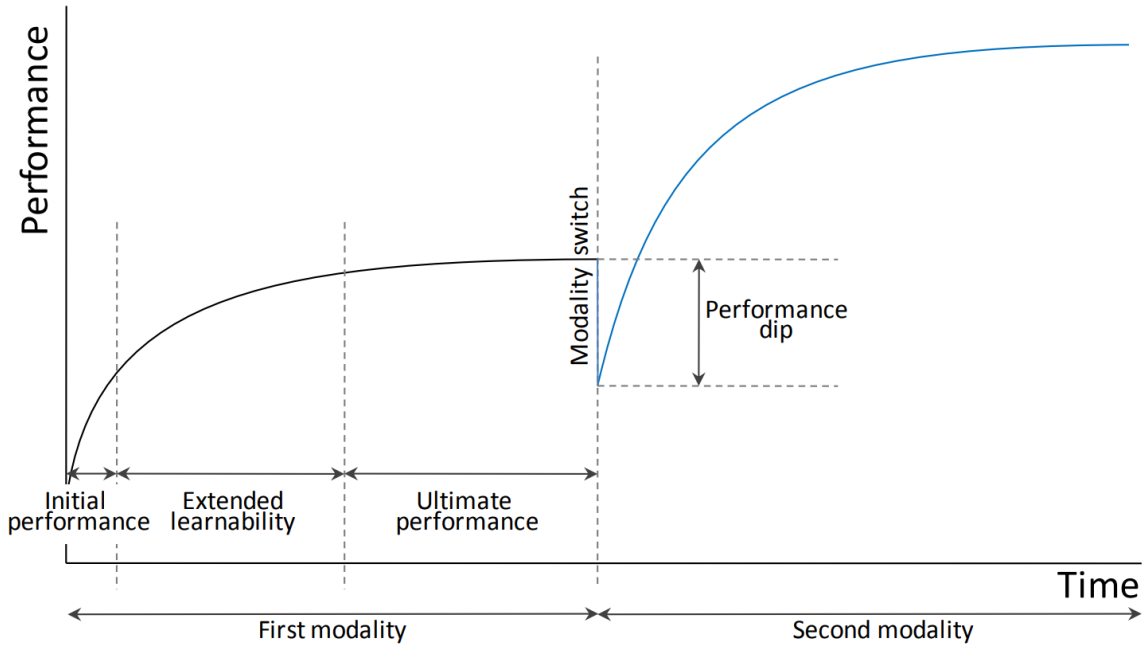

Figure 1. Typical Dip in performance (Scarr et al., 2011).

Despite their efficiency benefits, expert interface components and advanced features like hotkeys are seldom used (Alexander, 2009; Cockburn et al., 2014). Most users know how to use only a fraction of the commands that an interface offers (Cockburn et al., 2014; Matejka et al., 2009). For example, AutoCAD users typically only use 30-40 commands among the thousand commands that are available (Matejka et al., 2009). An often-used method of interface teaching is to provide users with training manuals. However, learners at every lever tend to avoid reading (Carroll & Rosson, 1987) the only exception being older adults (Leung et al., 2012; Mitzner et al., 2008). Instead of reading manuals, new users have a habit of jumping right in and prefer to learn and discover new features while working (Carroll & Rosson, 1987). This suggests that despite their popularity, in practice user manuals may not be the most efficient teaching method.

Current literature suggests various methods to help users to transition from novice to expert users. Flat interface structures with consistent and predictable selection methods allow selecting commands with less actions (Cockburn et al., 2014), thus making the selection process faster even if they user is not using hotkeys. Additionally, skill development can be facilitated by making the novice and expert actions physically similar to each other (Kurtenbach, 1993). For example, if novice users are assisted and encouraged to use hotkeys

at the early stages of the learning process, users need not unlearn beginner methods when transitioning to expert tools. This will reduce the effort required to become an expert use due to the user already being familiar with the methods that are aimed at expert users. (Cockburn et al., 2014).

Recommendations within the system can be utilized to inform the users about faster methods to complete a task as well as to inform them about unused tools. Recommendations should be relevant to the user's context, presented in a way that doesn't break the flow of work and be continuously available. Generated recommendations should be new and useful to the users given their current context and task. (Cockburn et al., 2014). However, while recommendations can be useful, knowledge of more efficient tools might not be sufficient to achieve high performance levels due to users' unwillingness to improve (Carroll & Rosson, 1987).

2.1.4.2 Keyboard shortcuts

While keyboard shortcuts are known to be more efficient compared to selecting commands with a mouse (Malacria et al., 2013), users rarely use any of them, favoring icon toolbars instead (Alexander, 2009; Cockburn et al., 2014; Scarr et al., 2011). There are many potential reasons for this. Learning keyboard shortcuts is often a tedious process, as it often requires taking the cursor to the location where a click would complete the selection, then pausing and waiting for a tooltip to appear and finally committing the key combination to their memory before proceeding (Malacria et al., 2013). Incorrect memory can result in errors and increased time to complete tasks, discouraging users from continuing with the shortcut learning process (Carroll & Rosson, 1987). Selecting a tool from a menu and using a shortcut are physically very different actions and if the user initially learns to use the toolbar, unlearning that habit can be challenging (Grossman et al., 2007). Additionally, keyboard shortcuts are often arbitrary and inconsistent across software applications (Grossman et al., 2007), which is especially harmful to the efficiency of infrequent users (Quesenbery, 2001).

Many of these difficulties originate from the fact that keyboard shortcuts are not as accessible as toolbar buttons that can be used by simply pointing and clicking and don't require memorization of commands. Keyboard shortcuts are usually located in separate menus, user

manuals, cheat sheets or inside a tooltip that is only shown when hovered over with a mouse (Giannisakis et al., 2017). These teaching methods have been proven to be ineffective, because shortcuts displayed in dropdown menus tend to be ignored by users (Grossman et al., 2007) and users at all levels avoid reading manuals (Carroll & Rosson, 1987). In literature various experimental keyboard shortcut teaching methods have been proposed (Giannisakis et al., 2017; Grossman et al., 2007; Krisler and Alterman, 2008; Malacria et al., 2013). However, none of these methods have become widely adopted in practice.

2.1.4.3 Evolving interfaces

It is widely known that breaking difficult tasks down into smaller, manageable subtasks aids the learning process of new skills. Difficult to master skills such as playing musical instruments or speaking foreign languages are taught in carefully designed steps that introduce important concepts one at a time. This gives the learner enough time to master one aspect before moving on to the next learning challenge. Most software systems, however, do a poor job of teaching the users how to use the interface (Shneiderman, 2003). Up to 50% of the time spend on using software is wasted because of confusing menus and difficult to find functionality (Ceaparu et al., 2004). First time users are often confused by feature rich interfaces they've never used before because of the overwhelming amount of information and tools available, not knowing what to focus on first (Carroll & Rosson, 1987).

To solve the issue of poor learnability of interfaces, designs that gradually introducing the users to new functionality have been proposed. Shneiderman's (2003) multi-layer interface divides the interface into layers between which the users can switch freely. Separating the interface into layers helps users to learn to use new functionality in sequential manner, while reducing the overall complexity of the interface. Beginner users could start from the first layer and move on to more advanced layers after they've had time to get familiar with the functionality offered by the first layer. (Shneiderman, 2003).

The structure of layers in a multi-layer interface should be carefully considered. The layers could be structured sequentially or semi-sequentially, allowing users to decide which layers to include at later stages of learning. However, it can be challenging to identify the logical sequence of features, notably if there are user groups with different needs. (Shneiderman,

2003). Moreover, additional complexity resulting from switching layers can overwhelm novice users, resulting in a performance dip (Cockburn et al., 2014; Leung et al., 2010).

Findlater et al. (2009) proposed a similar solution. Their ephemeral adaptive interface is based on predicting which functionality is the most relevant to the user while limiting the visibility of other tools. When the user opens a menu, items that are predicted appear instantly while less important items fade in gradually, drawing the user's attention to the predicted objects first. The space that the menus occupy remain the same to keep the interface consistent over time, allowing users to build habits that they can use even as advanced level users. According to initial test results, ephemeral adaptive interfaces are faster than regular interfaces when the prediction is accurate, but not significantly slower the predictions fails. (Findlater et al., 2009). Predictions should be created carefully due to predictability of adaptive interfaces having an impact on user satisfaction (Gajos et al., 2008). The downside of increasing the visibility of novice-friendly functionality is that as a result expert functionality is often suppressed, which reduces the likelihood that they will be discovered (Cockburn et al., 2014).

The multi-layer and ephemeral adaptive interfaces differ from each other by the fact that the former approach gives the users full control of the learning process, while the latter does not. Multi-layer designs have improved learnability and helped the users to retain the ability perform tasks in the tested software environment. Transitioning into a full functionality layer can temporarily decrease task performance, but no negative impacts were found when learning advanced tasks. (Leung et al., 2010). Systems with experimental teaching features like these need to be developed carefully and usability tested before they are used in practice (Shneiderman, 2003).

2.2 International interfaces

The appeal of a software applications that are designed for international audiences can be greatly increased through Globalization, Internationalization, Localization and Translation (GILT) (Esselink, 2000). This chapter contains a literature review about the different sectors of the GILT model and explores how it can be extended by cultural consideration.

2.2.1 Globalization

In literature the term globalization is used in different ways. According to Localization Industry Standards Association (LISA), *"globalization addresses the business issues associated with taking a product global. In the globalization of high-tech products this involves integrating localization throughout a company, after proper internationalization and product design, as well as marketing, sales and support in the world market*". (Esselink, 2000). Globalization is often defined as an umbrella term related to modifying or designing products for global audiences (Salgado et al., 2014). Globalization consists of the other aspects of GILT (Byrne; 2009; Pym et al., 2006; Salgado et al., 2014), as shown in Figure 2.

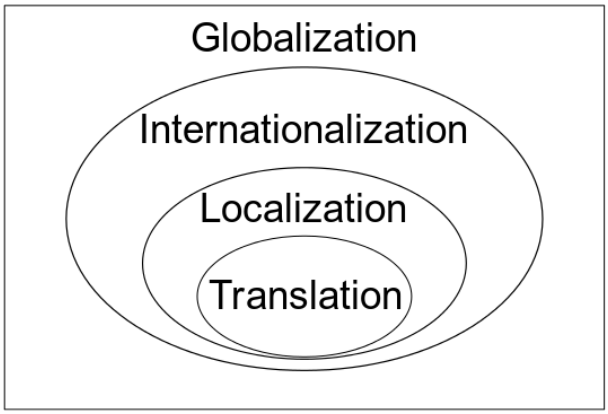

Figure 2. GILT process, redrawn from Pym et al. (2006).

2.2.2 Internationalization

European Telecommunications Standards Institute (ETSI) (2007) defines internationalization as "the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for redesign". In other words, internationalization is about designing and creating a software base which can then be modified for various audiences and markets worldwide (Salgado et al., 2014).

The benefits of internationalization are numerous. An internationalized product is capable of dealing with a multitude of languages and cultural conventions and expanding them to new locales is less effortful. (Esselink, 2000; Shen et al., 2006). Internationalization can

lower development and maintenance costs when adding new features or fixing bugs and minimize the required time to reach users (Shen et al., 2006). Internationalization makes the localization process easier for both the development team as well as the localization vendor, which reduces the cost of localization (Esselink, 2000). Furthermore, designing a product with internationalization in mind is less expensive and requires less time than re-engineering existing software (Esselink, 2000; LISA, 2008).

In practice, during internationalization all culturally specific elements are isolated from a product so that they can later be replaced with localized versions (LISA, 2008; Russo & Boor, 1993). These elements include text, functionality, date and time formats, images, icons, and even colors (Russo & Boor, 1993). When it comes to text in interfaces, enough space should be reserved in the interface so that the texts will fit even when they are translated. This is because translated sentences are typically up to 30% longer than English sentences, while individual translated words may be as much as 100% longer. (Esselink, 2000). An internationalized software should be able to handle international character sets (Nielsen, 1990) and language used in texts should be jargon free, consistent and include short sentences and simple vocabulary (Esselink, 2000). The flow of text can also vary, as in Western languages text typically flows from left to right, but in languages such as Arabic or Hebrew the flow of text is reversed. Translation of text strings can be made easier during internationalization by externalizing the strings into resource files, because this allows localizers to create translations without changing the code. Additionally, the change of missing translatable strings is reduced which improves the quality of the product. (Esselink, 2000).

Formats of numbers, dates, time, currency and telephone numbers vary between languages. For example, numbers and currencies can have different separators for thousands, the sequence of day, month and year can vary in dates, time format can be either 12 and 24 hours and time zones can be different depending on where the user is located. In an internationalized interface same format should be used for inputting and displaying text and it should be a format that is familiar to the user. (Nielsen, 1990).

Icons can have different meanings in different cultures and as such they must be considered carefully (Esselink, 2000; Nielsen, 1990). For example, a cocktail glass that signifies after-work events would be inappropriate in locales where alcohol is not associated with social activities (Nielsen, 1990), such as in some states in India where sale of alcohol is prohibited (The Indian Express, 2016). International standards should be checked to make sure that icon designs will not conflict with the local conventions or existing icons. Moreover, inclusion of text in icons should be carefully considered. Text within icons is difficult to translate and they seldom take the same amount of space in all languages. However, text in icons can increase their usability, which is why in some cases the inclusion of text can be worth the added difficulty in translation. (Nielsen, 1990). Some evidence suggests that users tend to ignore unlabeled icons, which results in the functionality behind those icons also to be ignored (Harris, 2005).

Preferred functionality of the software can also be locale specific. An example of this is collating sequences that define the order of sorting. For example, in the collating sequence of German the letter “ä” comes after the letter “a”, but in Swedish the letter “ä” is near the very end of the sequence. In some languages strings with two characters can be treated as a single character, such as “Il” in Spanish while in other languages single characters can be treated as two characters. An internationalized interface the collating sequence can be changed based on the user’s locale or preferences. (Nielsen, 1990).

The degree of required internationalization always depends on the languages into which the product will be localized. For example, English is easier to localize into German or Spanish than to Asian languages or Arabic as the latter languages require special planning and design considerations. (LISA, 2008).

2.2.3 Localization

According to Esselink (2000), localization is “the translation and adaptation of a software or web product which includes the software application itself and all related product documentation” For each of the target locale a localized copy of the original software is created using the previously internationalized base design (Esselink, 2000). Locales don’t always perfectly match with languages or borders of countries, as in countries like

Switzerland multiple languages can be spoken in a country. Moreover, even if the same language is spoken in different countries, they can still be considered different locales. For example, French is spoken in France, Switzerland and Canada, but all of them are different locales that require their own localization versions. (Esselink, 2000; Nielsen, 1990).

Localization allows users to interact with a software interface using their first language, which can increase its chances of succeeding in global markets (Esselink, 2000). Fully localized products address a wider audience, which increases the usability of the software and makes it better accepted and more accessible (Carey, 1998). Studies have indicated that when compared to original software interfaces, users have positive attitudes towards localized interfaces (Ford & Gelderblom, 2003; Nantel & Glaser, 2008). Users in certain locales such as in Japan or France may not tolerate interfaces that have not been fully localized to their language (Carey, 1998).

Before localization can begin, a decision about the completeness of localization needs to be considered. According to Carey (1998) the levels of localization are:

1. Translating nothing
2. Translating only packaging and documentation
3. Enabling code
4. Translating dialogs and software menus
5. Translating tutorials, online help other supplementary material
6. Adding locale-specific hardware support
7. Full customization of features

Later levels of localization include all activities of the previous levels. The more complete the localization of a product is, more time-consuming and riskier the localization is going to be. However, the benefits of localization are also greater in fully localized products when compared to only partially localized products. Sometimes full localization may be the only option, as some government agencies will refuse to purchase software that isn't fully localized. Partial localization is sometimes acceptable, for example if the size of the target audience is small or if it is known that the target audience will tolerate an English version of the product. (Carey, 1998).

Designing sophisticated localized versions of software can be very expensive (Höök, 2000) and often extensive ethnographic analysis is required for each locale (Yeo, 1996). ETSI recommends considering localization at the beginning of the software planning process because of its expensive nature. As quality assurance tests need to be performed for all supported language versions of the software, testing a localized software system often also requires a lot of resources. (ETSI, 2007). To reduce the costs and time requirements of localization, projects can be run in parallel alongside with software product to later simultaneously launch all language versions (Esselink, 2000). Additionally, testing content in the original language prior to localization can significantly reduce the cost of localization (ETSI, 2007).

Conventional localization has been criticized in literature. Localized interfaces typically differ only in language and a trivial visual aspects (Reinecke et al., 2013), as localization often only considers date or time formats and language of the target locale (Kersten et al., 2002). Localization of images is often left out (Esselink, 2000; Reinecke et al., 2013). Additionally, UIs design is usually done with merely a small number of target cultures in mind. This is because the process of designing specialized interfaces for every cultures is expensive in terms of costs and time. Additionally, localization doesn't take into account the cultural ambiguity of some users, as versions are often limited to one locale with no options to combine features of different localized versions. This can be bothersome to users who have been exposed to multiple cultures and would prefer to mix elements from different versions. (Reinecke et al., 2013).

2.2.4 Translation

Translation is the central activity of localization and it includes translation of all GUI elements of a software, for example menus, loading tips and errors. Most localization projects start with the software translation, as documentation often includes references to the software interface and translating documents before the interface components would lead to inconsistent translations. (Esselink, 2000).

Various software tools have been developed to improve productivity of translators and to reduce translation costs. Machine translation (MT) refers to computers automatically

translating from a natural language to another (Lopez, 2008). Despite a lot of effort being put into MT research, even state-of-the-art MT systems such as those that use machine learning systems (Lopez, 2008) are too error-prone to be used without human post-processing (González-Rubio & Casacuberta, 2014; Vidal et al., 2006). Despite these limitations, MT can still be utilized to increase productivity of human translators by using it in computer assisted translation (Vidal et al., 2006).

A translation Memory (TM) contains source-translation pairs with the idea of recycling translations inside a TM tool. TM tools can be used to manage the translation process by allowing translator to view source and target strings and by creating and saving source and target pairs during translation automatically. Previously translated segments can then be proposed to the translator in the case of reappearance. (Moorkens, 2015). TMs are crucial in translation as they increase both translator productivity and translation quality (Lagoudaki, 2006), and reduce inconsistency by enabling recycling of previously translated work, thus reducing the overall cost of translation work (Moorkens, 2015; Olohan, 2011).

Termbases are used to store source texts, approved translations and context information. The main difference between TMs and termbases is that data stored in a termbase must be approved before storing, while TM source-translation pairs are stored automatically during translation. Termbases can help eliminate ambiguity and improve text quality through consistent usage and spelling of brands and corporate terms. Termbases also inform translators about phrases that ought not to be translated, such as brand and product names. (Helmut, 2016).

2.2.5 Beyond localization: intercultural design

Since localized interfaces often only differ in language aspects (Reinecke et al., 2013), conventional internationalization and globalization can be extended by considering target cultures in more detail. Studies suggest that culture can significantly influence how users interact with interfaces (Malinen & Nurkka, 2013) and people belonging to the same cultural groups tend to process and perceive information similarly (Reinecke et al., 2010). For example, people in the same cultural group have been demonstrated to exhibit similar navigation behavior (Kralisch & Berendt, 2004; Kralisch et al., 2005). Culturally adapted

interfaces have also been recognized to improve working efficiency (Reinecke & Bernstein, 2007).

Frameworks to recognize cultural differences have been published by several authors such as Schwartz (1999), Trompenaars & Hampden-Turner (2011) and Lewis (2011). Most Human Computer Interaction (HCI) research regarding cultural considerations, however, is built upon Hofstede's cultural dimensions (Jaakkola & Thalheim, 2014). In Hofstede's model the most dominant culture of each country is described in terms of cultural dimensions. Cultures are scored in each of the dimensions based on how well the dimension describes the culture. (Hofstede & Hofstede, 2004; Hofstede et al., 2010). The six cultural dimension in Hofstede's model are:

1. Power Distance (PD), the extent to which inequality in power tolerated in social hierarchies;
2. Individualism vs. collectivism, the degree to which people control their impulses and desires;
3. Masculinity vs. femininity, the degree to which a culture separates traditional gender roles;
4. Uncertainty avoidance (UA), the extent to which a culture is comfortable with uncertainty regarding future events;
5. Long-term orientation (LTO), the extent to which delayed gratification of needs are accepted;
6. Indulgence vs. restraint, acceptance of enjoying life and having fun vs. controlling the life by strict social norms.

Cultural dimensions can help in the design of user interfaces. Cultural dimensions can be used for generating ideas and justifying intuitions as well as personal experiences (Yaaqoubi & Reinecke, 2018). Cultural dimensions provide guidelines about how a national culture of a country may compare to another and utilizing them in the localization process can reduce the need for time-consuming and expensive cultural studies (Price et al., 2014). However, while utilization of cultural dimensions may help reduce research costs related to cultural

differences, development of culturally adaptive interfaces is extremely time-consuming and prohibitively expensive (Reinecke & Bernstein, 2007).

2.3 Software maintainability

Software maintenance has been defined as “modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment” (IEEE Standard Glossary of Software Engineering Terminology, 1983). The estimated cost of software maintenance has been increasing since the 1970 and nowadays the maintenance phase alone can require more than 90% of the total expenses of a project (Koskinen, 2015). Because of the costly nature of software maintenance, maintainability of software is very important as system that is highly-maintainable is also easy to correct and expand to satisfy new requirements (Martin & McClure, 1983).

Maintaining a software system that was designed without maintainability in mind is difficult (Schneidewind, 1987). This chapter contains a literature review about software maintainability to understand how maintainability of software systems can be improved. In the following subchapters readability of source code, refactoring and regression testing are discussed.

2.4 Source code readability

Software code readability is believed to strongly influence the maintainability of software systems (Aggarwal et al., 2002). Most maintenance work involves making changes to source code that was originally written by someone else. Before implementing fixes and new features maintenance programmer must read and understand the code. (Lionel & Deimel, 1985). Poor readability of source code hinders programmers from understanding what the code is doing (Mannan et al., 2018), which in turn reduces developer productivity (Deissenboeck & Pizka, 2006). It can cause developers to make introduce bugs and to make less than optimal changes (Mannan et al., 2018). According to Buse and Zimmermann’s study (2012), readability is one of the top three information needs of programmers, as up to 90% of developers would use readability metrics if they were available.

2.4.1 Identifier names

Selecting identifier names can have a substantial effect on software readability, as approximately 70% of the source code consists of identifiers such as variable, function and

class names. Virtually all programming languages allow programmers to use any sequences of numbers and characters as identifiers, which can negatively impact the readability of the code if the identifiers are arbitrarily selected. Despite the importance of identifiers, many developers neglecting them. This is because in projects involving multiple developers, everyone has only a limited knowledge about the identifiers that are used in other parts of the system. Furthermore, as the software system evolves, identifiers can become obsolete as the concepts that they refer to become altered or even abandoned. (Deissenboeck & Pizka, 2006).

Deissenboeck and Pizka (2006) created guidelines for identifier naming that aim to provide easier to understand identifiers and reduce complexity. The guidelines suggest avoiding homonyms (words with identical spelling, but different meaning) and synonyms (words with different spelling but same meaning) when selecting identifier names. Identifiers should also be concise, that is they should describe the concept that they represent tersely but also as accurately as possible. Using names that are too general should be avoided, as there is a risk that the same name could be used differently in a different context. (Deissenboeck & Pizka, 2006). For example, in the case of selecting a variable name for a file path, *absolutePath* would follow the conciseness guideline, as it efficiently describes the concept while distinguishing itself from other similar concepts such as relative path names.

Full-word identifiers and carefully considered abbreviation lead to better code comprehension than arbitrary abbreviations and single character identifiers (Lawrie, 2007). With modern development environments additional typing effort can be minimized using autocompletion tools (Hofmeister, 2019). However, the use of excessively long identifiers is not recommended, since they might overload the short-term memory of developers, which in turn can negatively impact code comprehension (Lawrie, 2007).

The need to improve the readability of identifiers has been recognized by both researchers and practitioners (Kim & Kim, 2016). Several theoretical approaches have been suggested to solve this issue, such as Identifier Dictionaries (IDDs) (Abebe et al., 2009; Deissenboeck & Pizka, 2006). IDDs are databases that store identifier names, the type of the object being identified and a comprehensive description. IDDs allow maintainers to lookup concepts and

then locate the corresponding identifiers in the source code, which increases maintainability by speeding up the source code comprehension process (Deissenboeck & Pizka, 2006). However, false positives are still quite common in these approaches (Kim & Kim, 2016), which may be a reason why they haven't become widely adopted in practice.

2.4.2 Coding standards

A commonly used approach to improving readability is coding standards and guidelines. Coding standards aim to homogenize the code base of a software system by ensuring a common programming style and limiting the use of problematic structures (Boogerd & Moonen, 2008; Boogerd & Moonen, 2009). Coding standards are usually based on opinions of experts and they can target different goals such as maintainability, portability reliability (Boogerd & Moonen, 2008). According to literature, adhering to coding standards can enhance team communication (dos Santos & Gerosa, 2018; Li & Prasad, 2005), improve code quality (Li & Prasad, 2005) and increase maintainability and reliability (Boogerd & Moonen, 2008; Boogerd & Moonen, 2009). Coding standards may also prevent faults and errors (Boogerd & Moonen, 2009).

Coding standards have been criticized for often containing rules that are too generic (Boogerd & Moonen, 2008), too difficult to enforce (Deissenboeck & Pizka, 2006) and only applicable in certain contexts (Boogerd & Moonen, 2009). Having to rely on rules that can only be applied in certain contexts hurts the accuracy of automatic checking tools that are designed to help users follow coding standards (Boogerd & Moonen, 2008). As a result, coding standard compliance tools typically have high false positives rates that range from 30-100% (Kremenek et al., 2012). Coding conventions have been said to only address superficial features such as identifier names while semantic properties are left unmentioned (Hofmeister, 2019). Moreover, there is a lack of substantial evidence that many rules present in coding standards reduce the number of faults. Boogerd & Moonen (2008) claim that compliance to unnecessary rules can make the software less reliable and increase the number of faults in software.

2.4.3 Code reviews

Besides controlling quality of code (d'Astous et al., 2003), increasing learning, finding alternative solutions (Bacchelli & Bird, 2013) and defects (Boehm & Basili, 2001; Shull et al., 2002) and discovering security bugs (McGraw, 2004), code reviews can also be used to improve code quality by making sure that the code is written according to coding standards (Bacchelli & Bird, 2013, d'Astous et al., 2003). Traditional software inspections are rigidly structured, formal reviews that involve face-to-face meetings (Fagan, 1976). Code reviews in traditional style have been said to be difficult to adopt in nowadays rapidly releasing software projects (Votta, 1993). They are also expensive over a long period of time, as they require simultaneous attendance of all team members, sufficient preparation, efficient moderation, and readiness of the work product for review (Russel, 1991).

Because of limitation of the traditional code reviews, today's software projects use a lightweight, tool-based approach often referred to as Modern Code Review (MCR). The goal of MCRs is to improve the efficiency of review processes and to make them less time-consuming. (Bacchelli & Bird, 2013). MCRs do not require review in-person meetings or checklists (Shimagaki et al., 2016) and they often reduce the number of involved people to two (Beller et al., 2014).

2.5 Refactoring

According to Fowler et al. (1999), refactoring means changing the internal structure of a software system without changing its behavior. While compilers aren't concerned about cleanliness of source code, to software maintenance programmer refactoring is important as it can increase the readability of the code. The goals of refactoring can include making the code cheaper and less time-consuming to modify (Fowler et al., 1999), mitigating design degradation, aiding bug fixes and development of new features and reducing maintenance effort and reducing maintenance effort (Bavota et al., 2015; Kim et al., 2012; Silva et al., 2016). It can also be used to understand code that the reader is not familiar with (Fowler et al., 1999). Refactoring can decrease time required to fix bugs (Kim et al., 2011) and positively impact the quality of code (Ratzinger et al., 2008). Compared to regular changes in a large software system, refactoring changes can be more reliable (Kim et al., 2012).

While refactoring is commonly used in practice (Murphy-Hill et al., 2009) many practitioners perceive it as risky and involving substantial costs (Kim et al., 2012). Refactoring requires finding a balance between performance and code simplicity, because changes that make the source code easier to understand might cause the program to run more slowly (Fowler et al., 1999). Incomplete refactoring can result in long-lasting bugs that are not visible in compiler errors, making these faults more difficult to locate (Görg & Weißgerber, 2005). Developers tend to document refactoring incorrectly (Stroggylos & Spinellis, 2007). Moreover, changes that developers make during refactoring often do not improve the quality of software structure. Even worse, refactorings with negative impact seem to be just a common as refactorings with positive impact. (Cedrim et al., 2016).

Locating issues that refactoring could fix is a well-studied topic in literature. Kent Beck invented the term code smell as "quality issues that can be refactored to improve the maintainability of software" (Fowler et al., 1999). Typical examples of code smells are duplicate code, long parameter lists, large classes (Fowler et al., 1998) and spaghetti code, "unmaintainable and incomprehensive code without any structure" (Brown et al., 1998). Code smells described as suboptimal or poor solutions (Khan & El-Attar, 2016; Khomh et al., 2011; Sousa, 2016) and they can be indicators of significant design problems (Moha et al., 2010; Sousa, 2016; Yamashita, 2014). Code smells often violate best practices (Sharma et al., 2016) and negatively impact quality and maintainability of the source code (Khomh et al., Yamashita, 2014). Sharma and Spinellis' (2018) survey revealed that code smells have an impact on reliability, performance and modifiability of the software. Additionally, code smells can reduce comprehensibility of source code (Abbes et al., 2011). Code smells are sometimes used synonymously with anti-patters (Sharma & Spinellis, 2018), although some authors define these two concepts differently (Brown et al., 1998; Hallal et al., 2004).

There are various reasons why smells are introduced in source codes. One of the major causes is developers lacking technical skills and being unaware of the importance of code quality (Curcio et al., 2016; Martini et al., 2014). Time pressure such as penalties for delayed deliveries or prioritization of new features over quality can encourage developers to take shortcuts, resulting in code smells and technical debt (Lavallée & Robillard, 2015; Martini et al., 2014). Frequently changing requirements (Lavallée & Robillard, 2015; Martini et al.,

2014), knowledge transfer issues due to poor documentation (Lavallée & Robillard, 2015) and ineffective or missing processes (Tom et al., 2013) have also been reported to cause code smells.

Various automatic tools and methods have been suggested to find and remove code smells with the goal of improving software maintainability (Palomba et al., 2014; Sharma & Spinellis, 2018). However, these detection tools typically suffer from large amounts of false-positives (Fontana et al, 2016; Moha et al., 2006; Pantiuchina et al., 2018). According to Pantiuchina et al (2018), "only one or two classes [out of hundreds] will become smelly in the future, and the real challenge is to identify them without flooding developers with a high number of false positives". Additionally, smell detection tools do not take context into account (Sharma & Spinelli, 2018), and out of context code metric values have been called meaningless (Gil & Lalouche, 2016).

Automatic smell detection becomes even more complicated if detection of code smells is considered a subjective process, as suggested by some authors. Some patterns that are described as code smells in literature aren't always seen as design problems, but as conscious choices made by developers. (Palomba et al., 2014). For example, studies show that developers consider the prevalence of God classes an important threat to code quality (Palomba et al., 2014), but at the same time junior programmers have been observed to "work better on a version of a system having some classes that centralized the control, i.e., God classes" (Olbrich et al., 2010). In the case of Linux Kernel, God classes are not considered as design or implementation issues (Jbara et al., 2014). This is because according to Jbara et al (2014), fragmenting complex functions into smaller ones would improve the complexity metrics of the Linux Kernel code, but it might also degrade the code quality, thus making it more difficult to maintain.

2.6 Regression testing

Regression testing has been defined as "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements" (ISO/IEC/IEEE 24765, 2010). In

other words, the purpose of regression testing is to avoid accidental changes in unchanged parts of the software when new modifications to the software system are implemented. (Bourque & Fairley, 2014; Yoo & Harman, 2012). Regression testing is important during maintenance as it improves maintainability by allowing maintenance programmers to make changes without unknowingly breaking other parts of the code, (Bourque & Fairley, 2014). Automated regressions tests are also a key part of refactoring (Fowler et al., 1998).

Regression testing is often done automatically utilizing unit tests that were created during the implementation of the software. Test automation can be beneficial in various ways, including improved product quality, reduced testing time and human effort and better reusability of tests compared to manual testing (Rafi et al., 2012). However, automated tests should not entirely replace manual testing because all testing tasks are not easy to automate (Berner et al., 2005; Karhu et al., 2009). According to a practitioner survey, automated test should be used as a completement to manual testing to reduce the amount of mundane testing work (Rafi et al., 2012). Additionally, it should be noted that just like any other part of a software system, test automation also must be maintained for it to remain relevant, which increases the maintaining workload of maintenance programmers (Berner et al., 2005; Karhu et al., 2009).

Several methods for picking out test cases in regression testing have been developed. The strategy is the retest-all approach, where all tests are run every time a change is implemented (Rothermel & Harrold, 1996). However, retesting everything in large-scale projects can be very expensive (Gupta et al., 2011). Executing all test cases within a short amount of time becomes increasingly difficult as the size of the software and thus the number of test suites increases (Garg et al., 2012). For example, Elbaum et al. (2000) reported that it would take seven weeks to run the entire test suite of one of their products. Other regression test selection methods rely on reusing parts of the development test suite, as it reduces the cost of regression testing (Rothermel & Harrold, 1996). Aside from retest-all method in small projects, the three most commonly used techniques for regression testing are Regression Test Selection (RTS), test suite minimization, and test case prioritization (Yoo & Harman, 2012).

RTS techniques "reuse test cases from an existing test suite to test the modified part of the program" (Rothermel & Harrold, 1996). To minimize the required time to retest a changed software system, a batch of test cases is selected and used to test the changed parts of the software (Rothermel & Harrold, 1996; Yoo & Harman, 2012). While test suite minimization works by using metrics that are measured from a specific version of software, in RTS test cases are chosen because of their relevancy to the modification between the current and previous and software versions (Yoo & Harman, 2012). RTS techniques allow developers to locate errors early in development and are especially effective when the changed software is frequently tested (Kim et al., 2000). However, while RTS has been broadly studied (Yoo & Harman, 2012), it has not been widely adopted in practice (Gligoric et al., 2015) because of lowered fault detection ability (Rothermel et al., 1998) and lack of support for automated, language-neutral RTS tools (Gligoric et al., 2014; Harrold et al., 2001).

In **test suite minimization** and test suite reduction try to find test cases that could be eliminated from the suite of tests without substantially reducing its fault-detection rate (Rothermel et al., 2002; Shi et al., 2014). Empirical studies have provided evidence that minimization techniques produce considerable savings in test suite size (Rothermel et al., 1998). However, results of studies that have tested the impact of minimization on fault-detection capability been inconclusive. Some studies suggest that the impact of minimization is insignificant (Wong et al., 1998; Wong et al., 1999), while other studies have provided contradicting results (Rothermel et al., 1998).

In **test case prioritization** the test cases are sorted in the order of effectiveness to provide maximum benefit even if the execution of tests is interrupted (Rothermel et al., 2001; Wong et al., 1998). The aim is to identify flaws as quickly as possible while avoiding the lowered fault detection rates caused by RTS and test case selection techniques (Yoo & Harman, 2012).

3 LOCALIZATION IN PRACTICE

Tolvanen (2003) described the software localization process of Datex-Ohmeda, company nowadays owned by GE. Their software localization process begins at the engineering project level where software developers collect strings from interfaces of medical devices and input them into the localization system as text items. The source English texts and their translations are prepared by localization coordinators. The translations are then done by native translators of the target language. Finally, the translated software texts are post-processed and reviewed by appropriate parties before the texts are exported back to the software source code. (Tolvanen, 2003). The software localization process is illustrated in Figure 3.

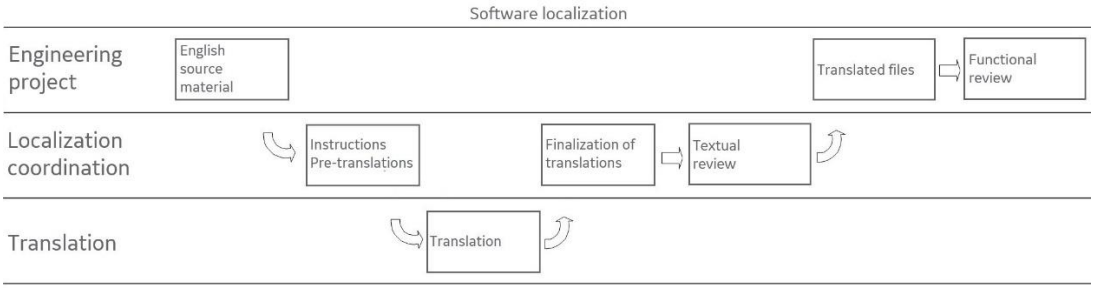

Figure 3. Software localization process at GE Healthcare Finland

The process of document translation at GE Healthcare differs from the software translation process in a few aspects. First, source English contents are provided by technical writers. Much like in the software translation process, English materials are prepared for translation by the localization coordinators. The prepared documents are then translated and reviewed by appropriate parties before they are finally delivered to engineering projects. Figure 4 visualizes the phases of the document localization process.

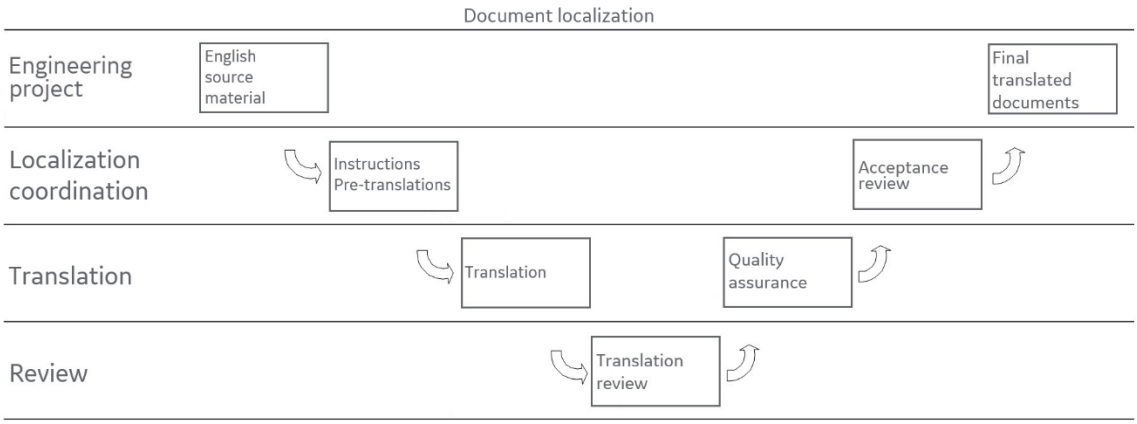

Figure 4. Document localization process at GE Healthcare Finland

Opportunities for improvement have been identified in the localization processes. Currently software translation tools do not allow translators to directly edit the translations in the localization database (Tolvanen, 2003). Because of this the localization coordinators must send translatable English software texts to translators in batches. This has resulted in inconveniences and delays in translation, as the translators must wait until all the texts in a batch have been prepared before they can begin translating. Similarly, localization coordinators must also wait until the all translation packages of a software localization project have been finished and received before post-processing on translations can begin. In addition to the agility issue, working with packages has been burdensome to both above-mentioned parties due to issues related to large package sizes, formatting and different operating systems.

Estimating required to complete translations is especially important as translation is often the most time-consuming part of the localization process because of volume of translatable materials. However, current tools also do not make the progress localization projects visible for the localization team. Because of this, estimating time to complete localization projects has been challenging. Additionally, improvement opportunities in usability of the interfaces of the localization tools have been noticed.

The goal of this thesis is to provide solutions to these issues. A decision was made to facelift and expand upon Tolvanen's (2003) WebLocKit which is nowadays used to input English software text items into the localization system and to create automated event and status

reports. Adding software translation tools to the web environment will allow the translation to begin as soon as some English text items have been prepared by the localization coordinators. Document review tools in a web environment will provide the localization team up-to-date information about the progress of document translation and reviews. The goal of these changes is to improve efficiency and satisfaction of translators, reviewers and localization coordinators. Moreover, using a web environment bypasses many IT issues related to installations users will only need a modern web browser. Finally, usability issues related to current tools can be analyzed and fixed during the design process.

4 REQUIREMENTS AND DESIGN

This chapter explains the requirements elicitation process for the new web platform as well as the design decisions based on the requirements.

4.1 Requirements elicitation

Requirements were gathered using various sources and elicitation methods. Initial requirements were gathered by analyzing existing localization software and their user manuals for ideas and inspiration. Analyzed software included in-house developed tools, translation tools commonly used in the translation industry, current version of WebLocKit and document review tools developed by the translation companies. The observations database of the localization team was also utilized for improvement ideas, as it includes concepts for functionality suggested by the current userbase of the localization tools.

After initial requirements gathering preferences of users for the new system were collected. Software developers and technical writers were interviewed using semi-structured interviews as described by (Runeson & Höst, 2009). As translators and document translator reviewers were planned to become one of the core users of the new web platform, collecting their preferences was of utmost importance. However, interviewing the them was not feasible as they typically work all over the globe. Because of this, an online survey was conducted to better understand the user needs regarding software translation and document review tools. The survey was designed according to survey guidelines of Kitchenham & Pfleeger (2002). Requirements were organized based on what tool they belong to as well as into functional and non-functional requirements using the Functionality, Usability, Reliability, Performance and Security+ (FURPS+) model (Umar & Khan, 2011). Examples of requirements for the software text input tool are shown in Appendix I.

4.2 Architecture

This chapter explains the architecture of the new web environment in terms of expected functionality, interface design and technical details. Instead of building on the existing web environment, a decision was made to re-design and implement the new WebLocKit from the

beginning using a modern JavaScript frontend framework. This is because the old version of WebLocKit was implemented using Java Server Pages technology (Tolvanen, 2003) and it was estimated that re-designing and updating the current web environment would take more effort than re-engineering the system. The new design utilized the already existing localization database. Despite resulting in some unavoidable design limitation, this was ultimately the only option because the web platform will need to be able to access same data as the other localization applications.

4.2.1 Functional design

Figure 5 portrays the use case diagram that was created based on requirements and the needs of the end users. Six different kinds of tools were identified from use cases: text input, software translation, document translation review, document management, report download and event report creation. User groups will only have access to tools that are relevant to their work to reduce the chance of unauthorized or unintended changes in localization data.

Text input tool is used by software developers to manage source English text items that contain data about English strings collected from medical device user interfaces. The text input tool shows what the text will exactly look like in the interface of the target device. This allows software developers to make sure that the strings will fit in the original device interface. As a new feature users will be able to manage text items with a common maximum width, referred to as flexible texts. All modifications to text items are stored in the localization database which enables the use of a change log where authorized users will be able to track changes made to an individual text item.

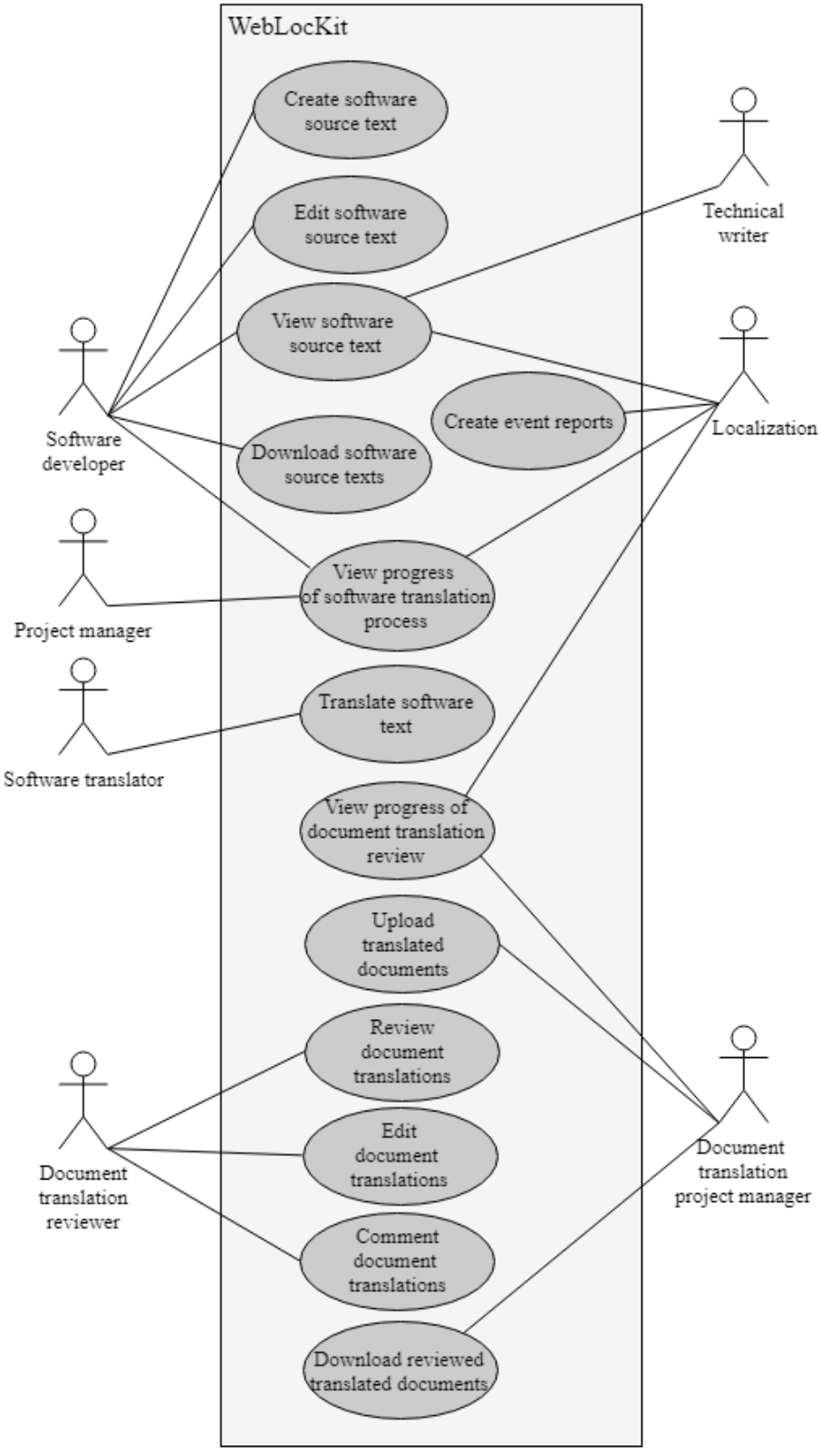

Figure 5. WebLocKit use cases.

As described by Tolvanen (2003), information that is stored in each text item includes:

- Contents, original English software string collected from the user interface.
- Enumeration, a unique identifier.
- Description, description of the text item.
- Comments, information of the text item properties.
- Path, instructions on how to find the text in the original UI.
- Text group, category of the text item.
- Width of contents, maximum space that the contents can occupy in the original UI.
- Number of rows, maximum number of rows that the contents can occupy in the original UI.
- Font, font name of the contents in the original UI.
- Font size, size of the font in the original UI.
- Unit, unit of width of contents and font-size values.

Some user groups will only need to view the text items but not modify or create them. An example of such group is the technical writers, who require read-only access. A reduced version of the text input interface will be provided to users who need read-only access to reduce the number of unwanted changes.

Software translation tool is used to translate texts item strings that software developers previously entered into the localization system. Translation memory and a termbase are planned to be integrated into the translation environment as they've been demonstrated to speed up of translation work as well as to increase the translation consistency (Olohan, 2011). The translation interface will provide useful warnings to the translators, for example if the translation contains unnecessary empty spaces or if numerals in source text and in translation do not match. Text item-specific comments will allow the translators and localization coordinators to communicate using the interface rather than having to rely entirely on email communication.

Previously all translation work was done using offline tools, which resulted in the software translators having to wait until all English texts have been prepared until they could begin translating (Tolvanen, 2003). Decision to implement software translation in a web environment was made because according to Tolvanen (2003), software text inputting

suffered from similar problems before it was fixed with the previous version of the web localization tools. The translation tools will allow the translators to translate in parallel with localization coordinators who prepare individual texts, since in the online environment the translators will be directly editing the data in the localization database. This will also allow the localization management to view the progress of translation projects in real-time.

Document translation review tool is used to evaluate the quality of translations of documents such as user manuals. Reviewers will be able to approve and modify translations as well as comment to individual translations. Like in the software translation tool, warnings and tools to speed up the work are provided. From the point of view of the localization team, document translation and reviewing will no longer be a black box, but instead the localization team will have access to up to date information about the progress of the review process.

Finally, **Document management** is a tool to be used by document translation project managers to upload translated documents and to download reviewed documents. **Download** and **Event report creation** tools exist in the original WebLocKit. In this thesis the interfaces of these two tools will be improved while their functionality will remain unchanged.

4.2.2 Interface design

Requirements document, recommendation from literature and end user feedback were used to justify decisions regarding the UI. For each of the three major tools of the web environment an Interaction Flow Modeling Language (IFML) (Brambilla & Fraternali, 2014) diagram was drawn. As an example, the IFML diagram for Text input tool is shown in Figure 6 portraying the impact of control tools on the TextInputNavigator component that renders a table of text items. Various filters and selection tools are located on top of the interface. As text items are organized into product lines and related product versions, hereby referred to as launch groups, corresponding selectors exist to control which text items should be fetched from the localization database. From the ColumnSelector dropdown menu users can change visibility and order of columns in the text item table. There is also a search bar that can be used to filter text items by a text string. Using the FilterSelector component users

can choose a filter that filters out all text items that are not included in the selected filter text file.

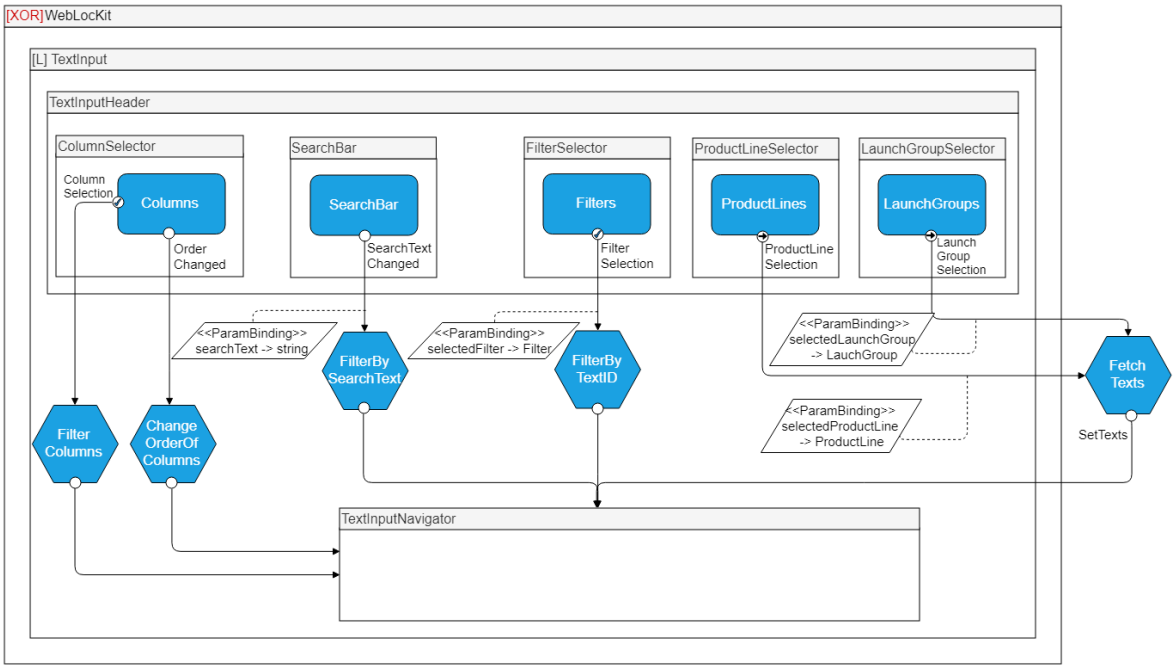

Figure 6. IFML diagram for Text input control tools.

The components on which the user can interact within the TextInputNavigator component are shown in **Figure 7**. As shown by the diagram, clicking on a navigator header will sort the text items based on the selected column. Selecting any of the text items opens a view in which the text items can be created, edited, viewed, or deleted. The rest of the IFML diagrams are similar to the Figure 6 and **Figure 7** and they can be viewed in Appendix II.

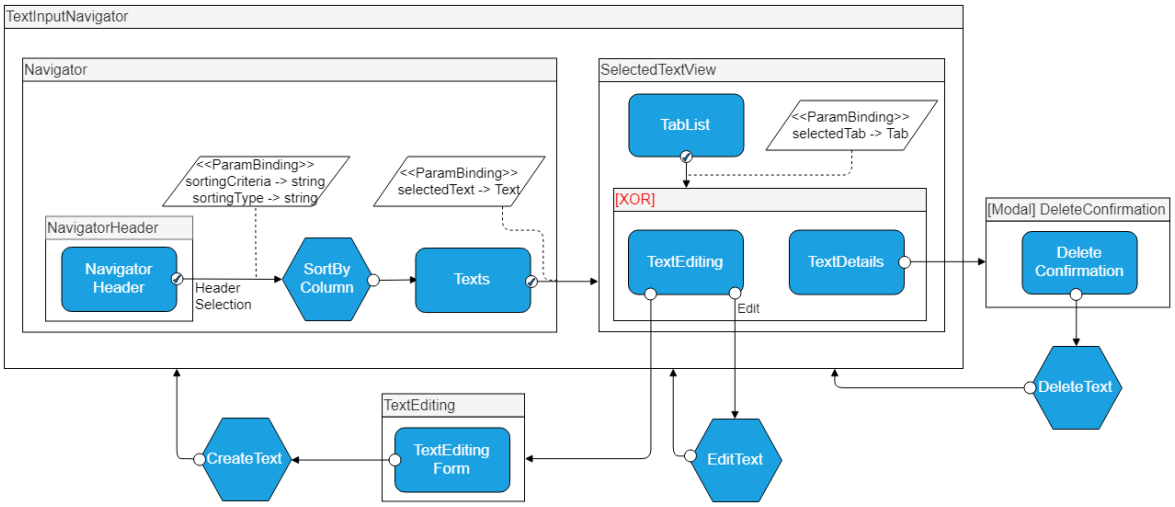

Figure 7. IFML diagram for text item table components.

Flat command structure was selected for the interface since it has been demonstrated to increase learnability and task completion efficiency (Cockburn et al., 2014; Scarr et al., 2012). Keyboard shortcuts will be provided to increase efficiency (Cockburn et al., 2014; Scarr et al., 2011) but using them will be optional as keyboard shortcuts can alienate infrequent users if they are the only method of interaction (Quesenbery, 2001). Tables will be used to display data of text items, and they will be implemented using Rutter's (2017) guidelines.

The interface will use methods to reduce user errors. The interface is designed to allow for undoing the most recent actions that alter data in the database, such as editing and removing text items. Additionally, if a user is trying to delete a text item, the user will have to confirm that they really do want to complete the deletion action, as shown in Figure 7. Error messages will contain human-readable information as suggested by Nielsen (2001) and they will be displayed every time the web environment fails to communicate with the backend.

The users of the web environment are going to be from a variety of countries and cultures. However, a decision was made not to localize the interface due to the user groups mostly consisting of software developers and translators who are expected to understand English. Additionally, Hofstede's cultural dimensions are not considered because the number of different cultures in the userbase is so large that designing a system that takes into account all the cultures would not be feasible. To make the interface more usable for international

users, culturally specific icons will be avoided. Users will be able to select a language specific collating sequence to use when sorting text items, as suggested by Nielsen (1990). Additionally, UTF-8 encoding will be enabled so that the interface will be able to handle all international character sets.

The different tools were designed to look similar to each other to maximize consistency and thus increase learnability (Quesenbery, 2001). This also helps with implementation and maintainability, as the same components can be used in several different interfaces. Figure 8 and Figure 9 demonstrate this similarity, as they show the interfaces of software source text input and software translation tools.

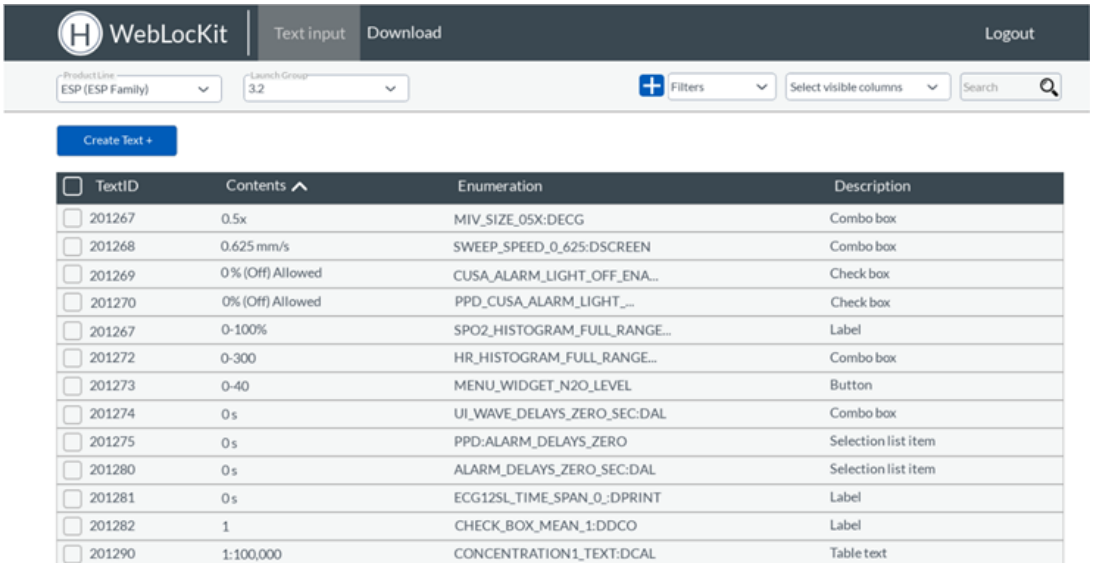

Figure 8. Mockup of software source text input interface

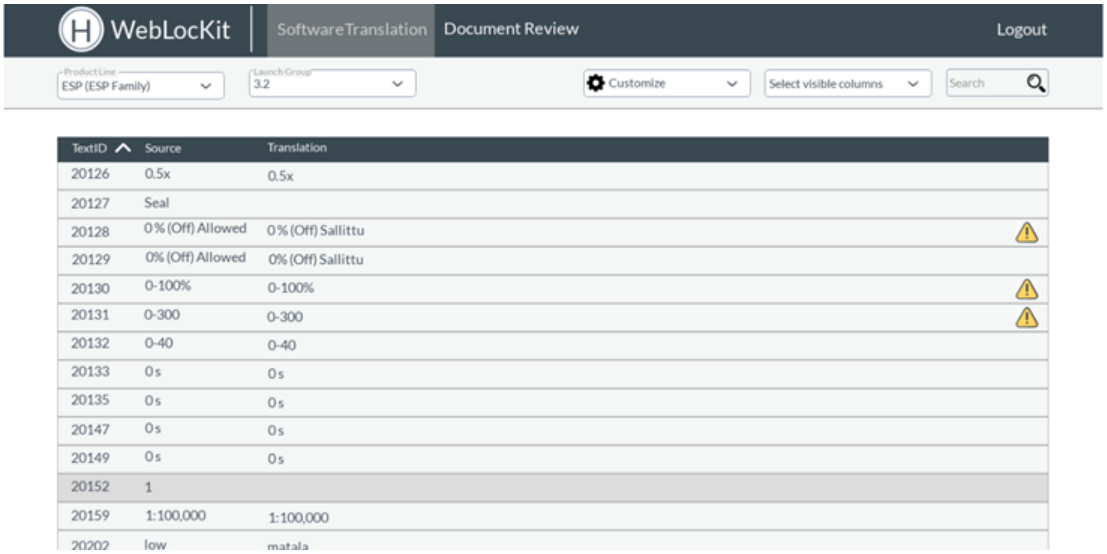

Figure 9. Mockup of software translation interface.

4.2.3 Technical details

To help developers with implementation the static structure of the design was captured in the style of class diagrams. The structural diagrams have been organized into three separate diagrams each of which describing the structure of either the text input, software translation or document translation review tools. As an example, Figure 10 shows the structure of the software translation tool. The structure is in the shape of the Document Object Model (DOM) tree of the website, as the structure of components in popular frontend frameworks such as React and Angular often follow the structure of its target DOM tree. The rest of the static structure diagrams are available in Appendix III.

To increase readability and maintainability of the source code, identifier names of the components, variables and methods were selected to be carefully considered abbreviations or full-word identifiers as was recommended by Lawrie's study (2007). Deissenboeck & Pizka's (2006) guidelines to select identifier names that are consistent, unique and specific enough was followed to avoid similar sounding identifier names that might confuse the maintenance programmers. IDDs were not implemented in the development environment, as they are not widely used in practice. Documentation will be done using inline documentation so that it will be consistent with other localization application conventions of the company.

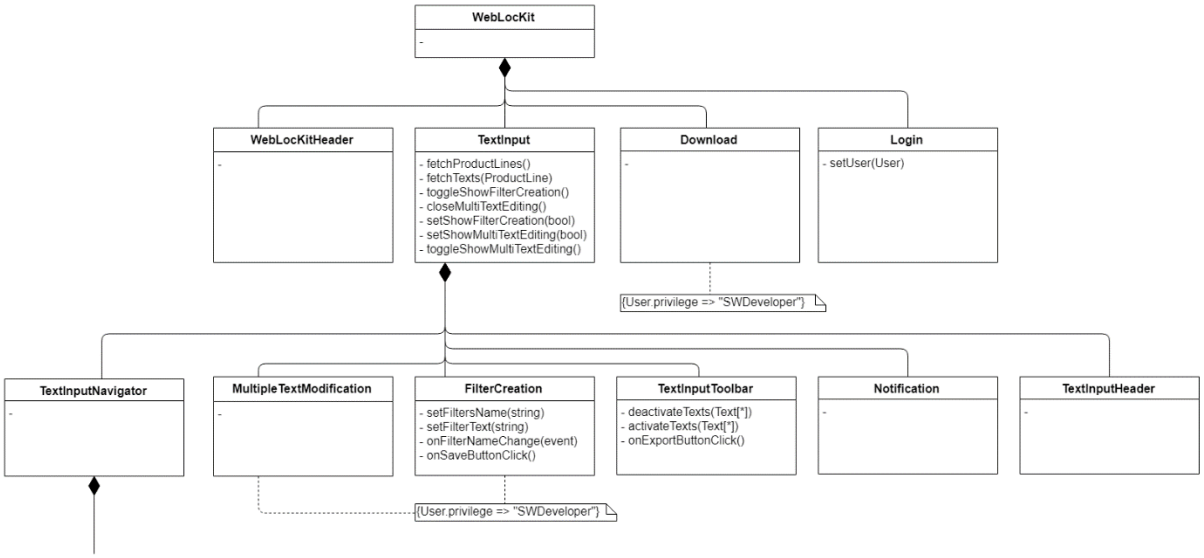

Figure 10. Static structure of the text input tool.

A development environment was implemented to enable implementation as soon as the design is finished. Coding standards were selected and embedded into the linting tools of the development environment. This way the programmers will get linting errors and warnings as soon as they write code that goes against the guidelines of the selected coding standards, minimizing the time required to manually search for these inconsistencies and thus improving maintainability of the code base. Additionally, a unit testing framework was installed to the development environment enabling automated tests to be used during refactoring and regression testing, as suggested by maintainability literature. Decisions was made to run unit tests every time a change in the source code is detected. An option to only run tests on changed files was enabled to minimize the test execution time.

To maximize responsiveness of the UI, communication with backend will be kept to as little as possible. This is achieved by fetching the most commonly used data at the beginning of the interaction. Subsequent communication with the backend is done only if the actions of the user create new data items that need to be stored in a database or if they make changes to existing data items. All communication with the backend will happen asynchronously not to block user interaction while waiting for data transferring to finish.

The communication between frontend and backend is visualized with sequence diagrams by providing examples from the point of view of the text input tool. For the purposes of

maintainability all asynchronous communication with the localization server is abstracted to a class or a higher order function, referred in the design diagrams to as LocalizationApi. As soon as the user has logged in to WebLocKit and opened the Text input tool, an asynchronous request is dispatched to fetch data of all available product lines and launch groups. The product line data must be queried from a database rather than stored it in a static file because it may change in future. Likewise, the product line data fetch is delayed until this point to avoid unauthorized users from gaining access to the localization data. The process of fetching product line data is illustrated in Figure 11.

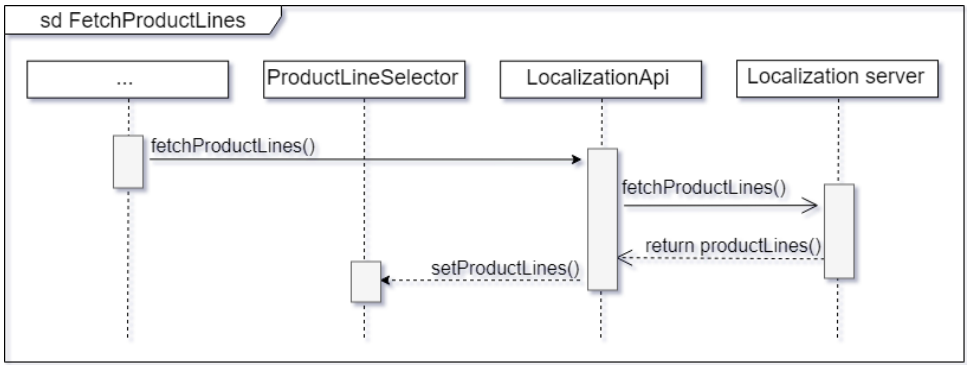

Figure 11. Sequence diagram for fetching product line data.

Once the product line data has been successfully collected from the database, the ProductLineSelector dropdown first introduced in Figure 6 is populated with the query results, after which the user can select a product line which they are interested in viewing or editing. Selecting a product line or a launch group will dispatch another asynchronous request to the server that queries the database for all text items related to the selected product line and launch group combination, as shown in Figure 12 and **Figure 13**. The fetched text items are then displayed in a table format. As typical users are expected to manage only one product line, the environment will remember what the user had selected previously and automatically complete the required steps to fetch text item data to reduce the number of manual steps required from the user.

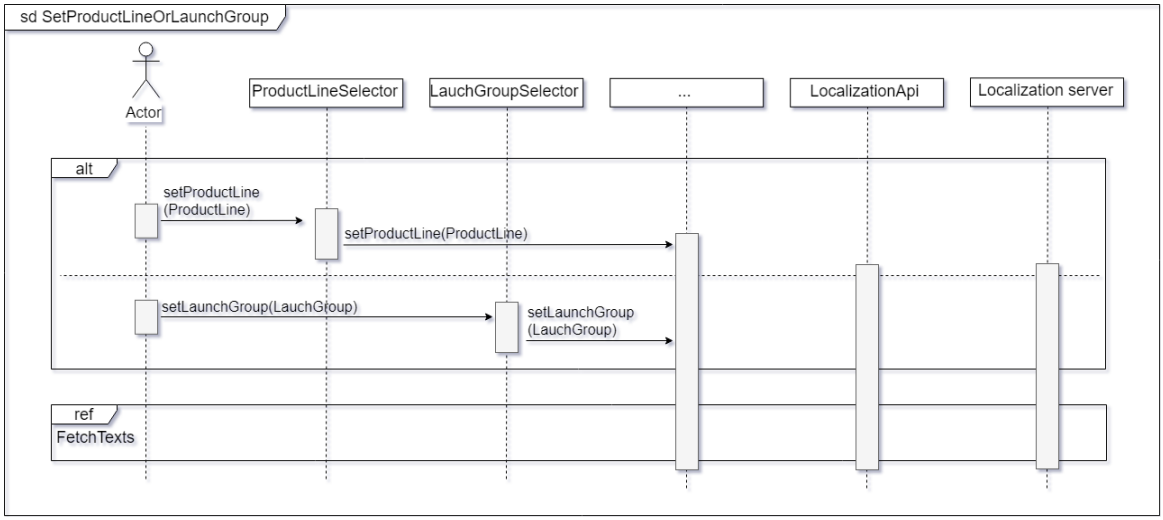

Figure 12. Sequence diagram for selecting product line or a launch group.

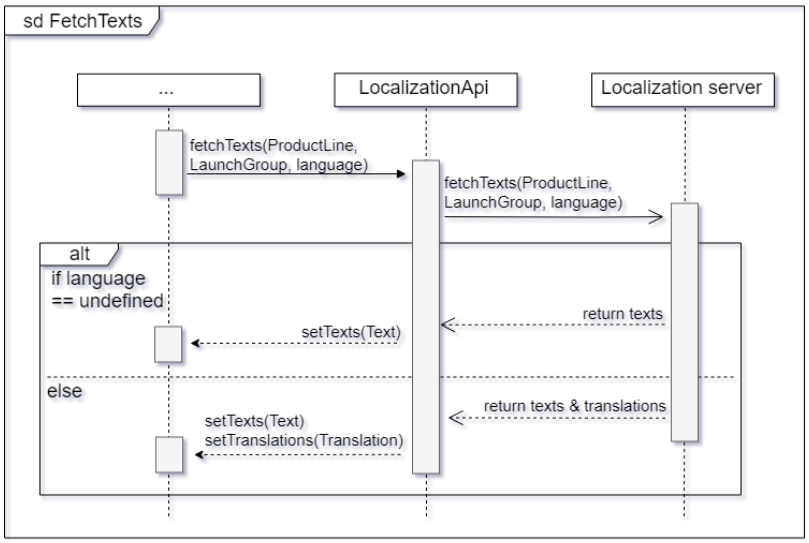

Figure 13. Sequence diagram for fetching text items.

When the table has been populated with text items, the user can create new text items or edit details of an existing text item. For convenience of software developers, a real time preview is provided that shows what the contents of the text item will look like in the target interface. The preview is created in an image format on the server, as demonstrated in Figure 14. Before the text item can be saved it is first validated on the server for security reasons. Text item validation and general flow of creating a text item are shown in Figure 15 and Figure 16 respectively.

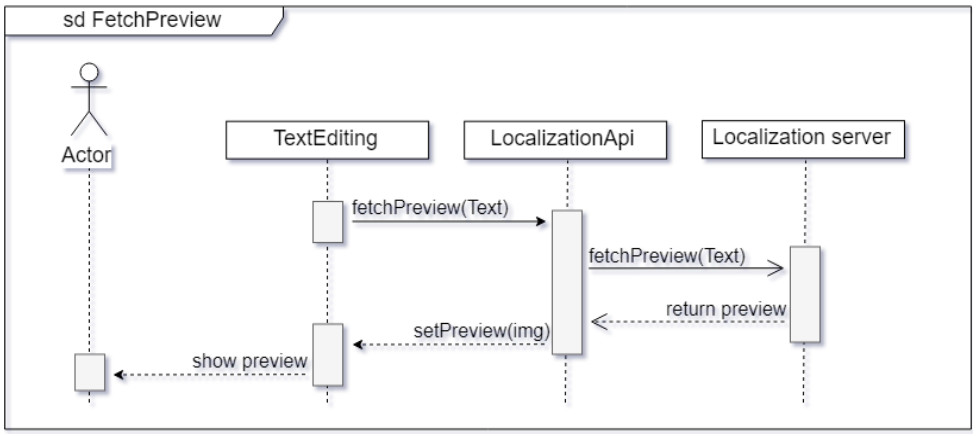

Figure 14. Sequence diagram for fetching previews.

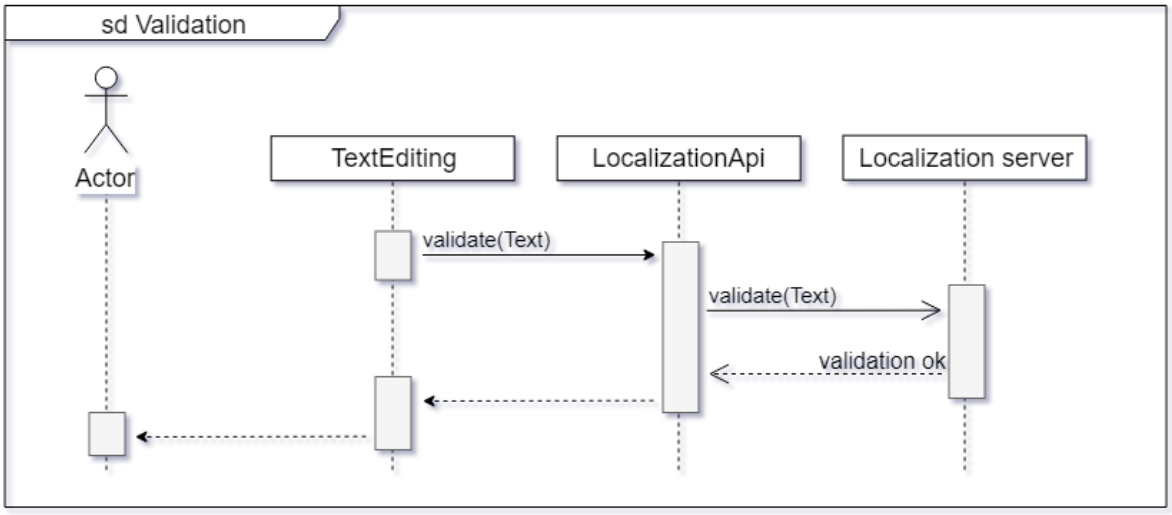

Figure 15. Sequence diagram for validating text items.

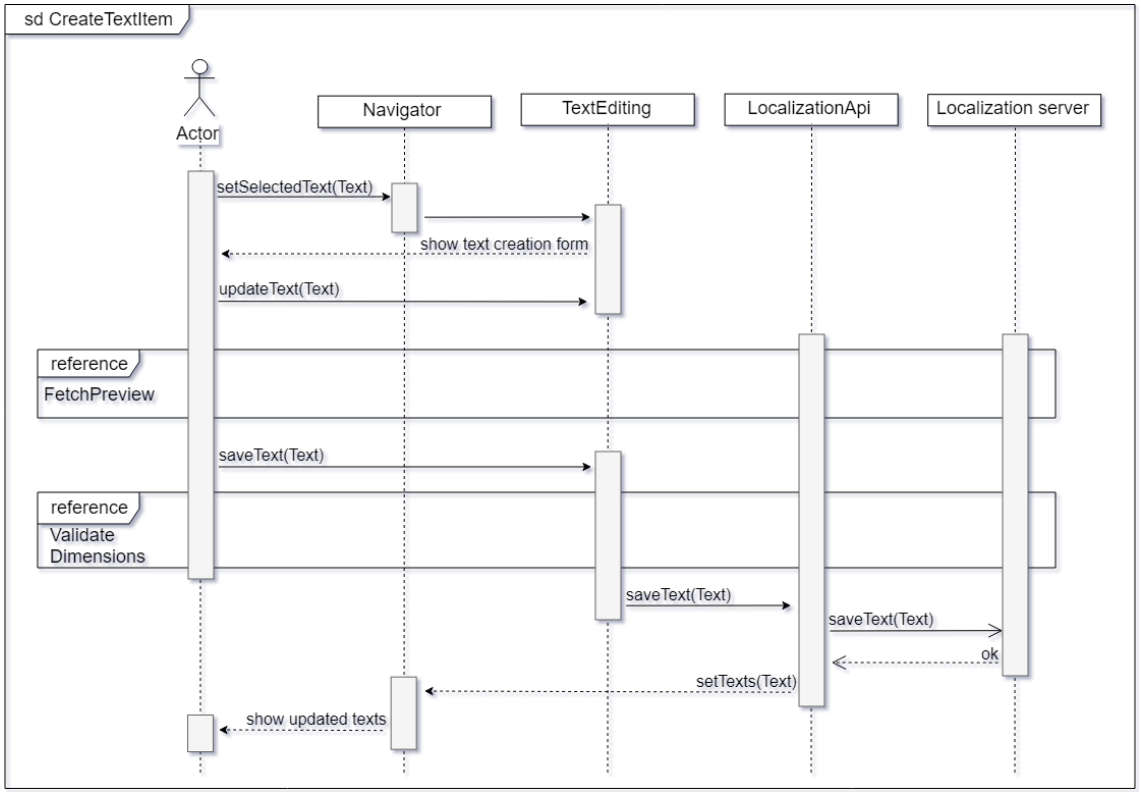

Figure 16. Sequence diagram for creating a new text item.

Finally, if any errors occur during any of the asynchronous requests from client to server, an appropriate error message will be displayed by the client-side Notification component. For the sake of saving space and reducing complexity of the sequence diagrams the error cases were left out. The error case is portrayed in Figure 17.

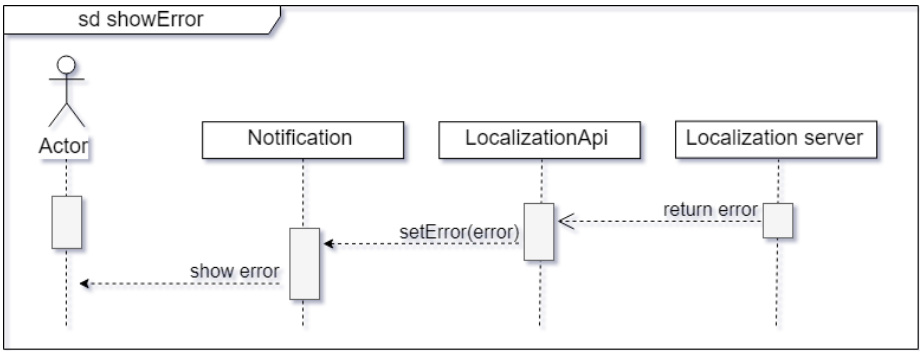

Figure 17. Sequence diagrams for showing an error.

5 EVALUATION OF THE DESIGN

This chapter describes how the design described in this study was evaluated using usability testing. Results of the usability test are also provided.

5.1 Usability test

Usability test was designed and run on real end users of the web environment. Usability tests were completed on a functional prototype that was implemented on top of the development environment. Usability testing was limited to the text input tool only, because users of software translation and document translation review tools reside in their home countries all over the world which is why organizing a usability test for these users would have to have happened in less than optimal online testing environments.

In the usability test users were given tasks that they had to complete using the prototype. Tasks used in the usability test were chosen based on what was believed to be the most commonly used functionalities in the final implementation of the text input tools. Tasks used in the usability test were categorized as follows:

Interface customization

1. Customize the order and visibility of table columns

Text item modification

2. Creating a new text item using an existing one as a template
3. Modifying a text item
4. Deactivating multiple text items
5. Performing action on all text items
6. Changing meta data of multiple text items at once

Finding information

7. Using the change log
8. Using filters
9. Sorting text items

The performance of the users was evaluated using pass/fail/passed with assistance criteria. Tasks that the testers couldn't pass without assistance were repeated to find out if tasks can be completed after learning. Discussion with the testers was held at the end of the testing sessions to collect open feedback, with focus being on the tasks with which the participant had most difficulties completing.

Number of test users was limited to five according to recommendations of Nielsen (2000; 2012), as five testers gives the best benefit-cost ratio. This is because five users per user group is enough to discover most usability issues. Adding more testers would yield less and less results as the new testers would mostly discover issues that have already been pointed out by previous testers. (Nielsen, 2000; Nielsen, 2012).

5.2 Results

Quantitative results of the usability study are portrayed in Table 1. As shown by the table a majority of the participants were able to complete most tasks with or without assistance. The only exception to this was task 1, which resulted in a failure by one participant due a fault in the prototype which prevented the tester to successfully change the order of columns.

Table 1. Results of the usability tests.

Test case	**Pass**	**Fail**	**Assisted**
1 Customize the order of table columns	3/5	1/5	1/5
2 Sorting text items	5/5		
3 Creating a new text item using an existing one as a template	5/5		
4 Modifying a text item	3/5		2/5
5 Deactivating multiple text items	4/5		1/5
6 Performing action on all text items	5/5		
7 Changing meta data of multiple text items	3/5		2/5
8 Using the change log	5/5		
9 Using filters	5/5		

The most challenging tasks seemed to be the ones where the actions of the user resulted in changes in parts of the interface on which the user was not focused. When editing details of an English text item in task 4, some participants didn't notice that the contents input fields at the top of the editing view provided information in real time about whether the contents of the text item would fit in the original interface. The visuals of the contents field were automatically updated any time the user changed the font type, font size or maximum allowed width of rows of the contents field, as demonstrated by Figure 18. The borders of the contents field visualize the maximum allowed width of contents and if this limit it exceeded the borders will turn red. In post-testing interview these participants who had missed the changes of the contents field said that at the time they were too focused on modifying other fields to notice that these changes also influenced the contents input field. It was suggested that the contents field should look more distinct from other input fields to help users notice it easier.

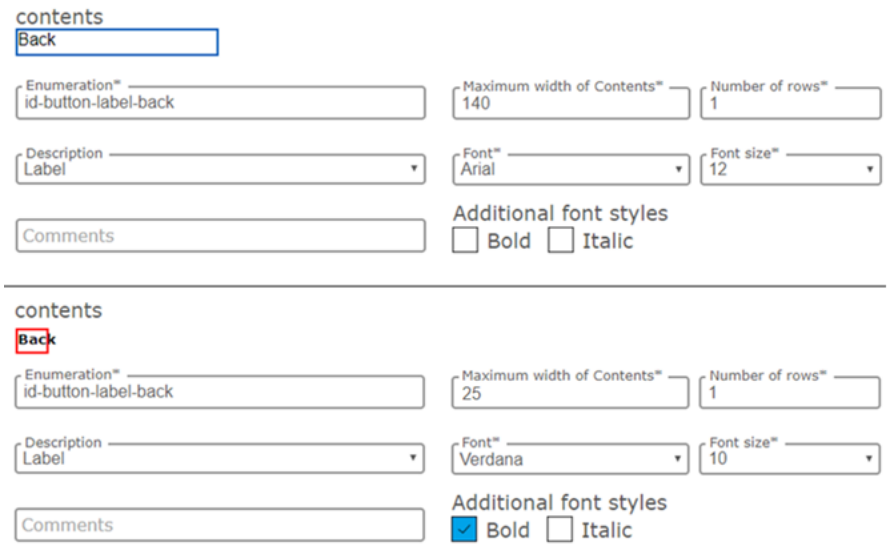

Figure 18. How changing the fields on the right affects the visuals of the contents field.

Task 1 was at first challenging to most participants. The order and visibility of table columns can be changed from a dropdown menu located top of the page as visible in Figure 19. However, the columns dropdown menu was not easily discovered by some participants. When trying to change the order of the columns the first thing that all participants tried was dragging and dropping the columns, which suggests that this action would be the most intuitive one. Like the editing view issue, changing the order of columns was considered

difficult because the actions of the user (clicking on arrow buttons in the columns dropdown menu) caused a change in other parts of the interface (the table behind the dropdown menu).

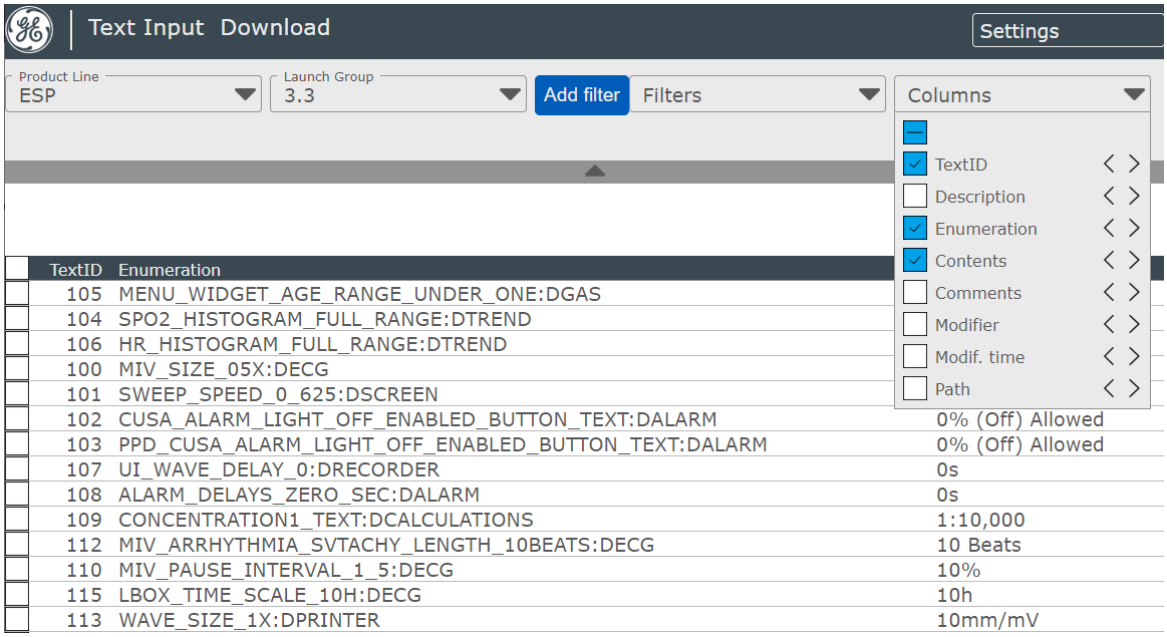

Figure 19. Columns dropdown menu expanded.

Task 7 revealed another opportunity for improvement. By default, the search bar on the top right corner of the interface searches from all properties of the text items. In task 7 the users had to change this setting to only search from a particular field. However, only one of the participants found the dropdown where the search options can be changed. To save space in the interface he dropdown menu becomes visible only after the user has clicked on the search bar, but the results of the usability test indicate that the menu will go unnoticed by most users.

Open feedback and additional ideas for improving the interface were collected at the end of the tests. Overall feedback was mostly positive. The new interface was preferred over the current one and participants that had had difficulties with some tasks during the usability test believed that they would be able to complete them after some practice. Some participants wished for shortcuts, such as closing a modal window with ESC key or double clicking on table items to open the editing interface. Keyboard shortcuts were not implemented in the prototype, but they designed to be usable in the final implementation, as mentioned in the

design chapter of this study. One of the participants suggested that filtering by typing in text in the search bar could be improved by highlighting the search text within the table. When creating or editing text items, it was suggested that inputting path values would greatly benefit from suggestions based on previously used path values, as these values are often reused for multiple text items. Finally, the text file-based filtering could be expanded to allow users to select different properties other than only id numbers of the text items.

6 DISCUSSION

One of the goals of this study was to improve the interface of the existing localization web interface and to add tools that speed up the localization process. According to the interview results of the usability study, the design described in this study fulfills the interface improvement goals. The new interface was preferred over the existing one and many of the new features such as editing multiple texts were considered as useful additions to the existing interface. However, this study cannot answer the question of if these features truly reduce cost or time required to complete localization projects. Answering this question would require data about the current time required to complete localization projects and compare that to projects that were completed with the help of the new localization web environment, but this data isn't easily available. As the web environment is currently at a prototype phase and no final implementation exists, answering this question will be left for future research, along with the implementation of the software.

The usability testing conducted in this study revealed several improvement opportunities regarding the UI. Fortunately, updating the design according to the results of the usability study will not require large scale re-design and many of them are quite trivial to implement in practice. In the final implementation controlling parts of the interface could be made more intuitive, such as selecting the order of table columns by dragging and dropping rather than selecting the order from a dropdown menu. Additionally, parts of the interface could use highlighting to grab the attention of the user in situations that were confusing to the participants, for example during filtering text items using the search bar or when the contents of the input field changes.

The suggestion to expand the text file filtering tool for other purposes needs to be carefully considered. Currently the filtering tool is primarily intended to be used by selecting some number of text items after which the filtering text file is generated automatically. Using the tool in a way that the test participant suggested would require users to manually input data into the filters, which would make it more difficult and burdensome to use. This study cannot answer whether this feature would be beneficial or not, as it would require data from experienced users. In addition, these changes would also increase the complexity of the

filtering tool both in terms of usability and maintainability, which is why the suggested changes might not be worth the design and development effort.

A concern for validity is that all participants of the usability test had been participating in previous localization projects or at least attended a training that introduced them to the localization process and tools used during it. To get better results it would have been beneficial to include at least one user with no previous experience in localization projects of GE Healthcare. Moreover, although the cultural background of the participants varied, all of them have been working in Finland for years and thus they can be assumed to have accustomed to the locale. Therefore, the results of this study cannot claim for sure whether the interface will be tolerated in all locales or cultures in other parts of the world.

Another concern is that the usability of software translation and document reviewing tools were not tested because the end users of these tools work from their home countries all over the world. However, it could be argued that due to similarity of the three interfaces designed in this study, usability testing all of them would have yielded redundant results as same issues would have been discovered by the different user groups. It is also possible that the interface model is only suitable for text inputting and users of other tools would find the design inconvenient. Additional usability testing on this topic would help alleviate these concerns.

In this study maintainability of the web environment was maximized by following the guidelines and suggestions provided by the literature study in chapter 2. Actions taken to improve maintainability include selecting and adhering to a set of coding standards, selecting meaningful and unique variable names and implementing a unit testing environment for refactoring and regression testing purposes. However, the effect of these actions on maintainability was not measured in this study, as it would require a full or at least partial implementation of the system. That is why estimating maintainability of the localization environment was left out of this study and the evaluation of the design utilizes only the results of the usability study.

Future research possibilities in the topic of localization would be to find out how localization tools impact productivity or to what extend do they reduce translation errors in a localized software product. Furthermore, the problem of how cultural dimensions (Hofstede & Hofstede, 2004; Hofstede et al., 2010) could most effectively be used in designing user interfaces remains unsolved. The concept of culturally adaptive interfaces could be explored further, as currently there is a lack of evidence regarding their scalability to accommodate a large quantity of different target cultures.

7 SUMMARY

The objective of this study was to design a web-based localization platform where English software texts can be inputted and translated, and document translations can be reviewed. The goal was to decrease costs and time related to localization by allowing different users to work on their tasks simultaneously through a web interface without having to install additional software. Moreover, the interface of the old environment was improved, and additional features were added to speed up parts of the localization process, such as editing multiple items at once, real time feedback on text item editing and .txt file-based filters.

Topics of usability and maintainability were explored in a literature study to find out how to improve usability of an UI (RQ1) and maintainability of software systems (RQ2). The changes that the existing web environment required were discovered by interviewing and surveying end users and by analyzing existing software. The identified changes and user needs were documented into a requirements document (RQ3). Based on requirements gathering and lessons learnt from the literature studies a design for the new web environment was created and explained (RQ4).

The design was usability tested on five end users utilizing a functional prototype. Almost all tasks were proven completable by first time users with some assistance. Several opportunities for improvement were identified. These include input field suggestions for fields that are often filled with similar values, search text highlighting and redesigning parts of the software so that the changes in the interface resulted from user actions are easier to notice.

REFERENCES

Abbes, M., Khomh, F., Guéhéneuc, Y-G., Antoniol, G. 2011. *An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension*. In 15th European Conference on Software Maintenance and Reengineering, CSMR 2011. IEEE Computer Society, pp. 181–190.

Abebe, S. L., Haiduc, S., Tonella, P., Marcus, A, 2009. *Lexicon Bad Smells in Software*. In 2009 16th Working Conference on Reverse Engineering, Lille, pp. 95–99.

Adar, E., Tan, D. S., Teevan, J. 2013. *Benevolent deception in human computer interaction*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '13, ACM.

Aggarwal, K. K, Singh, Y., Chhabra, J. K. 2002. *An integrated measure of software maintainability.* In Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318), Seattle, WA, USA, pp. 235–241.

Alexander, J. 2009. *Understanding and Improving Navigation within Electronic Documents.* Computer Science and Software Engineering. Christchurch, University of Canterbury. PhD thesis.

d'Astous P, Kruchten P, Robillard P, *Software engineering process with the UPEDU*, 2003

Bacchelli, A., Bird, C. 2013. *Expectations, outcomes, and challenges of modern code review.* In 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, pp. 712–721.

Bavota G., De Lucia, A., Di Penta M., Oliveto, R., Palomba, F. 2015. *An experimental investigation on the innate relationship between quality and refactoring*. In Journal of Systems and Software 107 (2015), pp. 1–14.

Berner, S., Weber, R., Keller, R. K. 2005. *Observations and lessons learned from automated testing*. In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 571–579.

Boogerd, C., Moonen, L. 2008. *Assessing the value of coding standards: An empirical study.* In IEEE International Conference on Software Maintenance, Beijing, pp. 277–286.

Boogerd, C., Moonen, L. 2009. *Evaluating the relation between coding standard violations and faults within and across software versions.* In 6th IEEE International Working Conference on Mining Software Repositories, Vancouver, BC, pp. 41–50.

Brambilla, M. Fraternali, P. 2014. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann.

Boehm, B., Basili, V. R. 2001. *Software Defect Reduction Top 10 List*. In IEEE Computer, 34(1), pp. 135–137.

Bourque, P., Fairley, R.E, eds. 2014. *Guide to the Software Engineering Body of Knowledge version 3.0*. Version 3.0, IEEE Computer Society. www.swebok.org.

Brown, W. H., Malveau, R. C., McCormick, H. W. S., Mowbray, T. J. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st John Wiley & Sons, Inc.

Buse, R. P. L., Zimmermann, T. 2012. *Information needs for software development analytics,* In 2012 34th International Conference on Software Engineering (ICSE), Zurich, pp. 987–996.

Byrne, J. 2009. *Localisation - When Language, Culture and Technology Join Forces*. In Language at Work - Bridging Theory and Practice. 3.

Carey, J. M. 1998. *Creating global software: A conspectus and review*. In Interacting with Computers, 9(4), pp. 449–465, ISSN 0953-5438.

Carroll, J., Rosson, M. 1987. *The paradox of the active user*. In J.M. Carroll (Ed.), Interfacing Thought: Cognitive Aspects of Human-Computer Interaction (Cambridge, MA: MIT Press).

Ceaparu, I., Lazar, J., Bessiere, K., Robinson, J., Shneiderman, B. 2004. *Determining causes and severity of end-user frustration.* In International Journal of Human-Computer Interaction.

Cedrim, D., Sousa, L., Garcia, A., Gheyi, R. 2016. *Does refactoring improve software structural quality? A longitudinal study of 25 projects*. In Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16), Eduardo Santanda de Almeida (Ed.). ACM, New York, NY, USA, pp. 73–82.

Ciancutti, J. 2010. *5 Lessons We've Learned Using AWS.* [online document]. [Accessed 13 March 2019]. Available at https://medium.com/netflix-techblog/5-lessons-weve-learned-using-aws-1f2a28588e4c.

Cockburn, A., Kristensson, P., Alexande,r J., Zhai, S. 2007. *Hard lessons: Effort-inducing interfaces benefit spatial learning*. In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'07). ACM, pp. 1571–1580.

Cockburn, A., Gutwin, C., Scarr, J., Malacria, S. 2014. *Supporting Novice to Expert Transitions in User Interfaces.* In November 2014 ACM Computing Surveys (CSUR): Volume 47 Issue 2.

Cockburn, A., Quinn, P. 2016. *When Bad Feels Good: Assistance Failures and Interface Preferences*. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems.

Curcio, K., Malucelli, A., Reinehr, S., Paludo, M.A. 2016. *An analysis of the factors determining software product quality: a comparative study*. In Computer Standards & Interfaces, 48, pp. 10–18.

Deissenboeck, F., Pizka, M. 2006. *Concise and consistent naming*. In Software Quality Journal, 14(3), pp. 261–282.

Elbaum, S., Malishevsky, A. G., and Gregg Rothermel. 2000. *Prioritizing test cases for regression testing*. In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00), Mary Jean Harold (Ed.). ACM, New York, NY, USA, pp. 102–112.

Esselink. B. 2000. *A Practical Guide to Localization*. Rev. ed. Amsterdam/ Philadelphia: John Benjamins Publishing Company.

European Telecommunications Standards Institute (ETSI). 2007. *Human Factors (HF); Multicultural and language aspects of multimedia communications*. ETSI EG 202 421 V1.1.1.

Fagan, M. E. 1976. *Design and Code Inspections to Reduce Errors in Program Development*. In IBM System Journal, 15(3), pp. 182–221.

Findlater, L., Moffatt, K., McGrenere, J., Dawson, J. 2009. *Ephemeral adaptation: the use of gradual onset to improve menu selection performance.* In April 2009 CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.

Fitzmaurice, G., Khan, A., Pieké, R., Buxton, W., Kurtenbach, G. 2003. *Tracking menus*. In ACM UIST, pp. 71–79.

Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M. 2016. *Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification*. In IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, pp. 609–613.

Ford, G., Gelderblom, H. 2003. *The Effects of Culture on Performance Achieved Through the Use of Human–Computer Interaction.* In Proceedings of the Annual Research

Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology, pp. 218–230.

Fowler, M., Beck, K., Brant J., Opdyke W., don Roberts. 1999. *Refactoring improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston.

Gajos, K. Z., Everitt, K., Tan, D. S., Czerwinski, M., Weld, D. S. 2008. *Predictability and accuracy in adaptive user interfaces*. In Proc. CHI'08, pp. 1271–1274.

Garg, D., Datta, A., French, T. 2012. *A Two-Level Prioritization Approach for Regression Testing of Web Applications*. In 19th Asia-Pacific Software Engineering Conference, Hong Kong, 2012, pp. 150–153.

Giannisakis, E., Bailly, G., Malacria, S., Chevalier, F. 2017. *IconHK: Using Toolbar button Icons to Communicate Keyboard Shortcuts*. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. CHI '17, ACM.

Gil, J. Y., Lalouche, G. 2016. *When do software complexity metrics mean nothing? –when examined out of context*. In Journal of Object Technology 15 (1), 2:1.

Gligoric, M., Eloussi, L., Marinov, D. 2015. *Practical regression test selection with dynamic file dependencies*. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015). ACM, New York, NY, USA, pp. 211–222.

Gligoric, M., Negara, S., Legunsen, O., Marinov, D. 2014. *An empirical evaluation and comparison of manual and automated test selection*. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, pp. 361–372.

González-Rubio, J., Casacuberta, F. 2017. *Cost-sensitive active learning for computer-assisted translation*. In Pattern Recognition Letters, 37, pp. 124–134.

Görg, C., Weißgerber, P. 2005. *Error detection by refactoring reconstruction*. In MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories, pp. 1–5, New York, NY, USA. ACM Press.

Grossman, T., Dragicevic, P., Balakrishnan, R. 2007. *Strategies for accelerating on-line learning of hotkeys*. In CHI ’07, ACM, pp. 1591–1600.

Gupta, P., Ivey, M., Penix, J. 2011. *Testing at the speed and scale of Google*. [online document]. [Accessed 29 March 2019]. Available at http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

Hallal, H. H., Alikacem, E., Tunney, W. P., Boroday, S., Petrenko, A. 2004. *Antipattern-Based Detection of Deficiencies in Java Multithreaded Software*. In QSIC ’04: Proceedings of the Quality Software, Fourth International Conference. IEEE Computer Society, Cent de Recherche Informatique de Montreal, pp. 258–267.

Harper, S., Michailidou, E., Stevens, R. 2009. *Toward a definition of visual complexity as an implicit measure of cognitive load*. In ACM Transactions on Applied Perception, 6, 2.

Harris, J. 2005. *Enter the Ribbon*. [online document]. [Accessed 3 May 2019]. Available at http://blogs.msdn.com/b/jensenh/archive/2005/09/14/467126.aspx.

Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., Gujarathi, A. 2001. *Regression test selection for Java software*. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01). ACM, New York, NY, USA, pp. 312–326.

Helmut. 2016. *The Difference between a Translation Memory and Term Base (and how to use them correctly).* [online document]. [Accessed May 16, 2019]. Available at https://lingohub.com/blog/2016/09/difference-translation-memory-term-base/.

Hofmeister, J. 2019. *Shorter identifier names take longer to comprehend*. In Empirical Software Engineering, 24(1), pp. 417–443.

Hofstede, G., Hofstede, G. J. 2004. *Cultures and Organizations: Software of the Mind: Intercultural Cooperation and Its Importance for Survival*. McGraw-Hill, New York.

Hofstede, G., Hofstede, G. J., Minkov, M. 2010. *Cultures and Organizations. Software of the Mind*. *Intercultural Cooperation and Its Importance for Survival*. McGraw-Hill.

Höök, K. 2000. *Steps to Take Before Intelligent User Interfaces Become Real.* In Interacting with Computers, 12(4), pp. 409–426.

Hwang, T. K. P., Yu, H. Y. 2011. *Accommodating Both Expert Users and Novice Users in One Interface by Utilizing Multi-layer Interface in Complex Function Products*. In: Rau P.L.P. (eds) Internationalization, Design and Global Development. IDGD 2011. Lecture Notes in Computer Science, vol 6775. Springer, Berlin, Heidelberg.

IEEE Standard Glossary of Software Engineering Terminology. 1983. In ANSI/ IEEE Standard 729-1983, pp. 1–40.

The Indian Express. 2016. *States with total and phase-wise prohibition of alcohol in India*. [online document]. [Accessed 2 May 2019]. Available at: https://indianexpress.com/article/india/india-news-india/bihar-liquor-ban-states-having-total-prohibition-gujarat-kerala/.

ISO 9241-11. 1998. *Ergonomic requirements for office work with visual display terminals*. Part 11: Guidance on usability.

ISO/IEC/IEEE 24765. 2010. Systems and Software Engineering—Vocabulary, ISO/ IEC/IEEE.

Jaakkola, H., Thalheim, B. 2014. *Multicultural adaptive systems*. In Frontiers in Artificial Intelligence and Applications 272, pp. 172–191.

Jbara, A., Matan, A., Feitelson, D. G. 2014. *High-MCC Functions in the Linux Kernel*. In Empirical Software Engineering 2014, 19 (5), pp. 1261–1298.

Karhu, K., Repo, T, Taipale, O., Smolander, K. 2009. *Empirical Observations on Software Testing Automation.* In *2009 International Conference on Software Testing Verification and Validation*, Denver, CO, pp. 201–209.

Kersten, G. E., Kersten, M. A., and Rakowski, W. M. 2002. *Software and Culture: Beyond the Internationalization of the Interface*. In Journal of Global Information Management 10(4), pp. 86–101.

Khan, Y. A., El-Attar, M. 2016. *Using model transformation to refactor use case models based on antipatterns*. In Information System Frontiers 18 (1), pp. 171–204.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H. 2011. *BDTEX: a GQM-based Bayesian approach for the detection of antipatterns*. In Journal of Systems and Software. Ecole Poly- technique de Montreal, Montreal, Canada pp. 559–572.

Kim, M., Cai, D., Kim, S. 2011. *An empirical investigation into the role of refactorings during software evolution*. In ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering.

Kim, S., Kim, D. 2016. *Automatic identifier inconsistency detection using code dictionary*. In Empirical Software Engineering, Volume 21, Issue 2, pp. 565–604.

Kim J.-M., Porter A., Rothermel, G. 2000. *An empirical study of regression test application frequency*. In Proceedings of the 22nd International Conference on Software Engineering, pp. 126–135.

Kim, M., Zimmermann, T., Nagappan, N. 2012. *A Field Study of Refactoring Challenges and Benefits*. In Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE ’12). ACM, New York, NY, USA, Article 50.

Kitchenham, B.A., Pfleeger, S.L. 2002. *Principles of survey research: part 3: constructing a survey instrument.* In ACM SIGSOFT Software Engineering Notes, 27(2), pp. 20–24.

Koester, H. H., Levine, S. P. 1994. *Modeling the Speed of Text Entry with a Word Prediction Interface*. In IEEE Transactions on Rehabilitation Engineering 2(3), pp. 177–187.

Koester, H. H., Levine, S. P. 1996. *Effect of a word prediction feature on user performance*. In Augmentative and Alternative Communication 12, 3 (1996), pp. 155–168.

Koskinen, J. 2015. *Software Maintenance Costs.* School of Computing, University of Eastern Finland

Kralisch, A., Berendt, B. 2004. *Cultural Determinants of Search Behaviour on Websites*. In Proceedings of the 6th International Workshop on Internationalization of Products and Systems, Vancouver, BC, Canada, July 8–10.

Kralisch, A., Eisend, M., Berendt, B. 2005. *The Impact of Culture on Website Navigation Behaviour*. In Proceeding of the 11th International Conference on Human–Computer Interaction, Las Vegas, NV, July 2005, pp. 22–27.

Kremenek, T., Ashcraft, K., Yang, J., Engler, D. R. 2012. *Correlation Exploitation in Error Ranking*. In Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (SIGSOFT '04/FSE-12). ACM, New York, NY, USA, pp 83–93.

Krisler, B., Alterman, R. 2008. *Training towards mastery: Overcoming the active user paradox*. In Proceedings of the 5th Nordic Conference on Human-computer Interaction: Building Bridges, NordiCHI ’08, ACM.

Kurtenbach, G. P. 1993. *The design and evaluation of marking menus*. University of Toronto, Ontario, Canada. PhD thesis.

Lagoudaki, E. 2006. *Translation memories survey 2006: Users' perceptions around TM use*. In Proceedings of the ASLIB International Conference Translating & the Computer, 28.

Lane, D., Napier, A., Peres, C., Sandor, A. *The Hidden Costs of Graphical User Interfaces: The Failure to Make the Transition from Menus and Icon Tool Bars to Keyboard Shortcuts*. In International Journal of Human-Computer Interaction 18 (2005), pp. 133–144.

Lavallée, M., Robillard, P. N. 2015. *Why good developers write bad code: an observational case study of the impacts of organizational factors on software quality*. In Proceedings - International Conference on Software Engineering. IEEE, Poly- technique Montréal, Montreal, Canada, pp. 677–687.

Lawrie, D. 2007. *Effective identifier names for comprehension and memory*. In Innovations in Systems and Software Engineering, 3(4), pp. 303–318.

Lazar J., Jones A., Hackley M., Shneiderman B. 2006. *Severity and impact of computer user frustration: A comparison of student and workplace users*. In Interacting with Computers 18(2), pp. 187– 207.

Leung, R., Findlater, L., McGrenere, J., Graf, P. Justine, Yang. 2010. *Multi-Layered Interfaces to Improve Older Adults' Initial Learnability of Mobile Applications.* In ACM Transactions on Accessible Computing (TACCESS), v.3 n.1, pp. 1–30.

Leung, R., Tang, C., Haddad, S., Mcgrenere, J., Graf, P., Ingriany, V. 2012. *How older adults learn to use mobile devices: Survey and field investigations*. In ACM Transactions on Accessible Computing 4, 3, pp. 1–33.

Lewis, R. D. 2011. *When Cultures Collide. Managing Successfully Across Cultures*. Nicholas Brealey, London, 3rd edition.

Li, X., Prasad, C. 2005. *Effectively teaching coding standards in programming*. In Proceedings of the 6th conference on Information technology education (SIGITE '05). ACM, New York, NY, USA, pp. 239–244.

Lionel, E. Deimel, Jr. 1985. *The uses of program reading*. In SIGCSE Bull. 17, 2 (June 1985), pp. 5–14.

The Localization Industry Standards Association (LISA). 2008. *Internationalization.* [online document]. [Accessed 2 May 2019]. Available at https://web.archive.org/web/20110102060632/http://www.lisa.org/Internationalization.58.0.html.

Lopez, A. 2008. *Statistical machine translation*. In ACM Computational Survey 40(3), 8:1–8:49.

Malacria, S., Bailly, G., Harrison, J., Cockburn, A., Gutwin, C. 2013. *Promoting hotkey use through rehearsal with ExposeHK*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI ’13, ACM.

Malinen, S., Nurkka, P. 2013. *The role of community in exercise: cross-cultural study of online exercise diary users*. In Proceedings of the 6th International Conference on Communities and Technologies (C&T '13). ACM, New York, NY, USA, pp. 55–63.

Mannan, U. A., Ahmed, I., Sarma, A. 2018. *Towards understanding code readability and its impact on design quality*. In Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering (NL4SE 2018). ACM, New York, NY, USA, pp. 18–21.

Marcus, A., Gould, E. 2000. *Crosscurrents: cultural dimensions and global Web user-interface design*. In Interactions 7(4), pp. 32–46.

Martin, J., McClure, C. 1983. *Software Maintenance: The Problem and Its Solutions*. Englewood Cliffs, NJ: Prentice-Hall.

Martini, A., Bosch, J., Chaudron, M., 2014. *Architecture technical debt: understanding causes and a qualitative model*. In 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 85–92.

Matejka, J., Li, W., Grossman, T., Fitzmaurice, G. 2009. *Community commands: Command recommendations for software applications*. In Proceedings of the ACM Symposium on User Interface Software and Technology. ACM, pp. 193–202.

McGraw, G. 2004. *Software security*. In IEEE Security Privacy, 2(2), pp. 80–83.

McLoone, H., Hinckley, K., Cutrell, E. 2003. *Bimanual interaction on the Microsoft Office Keyboard*. In INTERACT, pp. 49–56.

Miniukovich, A., De Angeli, A., Sulpizio, S., Venuti, P. 2017. *Design Guidelines for Web Readability*. In Proceedings of the 2017 Conference on Designing Interactive Systems. DIS '17.

Miniukovich, A., Sulpizio, S., De Angeli, A. 2018. *Visual complexity of graphical user interfaces*. In Proceedings of the 2018 International Conference on Advanced Visual Interfaces. AVI '18.

Mitzner, T. L., Fausset, C. B., Boron, J. B., Adams, A. E., Dijkstra, K., Lee, C., Rogers, W. A., Fisk, A. D. 2008. *Older adults' training preferences for learning to use technology*. In Proceedings of the Human Factors and Ergonomics Society Annual Meeting 52, 26, pp. 2047–2051.

Moha, N., Guéhéneuc, Y., Duchien, L., Meur, A.L. 2010. *DECOR: a method for the specification and detection of code and design smells*. In *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp 20–36.

Moha, N., Gueheneuc Y., Leduc, P. 2006. *Automatic Generation of Detection Algorithms for Design Defects*. In 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, 2006, pp. 297–300.

Moorkens, J. 2015. *Consistency in Translation Memory Corpora: A Mixed Methods Case Study*. In Journal of Mixed Methods Research, 9(1), pp. 31–50.

Murphy-Hill, E., Parnin, C., Black, A. P. 2009. *How We Refactor, and How We Know It*. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, pp. 287–297.

Nantel, J., Glaser, E. 2008. *The Impact of Language and Culture on Perceived Website Usability.* In Journal of Engineering and Technology Management, 25(1-2), pp. 112–122.

Nielsen J. 1990. *Designing User Interfaces for International Use*. Elsevier Science Publishers Ltd. Essex.

Nielsen, J. 1993. *Usability Engineering*. San Francisco (CA): Academic Press: Morgan Kaufmann.

Nielsen, J. 2000. Why you only need to test with 5 Users. [online document] [Accessed 5 May 2019]. Available at https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

Nielsen, J. 2001. *Error Message Guidelines*. [online document]. [Accessed 28 March 2019]. Available at https://www.nngroup.com/articles/error-message-guidelines/.

Nielsen, J. 2012. How Many Test Users in a Usability Study. [online document]. [Accessed 14 May 2019]. Available at https://www.nngroup.com/articles/how-many-test-users/.

Olbrich, S. M, Cruzes, D. S., Sjøberg, D. I. K. 2010. *Are all code smells harmful? a study of God Classes and Brain Classes in the evolution of three open source systems*. In Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 1–10.

Olohan, M. 2011. *Translators and translation technology: The dance of agency*. In Translation Studies 4, pp. 342–357.

Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A. D. 2014. *Do they really smell bad? a study on developers' perception of bad code smells*. In 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 101–110.

Pantiuchina, J., Bavota, G., Tufano, M., Poshyvanyk, D. 2018. *Towards just-in-time refactoring recommenders*. In Proceedings of the 26th Conference on Program Comprehension.

Peffers, K., Tuunanen, T., Rothenberger, M.A. & Chatterjee, S. 2007. *A Design Science Research Methodology for Information Systems Research*. In Journal of Management Information Systems, 24(3), p.45–77.

Price, R. J., Walton, R., Petersen, M. 2014. *Methodological journey: Lessons learned from a student-led intercultural pilot study*. In Rhetoric, Professional Communication, and Globalization, 5(1), pp. 90–107.

Pym, A., Perekrestenko, A., Starink, B. 2006. *Translation Technology and Its Teaching: (with Much Mention of Localization).* Intercultural Studies Group, Universitat Rovira i Virgili, pp 39.

Quesenbery, W. 2001. *What Does Usability Mean: Looking Beyond 'Ease of Use'*. In Proceedings of the 48th Annual Conference Society for Technical Communication.

Quinn, P., Zhai, S. 2016. *A Cost– Benefit Study of Text Entry Suggestion Interaction*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY.

Rafi, D. M., Moses, K. R. K, Petersen, K., Mäntylä, M. V. 2012. *Benefits and limitations of automated software testing: Systematic literature review and practitioner survey.* In 2012 7th International Workshop on Automation of Software Test (AST), Zurich, pp. 36-42.

Ratzinger, J., Sigmund, T., Gall, H. C. 2008. *On the relation of refactorings and software defect prediction*. In MSR ’08: Proceedings of the 2008 international working conference on Mining software repositories, pp. 35–38, New York, NY, USA. ACM.

Reinecke, K., Bernstein, A. 2007. *Culturally Adaptive Software: Moving Beyond Internationalization*. The 2nd UI-HCII'07, Springer, LNCS 4560, pp. 201–210.

Reinecke, K., Schenkel, S., Bernstein, A. 2010. *Modeling a User's Culture*. In The Handbook of Research in Culturally-Aware Information Technology: Perspectives and Models, IGI Global.

Rothermel, G. Harrold, M. J. 1996. *Analyzing regression test selection techniques*. In IEEE Transactions on Software Engineering, 22 (8), pp. 529–551.

Rothermel G., Harrold M. J., Ostrin J., Hong C. 1998. *An empirical study of the effects of minimization on the fault detection capabilities of test suites*. In Proceedings of the International Conference on Software Maintenance (ICSM 1998). IEEE Computer Press: Silver Spring, MD, pp. 34–43.

Rothermel, G., Harrold, M. J., von Ronne J., Hong, C. 2002. *Empirical studies of test-suite reduction*. Software Testing, Verification and Reliability, 12, 4 (2002), pp. 219–249.
Rothermel, G., Untch, R. J., Chu, C. 2001. *Prioritizing test cases for regression testing*. In IEEE Transactions on Software Engineering. 27, 10 (2001), pp. 929–948.

Runeson, P., Höst, M., 2009. *Guidelines for conducting and reporting case study research in software engineering*. In Empirical Software Engineering, 14(2), pp. 131–164.

Russell, G. W. 1991. *Experience with inspection in ultralarge-scale developments*. In IEEE Software, pp 25–31.

Russo, P., Boor, S. 1993. *How Fluent is Your Interface? Designing for International Users*. In Proceeding INTERCHI, ACM, pp. 342–347.

Rutter, R. 2017. *Web Typography: A handbook for designing beautiful and effective typography in responsive websites*. Ampersand Type; 1st edition.

Salgado, L., Pereira, R., Gasparini, I. 2014. *A survey of cultural aspects in Human Computer Interaction Research*. In Anais do WCIHC2014 - I Workshop sobre Questões Culturais em IHC, 8(3).

Schwartz, S. H. 1999. *A theory of cultural values and some implications for work*. Applied psychology, 48(1), pp. 23–47.

dos Santos, R. M, Gerosa, M. A. 2018. *Impacts of coding practices on readability.* In Proceedings of the 26th Conference on Program Comprehension (ICPC '18). ACM, New York, NY, USA, pp. 277–285.

Scarr, J., Cockburn, A., Gutwin, C., Quinn, P. 2011. *Dips and ceilings: understanding and supporting transitions to expertise in user interfaces*. In CHI' 11, ACM, pp. 2741–2750.

Scarr, J., Cockburn, A., Gutwin, C., Bunt, A. 2012. *Improving command selection with CommandMaps*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, pp. 257–266.

Shen, S., Woolley, M., Prior, S. 2006. *Towards culture-centred design*. In Interacting with Computers, 18(4), pp. 820–852, ISSN 0953-5438.

Shneiderman, B. 1987. Direct manipulation: *A step beyond programming languages (excerpt).* In Readings in Human-Computer Interaction: A Multidisciplinary Approach. R. Baecker and W. Buxton, eds. Morgan- Kauffman, pp. 461–467.

Shneiderman, B. 2003. *Promoting universal usability with multi-layer interface design*. In Proceedings of the 2003 conference on Universal usability.

Schneidewind, N. F. 1987. *The State of Software Maintenance*. In IEEE Transactions on Software Engineering, vol. SE-13, no. 3, pp. 303–310.

Sharma, T., Fragkoulis, M., Spinellis, D. 2016. *Does your configuration code smell?*. In Proceedings of the 13th International Workshop on Mining Software Repositories, pp. 189–200.

Sharma T., Spinellis D. 2018. *A survey on software smells*. In Journal of Systems and Software 138 (2018), pp. 158–173.

Shi, S., Gyori, A., Gligoricm M., Zaytsev, A., Marinov, D. 2014. *Balancing trade-offs in test-suite reduction*. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, pp. 246–256.

Shimagaki, J., Kamei, Y., McIntosh, S., Hassan, A. E., Ubayashi, N. 2016. *A study of the quality-impacting practices of modern code review at Sony mobile*. In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16). ACM, New York, NY, USA, pp. 212–221.

Shull, F., Basili V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M. 2002. *What We Have Learned About Fighting Defects*. In Proceedings of the 8th International Software Metrics Symposium (METRICS), pp. 249–258.

Silva, D., Tsantalis, N., Valente, M.T. 2016. *Why We Refactor? Confessions of GitHub Contributors*. In Proceedings of the 24th ACM SIGSOFT International Symposium on

Foundations of Software Engineering (FSE 2016). ACM, New York, NY, USA, pp. 858–870.

Sousa, L. d. S. 2016. *Spotting design problems with smell agglomerations*. In ICSE ’16: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, Pontifical Catholic University of Rio de Janeiro, pp. 863–866.

Stroggylos, K., Spinellis, D. 2007. *Refactoring - Does It Improve Software Quality?*. In Proceedings of the 5th International Workshop on Software Quality (WoSQ '07). IEEE Computer Society, Washington, DC, USA, 10-.

Tolvanen, J. 2003. *Kansainvälistämisen ja paikallistamisen menetelmät ohjelmistotuotannossa*. Master’s thesis. University of Helsinki.

Tom, E., Aurum, A., Vidgen, R. 2013. *An exploration of technical debt*. In Journal of Systems and Software, 86(6), pp. 1498–1516, ISSN 0164-1212.

Travis, D. n. d. *37 usability guidelines for help, feedback and error tolerance*. [online document]. [Accessed 28 March 2019]. Available at https://www.userfocus.co.uk/resources/helpchecklist.html.

Trompenaars, F., Hampden-Turner, C. 2011. *Riding the waves of culture: Understanding diversity in global business*. Nicholas Brealey Publishing.

Tuch, A. N., Bargas-Avila, J. A., Opwis, K., Wilhelm, F. H. 2009. *Visual complexity of websites: Effects on users’ experience, physiology, performance, and memory*. In International Journal of Human-Computer Studies, 67 (9), pp. 703–715, ISSN 1071-5819.

Umar, M., Khan, N. A. 2011. *Analyzing Non-Functional Requirements (NFRs) for software development*. 2011 IEEE 2nd International Conference on Software Engineering and Service Science, Beijing, pp. 675–678.

Vidal, E., Casacuberta, F., Rodriguez, L., Civera, J., Hinarejos, C. D. M. 2006. *Computer-assisted translation using speech recognition*. In IEEE Transactions on Audio, Speech, and Language Processing, 14(3), pp. 941-951.

Votta, L. G. Jr. 1993. *Does every inspection need a meeting?*. In Proc. of the ACM SIGSOFT Symposium on Foundations of Software Eng., pp. 107–114.

Wong, W. E., Horgan, J. R., London, S., Mathur, A. P. 1998. *Effect of test set minimization on fault detection effectiveness*. In Software Practice and Experience, 28(4), pp. 347–369.

Wong, W. E., Horgan, J. R., Mathur, A. P., Pasquini, A. 1999. *Test set size minimization and fault detection effectiveness: A case study in a space application*. In The Journal of Systems and Software, 48(2), pp. 79–89.

Yaaqoubi, J., Reinecke, K. 2018. *The Use and Usefulness of Cultural Dimensions in Product Development*. In Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18). ACM, New York, NY, USA, Paper CS18, 9 pages.

Yamashita, A. 2014. *Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data*. In Empirical Software Engineering, 19(4), pp. 1111–1143.

Yeo, A. 1996. *Cultural User Interfaces: A Silver Lining in Cultural Diversity.* In SIGCHI Bulletin 28(3), pp. 4–7.

Yoo, S. Harman, M. 2012. *Regression testing minimization, selection and prioritization: A survey*. In Software Testing, Verification and Reliability, 22(2), pp. 67–120.

APPENDIX I. Requirements related to the software text input tool

No	ID	Requirement	Subsystem	FURPS+
1	86	functionality to login without credentials	Login	F
2	92	users should be logged out after 3 hours of inactivity	Login	S
3	115	the system should use SSO login	Login	S / U
4	228	localization DB needs to store information of the user's first name and surname	Login	F / U
5	3	search bars should accept search modification characters such as wild card selection	Misc	U
6	5	should remember users' settings from previous sessions	Misc	U
7	97	the database update frequency should be limited to final changes, not on every key press	Misc	+
8	116	the UI of different tools should look similar to each other	Misc	U
9	153	keyboard shortcuts should be included	Misc	U
10	196	the user should be informed when the system is waiting for backend operations	Misc	U
243	260	product lines that a user is allowed to edit should be distinguishable from ones that cannot be edit by the user	Software text input	U
244	269	when flexible texts are deactivated OR deleted, all components of the flexible text should be deactivated or deleted	Software text input	U
245	272	functionality to edit details of multiple texts at once	Software text input	F
247	277	number of text items on the screen should be visible to the user	Software text input	U

APPENDIX II. IFML Diagrams

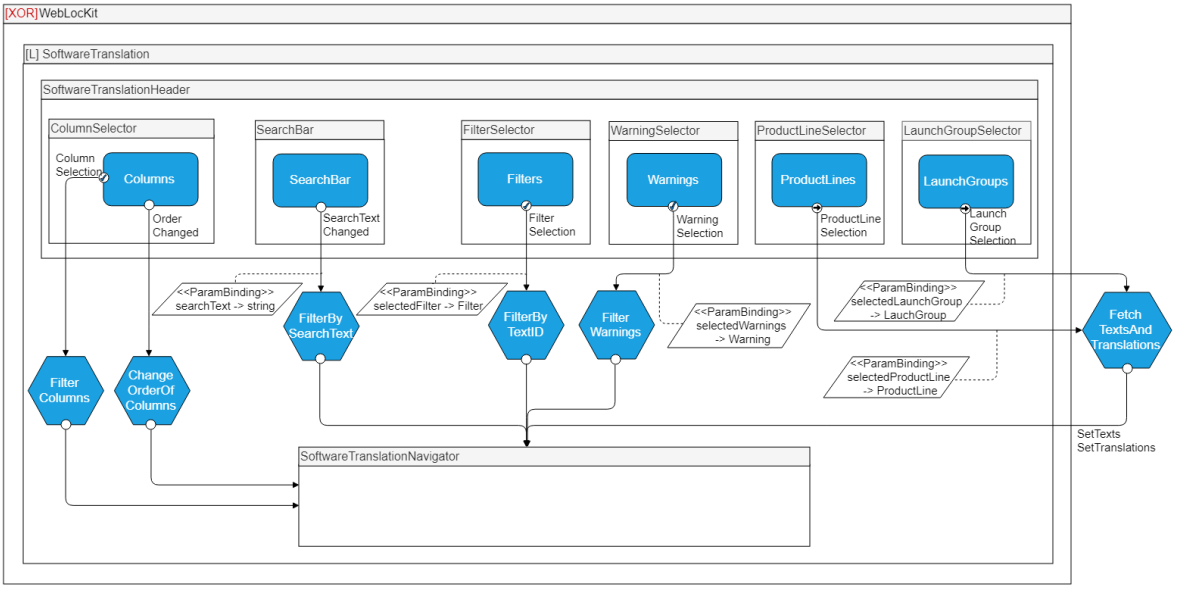

Figure A2.1. How selections in the software translation header affect the navigator table.

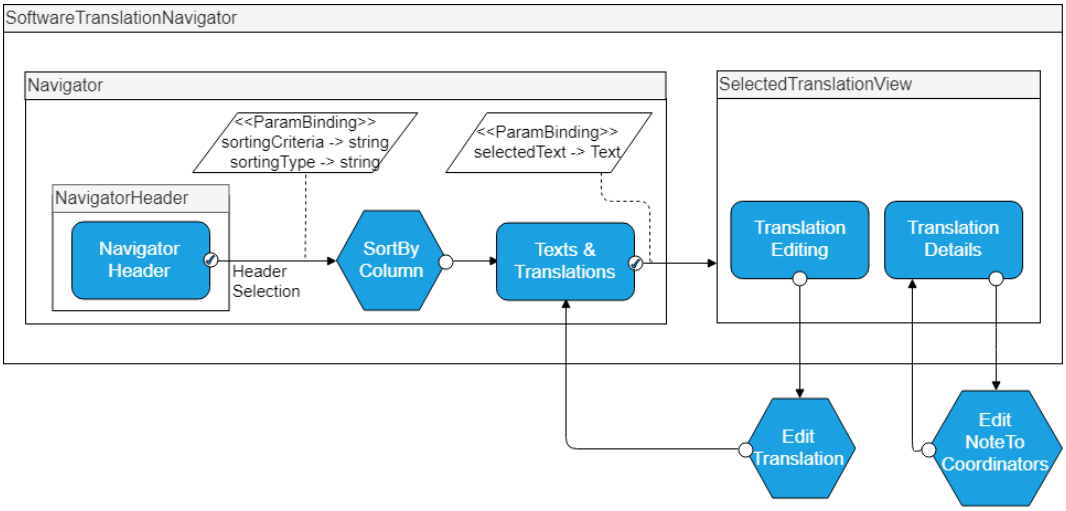

Figure A2.2. Internal view of the software translation navigator.

(Continues)

APPENDIX II. (Continues)

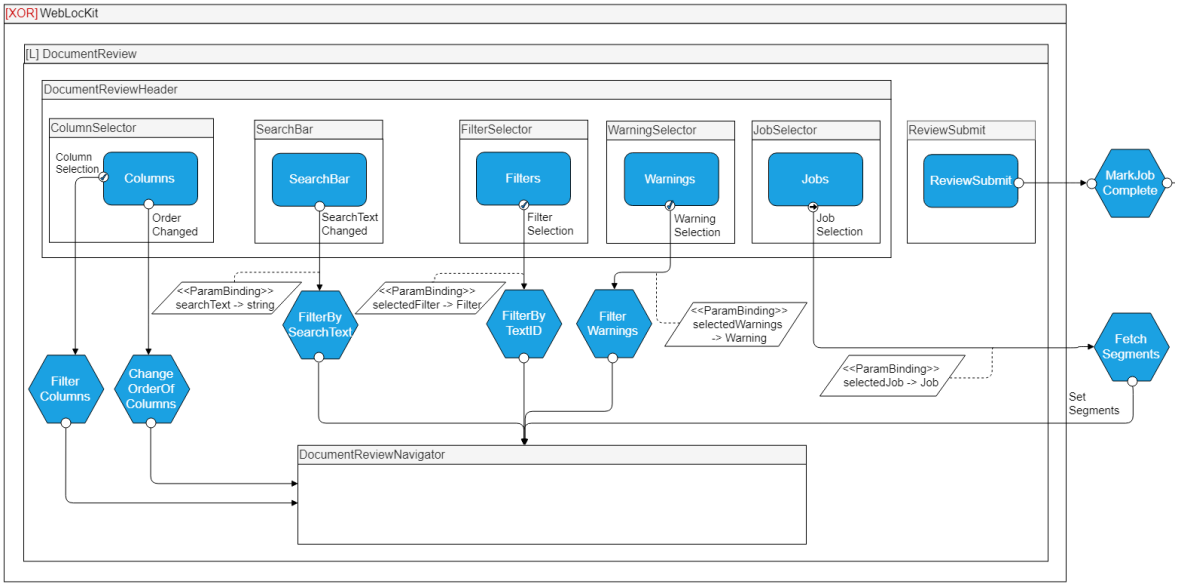

Figure A2.3. How selections in the document review header affect the navigator table.

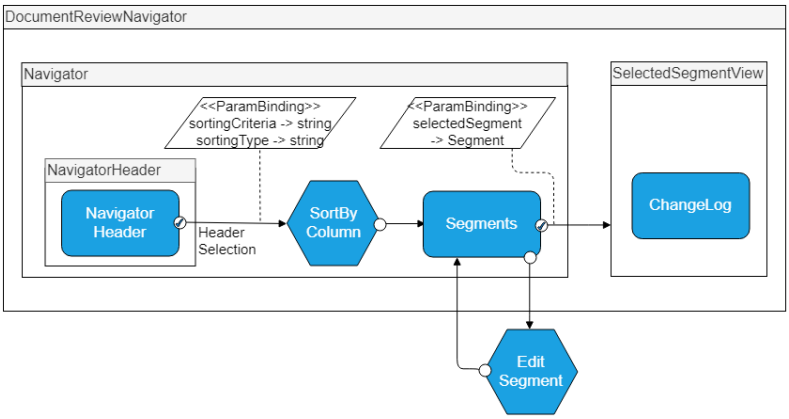

Figure A2.4. Internal view of the document review navigator.

APPENDIX III. Static structure of the platform

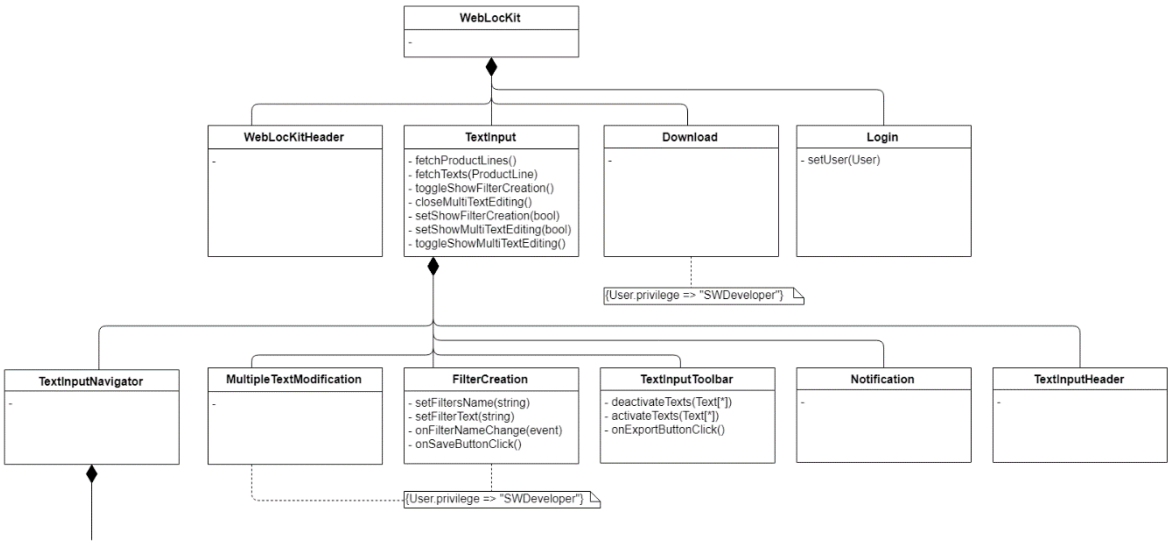

Figure A3.1. Structure from the point of view of the text input tool.

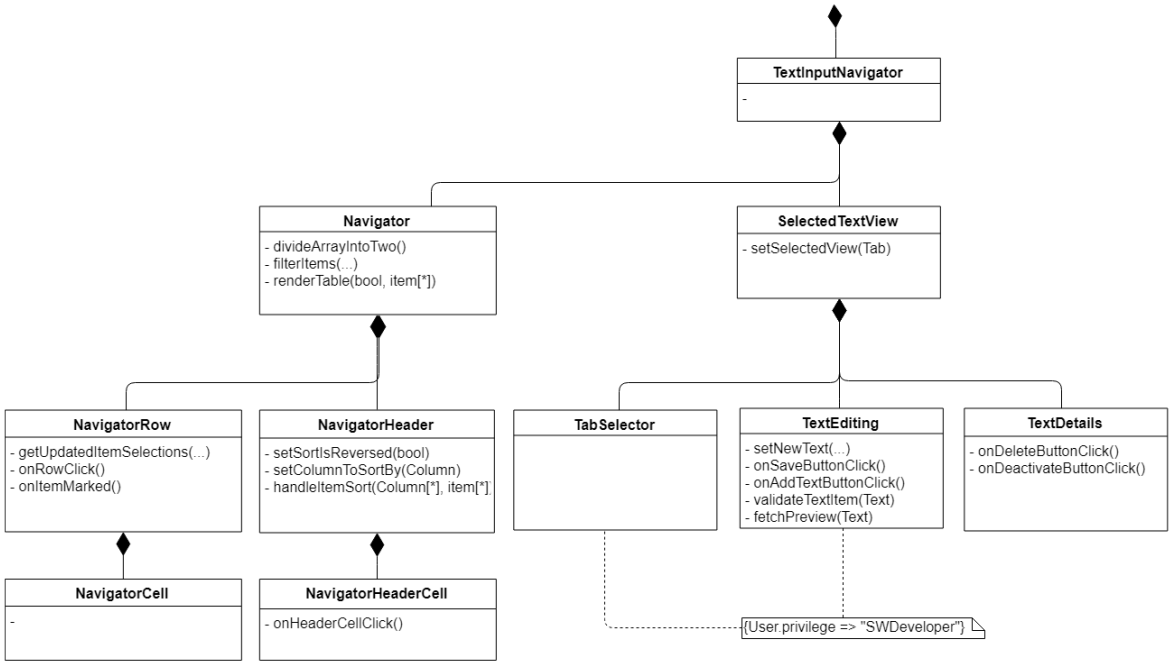

Figure A3.2. Structure of text input navigator.

(Continues)

APPENDIX III. (Continues)

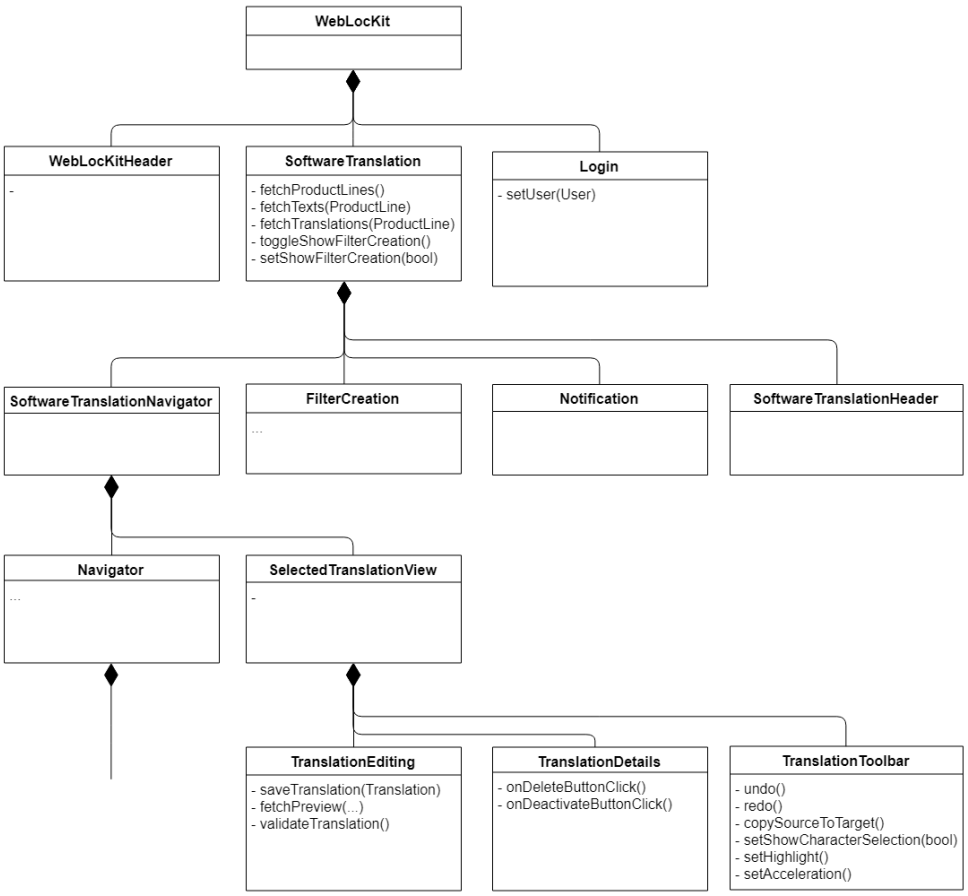

Figure A3.3. Structure from the point of view of the software translation tool.

(Continues)

APPENDIX III. (Continues)

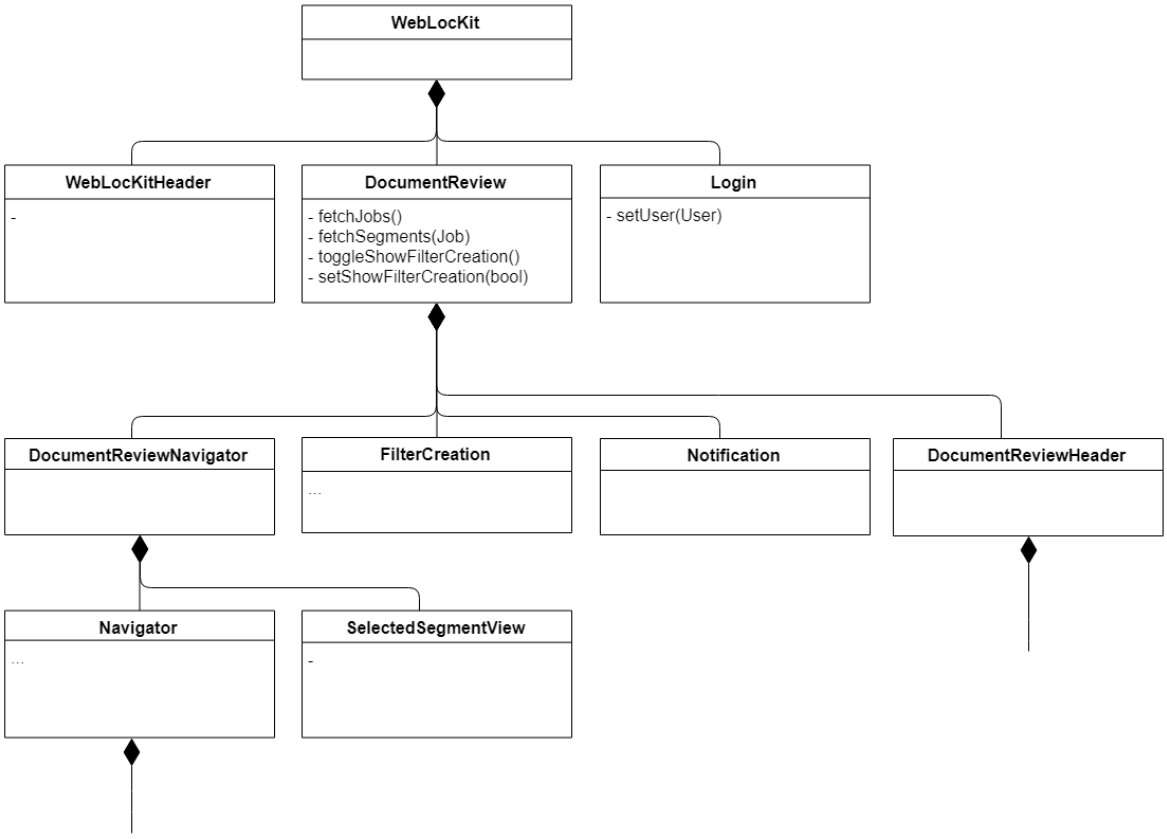

Figure A3.4. Structure from the point of view of the document review tool.