

ABSTRACT

Computers have fundamentally shaped the modern world. As applications make their way around the globe, providing software in a user's native language is an essential focus in software development. It serves a profound cultural purpose and acts as a critical accessibility measure. However, the global expansion of software presents massive, interconnected challenges for developers and linguists, proving that treating localization as a simple afterthought is a recipe for disaster. By examining historical precedents and modern development frameworks, this paper demonstrates that true localization requires developers to consider internationalization as a main focus.

INTRODUCTION

Computers have helped shape the modern world regarding the usability of its applications. An application, which works to serve a certain purpose, lives by its very name: it applies itself as a tool to solve certain problems. In the early days of computing, the problems these machines solved were rigidly mathematical and highly specialized. However, as technology has evolved, our reliance on the readability of the software has rapidly increased. Whether technical or measurable, applications are used to solve everyday tasks in an easier manner than any previous methods. Our end users in the modern world expect the software to be easily understood, and to serve a valid purpose. These end users will work with the application to accomplish a task, whether it may be a measurement tool, a form of digital entertainment, or by means of making a new application altogether. It's important for the end-users to understand what they are working with, or the application may appear to be yet another calculator in the eyes of the end user: To serve one purpose irrelevant to their cause.

In previous years, computers were used to serve this very purpose - solving mathematical equations. Some of the first computers took punch cards as input to provide a simple, yet relevant output for the calculations. Operating these large machines from the past required an in-depth understanding of machine code and physical hardware manipulation: a method that would seem impossible for general users in the modern world. These machines did not offer a typical user interface, but rather a simple dataset of inputs and outputs. Modern standards such as ASCII (American Standard Code for Information Interchange) would eventually be unveiled to allow the computer to appear to the general public as more than a simple calculation tool, but as a personal assistant, working to solve our issues in a more human-readable form. By establishing a universal 7-bit translation layer that standardized digital text, ASCII allowed machines from different manufacturers to finally communicate. For the first time, a sequence of binary code universally translated to a specific, readable letter of the alphabet.

However, this early standardization was inherently English-centric, lacking the capacity to represent European diacritics (For example, symbols for their currencies), let alone entirely different writing systems like Mandarin or Japanese. ASCII was limited to 128 characters, prioritizing the Latin alphabet, basic punctuation, and fundamental control codes (as 128 possible combinations fit within 7 bits). This revealed to the world that the current methods of converting binary machine code to characters was exclusively English-Centric. Despite these early limitations, these tools were used to create a much better understanding towards those who wished to use computers for non-mathematical purposes, such as writing and reading documents,

as well as playing computer-based games. While the computer may still use underlying mathematical properties to accomplish these tasks, the abstractions behind the methods they use work to ensure that the user believes the tool can work within their own understanding of thought (Read the computer output as text, and not machine code). Over time, as noted in the historical evolution of the industry outlined by Esselink, the process of adapting these abstractions shifted from highly centralized, in-house technical tasks to a broader global necessity (2003).

With this train of thought, users wish to understand their applications by sharing a commonality with the tools that they use. In other words, they wish to read the applications as if the application was either writing letters to the user, or speaking to them. Our devices should work together to solve our issues, and to troubleshoot our errors. This is how the user would understand the tools provided to them, just as any other machines in the past worked to do. However, these tools would eventually make their way around the world, serving their purpose within the context of their own cultures. As a result, applications would need to be made to cater towards those within different regions of the world, creating the need to cater towards those who speak many different human-spoken languages.

Research consistently shows that users strongly prefer and find it inherently easier to operate software in their native tongue (ACM, 2008). When forced to use software in a secondary language, users experience higher cognitive load, make more frequent errors, and generally report lower satisfaction with the product, especially when such a language is one they fail to understand. This transforms localization into a critical accessibility measure. As Keniston (1997) argued in early localization literature, providing software in native languages serves a profound cultural purpose: it helps public entities and marginalized groups preserve their linguistic heritage against the melting of their culture into an "American soup" of homogenized, English-default technology. When a culture cannot interact with the digital world using its own idioms, alphabets, and syntactical structures, it risks digital erasure. Therefore, robust software localization is not merely an act of translation; it is an act of cultural validation.

PROBLEM STATEMENT

The global expansion of software presents several critical, interconnected challenges for developers, linguists, and translation teams. In the early days of software distribution, localization was often treated as an afterthought—a simple matter of exporting text strings to a spreadsheet, hiring a translator, and hardcoding the translated text back into the application. However, as the scale of software projects grows, the traditional methods of static localization quickly break down, forcing the industry to reckon with three distinct problem areas: language selection, linguistic and cultural accuracy, and structural user interface design. Such design must provide to the user more care and flexibility compared to a simple translation tool.

DETERMINING THE SUPPORTED LANGUAGES

Every piece of software written will eventually have to answer the question of what languages it will be translated into. Not everybody speaks every language, but localising a software into another language is an expensive and time-consuming process. Organisations then must consider the potential users of their software, as well as balancing the costs and benefits for any language they would translate their software into. The question of which languages are

supported is of vital importance, though, as determining the languages the software can be read in, will end up dictating to some extent who can actually use the software.

COMPLICATIONS OF SOFTWARE TRANSLATION

The first rule of translation is that there is almost never a perfect translation. Languages vary greatly, from some having words that others don't, (Синий + Лазурный in Russian vs. Blue in English) to words that don't quite translate exactly translate properly, to languages using completely different sets of characters from each other. This alone would pose difficulties for anybody who wishes to automatically translate their software, as they would need to rely on context to guarantee an accurate translation, but intricacies in how native speakers of a language speak their own tongue further complicate things - phrases and idioms don't quite line up, (tell a Spanish speaker 'Ésta lloviendo gatos y perros' and they're going to look at you funny. Sure, the translation is accurate, but nobody actually says it like that.) words that vary via context, ('tú' vs 'usted' in Spanish for instance, both of them mean 'you', but they're used in different circumstances.) and regional dialects that sound nothing like their parent language. (Appalachian English, or Chilean Spanish, for instance.) All in all, this renders automatic translation challenging, as machine-translated text can often sound stilted or bulky, like the translation was an afterthought, which can cause users to feel somewhat disengaged from the software. But, on the other side of the coin, hiring a native speaker to manually translate your whole software is costly, time-consuming, and often not feasible on larger projects where there is a large quantity of material to be localised.

USER INTERFACE CONSIDERATIONS

Additionally, when localising software for other languages, one must consider the User Interface as well. For one, words in different languages aren't all the same size. A compact interface, then, might struggle to handle words in some languages (such as German or Finnish) where words, especially nouns, are built up in a compound manner, for example. Some languages are read right-to-left, so you've got to account for that in rendering. Oh, and that warning panel that comes up in red? Yeah, that'll have to be changed for the Chinese market, apparently red has different connotations there. Designing your user interface in such a manner that it can handle both the complexities of human language AND it can handle whatever cultural tendencies would affect design, (symbols that are offensive in some places and not others, colors just meaning different things, etc.) is a massive challenge, and one that heavily limits how accessible software is on the global market.

OBJECTIVES

The primary objectives of this paper are to identify best practices in software translation and to explore how to build and develop multi-lingual software that is accessible among its specific end users. By examining the linguistic limitations of direct translation, analyzing the cultural dimensions that dictate user comprehension, and outlining the necessary architectural UI considerations, this paper aims to demonstrate that true localization requires a large understanding of the relationship between flexible code architecture and deep cultural awareness. Through an analysis of historical precedents, contemporary linguistic challenges, and modern

full-stack development frameworks, this paper will provide a roadmap towards developing software that speaks to a wider range of users across the globe.

DISCUSSION OF PROBLEMS

As discussed above, there are a number of problems that face the aspiring localiser. The first issue one will encounter is the question of which languages to support, but translation itself can pose difficult, not to mention the need to build the User Interface in such a way that changing languages doesn't break everything. Here we go into greater depth on each of these issues.

DETERMINING THE SUPPORTED LANGUAGES

A huge deciding factor in language support for certain softwares involves the stakeholders of the application. Users would ideally like to be able to use the software in their native language, for both ease of use and comfort, and those within the organisation would like to minimise costs of translation. To satisfy these two contrasting goals, we must both consider what the most-used languages of our users are, and how much time, money, and effort it will take to localise our software for each language considered. There are, then, two main focuses when deciding who to translate for - economic considerations, and social ones - casting as wide a net as possible with as little material as possible.

Market forces often dictate that software is translated into the most economically viable languages (mostly due to where the product is likely to generate a good profit), but this approach can still leave out a significant portion of end-users. In the past, major software corporations have prioritized the "FIGS" languages (French, Italian, German, Spanish), alongside Japanese and Simplified Chinese. These regions represent a market where digital technology and subscription based services yield more of a profit, but digital accessibility is left out in the eyes of moneybags. For example, languages with millions of native speakers such as Swahili, Arabic, Bengali, or regional dialects of Hindi, are ignored by major tech corporations due to their immediate return on investment yielding less perceived benefit. Balancing cost-effective market reach with the contextual understanding of who actually needs the software remains a significant obstacle standing in our way. If developers only build tools for the most profitable linguistic demographics, they build digital barriers that prevent developing economies from fully participating in the modern tech ecosystem, or force people from those nations to learn a more 'valuable' language to be able to integrate into the tech industry. Additionally, this increased valuing of certain languages over others can lead to dangerous assumptions about the value of different cultures or traditions, which can perpetuate an exclusionary attitude both towards software development and existence in the global online ecosystem.

However, not all software exists to turn a profit. For some examples, open source software, software for use within an institution, or in the case of what even drove this exploration to begin with, a research aid intended for use in a foreign country. For these programs, the focus doesn't need to be on which languages will generate the most profit. Instead, they need to focus on covering as much of the likely user base as possible. In the case of, say, software used to gather data in Guatemala, one must consider both the language of those who live there, (Spanish) the language of the scientists using the data, (English) and the likely languages of tourists visiting the site. (English again, but also potentially French or German.) While these software

can frequently consider localising for languages one wouldn't see in for-profit software, they too often rely on volunteers for their translations, which can lead to fewer, and poorer-quality translations than otherwise.

COMPLICATIONS OF SOFTWARE TRANSLATION

How does one translate a software well? A major problem in modern development is the over-reliance on simple part-for-part translation, ignoring the complexities of human language. Novice developers often assume that language operates like a direct cipher, where one English word equates perfectly to one word in a foreign language. This is a fatally wrong assumption to make. As recent linguistic studies highlight, languages can almost never be perfectly directly translated between one another due to a lack of directly translatable words, different grammatical rules, and sometimes even completely different sentence structures. (HAL Thesis, 2023). For instance, English operates primarily on a Subject-Verb-Object (SVO) sentence structure. If a developer hardcodes an application to say **Welcome to [App Name], [User Name]!**, they are enforcing an English syntactical rule onto the code, ensuring the user understands the relevance of the greeting to that page. However, when this string is translated into Irish, which uses a Verb-Subject-Object (VSO) structure, or more commonly Japanese - a language that relies heavily on a Subject-Object-Verb (SOV) structure and utilizes distinct honorifics depending on the social status of the user (For example, older men may receive a different type of greeting compared to a younger woman) - that hardcoded English structure completely falls apart in the grammatical logic of the Japanese translation, resulting in a robotic and jarring user experience. As a result, translation calls for more care than just simple like-for-like mapping, requiring the group to have team members fluent in both languages (as modern technology such as machine translators sometimes fail to understand these different structures).

While modern Machine Translation (MT) speeds up development and localisation processes, it has the unfortunate side effect of completely ignoring cross-cultural nuance, differences in idioms, and references that make communication feel natural in those languages (Ukrainian Conference Proceedings, 2025). An automated system might translate an English colloquialism perfectly word-for-word, but the resulting phrase would likely make little-to-no sense to a native Turkish speaker. Worse still, machine translation frequently struggles with gendered languages, often defaulting to masculine pronouns and alienating female users and users of different genders, excluding even more groups in the process. Good localization requires an understanding of cultural differences, such as whether a culture relies on high-context or low-context communication, which may fundamentally change how information should be presented to the user. (Springer) According to Geert Hofstede's Cultural Dimensions, a culture with high "Uncertainty Avoidance" (such as Germany or Japan) prefers highly detailed, blunt, and exact error messages. Conversely, a high-context culture might find blunt error messages rude, preferring UI copy that is softer, more polite, and less direct. Failing to account for these differences in cultural norms can cause an experience that alienates users who find the software difficult to navigate as a result.

USER INTERFACE CONSIDERATIONS

Lastly, the localisation process requires paying attention to the actual design of the user interface. What do developers need to think about with regards to their User Interface (UI) when

they localize software? It is a fairly common misconception that the localization process only affects the backend text files of an application. In reality, translation also physically alters the visual layout of the software. Designing a UI that can adapt to the different constraints of languages it is localised for is vital because human languages can take up vastly different amounts of physical and visual space. According to IEEE research (5488634), designers must add approximately 30% extra space to each control in a dialogue box, as text strings are frequently longer when translated from English to other languages. This phenomenon, known as text expansion, is a nightmare for rigid front-end design. The English word "Settings" is compact and easily fits into a small navigation button. However, once it is translated into German, it becomes "Einstellungen." If the UI developer has hardcoded the width of that button, the German text will overflow, overlap with other visual elements, and end up rendering the navigation menu unusable. As such, interfaces that are designed to be localised cannot be perfectly rigidly defined, as they would become awkward or downright unusable once localisation occurs.

Additionally, the directionality of text—whether a language reads left-to-right, right-to-left, or top-to-bottom—forces software engineers to dynamically mirror and restructure the entire graphical interface, further preventing a one-size-fits-all design approach. Translating software into Arabic or Hebrew, then, requires more than just changing the alphabet; it requires implementing a Right-to-Left (RTL) interface. In an RTL UI, the entire visual hierarchy is flipped. "Back" arrows must point to the right. Progress bars must fill from right to left. Menus that slide out from the left edge of the screen must now slide out from the right. If the core architecture of the software was not built with this elasticity in mind from day one, attempting to retrofit an English LTR interface for an Arabic user base would require rewriting massive portions of the foundational codebase.

LITERATURE DISCUSSION

To fully grasp the complexities of modern software localization, one must look at the academic and historical foundations that have shaped the industry. Much of the research that we found on the subject supports the idea that these softwares have transformed into more of a practical application towards the focus of linguistics in software development.

THE HISTORICAL EVOLUTION OF LOCALIZATION

Early literature on the subject (Most were from the 1990s) highlights how drastically the approach to localization has shifted over the decades. The historical evolution of software localization shows a distinct move away from highly centralized, and a localized agenda for technical tasks. In the past, less localization was being done in-house, meaning the process was slowly becoming a broader and commonly outsourced global focus. This shift was predicted early on by researchers who saw the cultural implications of software. Some public entities want localized software to try to preserve against the melting of their culture into the American melting pot. Without software in their native languages, marginalized groups face a homogenized, English-default technology landscape that threatens their linguistic heritage. Not only that, but non-localized software pushed the world's digital divide into a more significant issue moving forward.

INTERNATIONALIZATION BEFORE LOCALIZATION

Before one can translate words and phrases into different languages, one must prepare the application's design. People want software in their native tongue because it's easier to understand. However, localization is very time consuming without the tools and knowledge to help. The most critical takeaway from foundational localization research is that developers must internationalise before localising. Internationalization is the architectural process of making the software flexible enough to handle different languages, whereas localization is the actual act of adding those specific languages in. Both of which are essential to building an application that can work in most of these languages.

COMPLEXITIES AND AUTOMATION

A recurring theme in modern linguistic research is the inherent danger of treating human language as a simple one-to-one cipher. Simple part-for-part translation leads to linguistic errors caused by the fact that languages usually aren't able to be directly translated between each other. This is due to many languages not having directly translatable words, or languages containing different sentence structures. With that in mind, differences in culture make certain phrases, idioms, or references make little sense to other individuals. To combat these time-consuming challenges, the industry has heavily leaned into automated systems for language translation. Machine translation is common today; it speeds up processes, but it also has the unfortunate side effect of completely missing any cross-cultural nuance (such as the English vs Japanese example explained earlier). Consequently, a key aspect of localization nowadays is trying to make sure the translations fit the culture it is intended to serve.

A fascinating example of this can be seen when viewing English software with a Turkish perspective. When looking at complex professional software, examples such as Photoshop are used to demonstrate how UI and feature names deeply rely on cultural context to make sense to the end-user. If a translator merely uses part-for-part translation for a largely popular photo-editing tool, the Turkish equivalent of the user interface may become confusing or entirely useless.

CULTURAL DIMENSIONS AND USER INTERFACE ARCHITECTURE

The literature makes it well known that cultural context must dictate UI design. Designers must add approximately 30% extra space to each control in the dialogue boxes as text strings can be longer when they are translated. Furthermore, the directionality of text radically alters the visual layout; some languages read left to right, others right to left, even up to down. It is essential to make an interface that can stay flexible with all of these attributes to make a fully functional translation within the application.

Beyond physical space, the psychological presentation of the software must be adapted. Culture is defined as "the collective programming of the mind that distinguishes the members of one group or category of people from others". When designing for a global market, developers must consider frameworks like Geert Hofstede's Cultural Dimensions, which include: Power Distance (PD), Individualism vs. Collectivism (IDV), Masculinity vs. Femininity (MAS), Uncertainty Avoidance (UA), and Long vs Short term orientation. Similarly, understanding

whether a target demographic relies on high-context or low-context communication fundamentally shifts how information, instructions, and error messages should be presented. To understand what is happening in the context of the errors, designers would consult with language and translation experts to determine the required space in each control. That way, they can understand the accurate measurement for each language, and build their interface with these measurements in mind.

BRINGING THE CODE AND THE LANGUAGES TOGETHER

The friction in software localization is often caused by a disconnect between the people writing the code and the people translating the text. Localization can be improved by having both developers and translators know more about each others' work. When software developers understand the linguistic limitations of their architecture, and linguists understand the structural constraints of the UI, the resulting application feels significantly more natural to the end-user. With this natural understanding of the design principles used to make an end user understand the application more, a seamless user experience becomes possible.

CONCLUSIONS/REFLECTIONS

The global expansion of software has proven that treating localization as an afterthought is a recipe for disaster regarding the design of applications with a multilingual design in mind. From the research and literature discussed, several core reflections emerge regarding how the tech industry must approach multi-lingual software development. Smaller projects with a limited amount of material can get away with static loc, or specific, pre-translated passages of text that can be stored. But the larger the project is, the greater the need for dynamic localization is. Hardcoding strings and fixing rigid UI boxes simply will not survive contact with a global user base where text expands by 30% or shifts reading direction entirely. It is essential to use these principles to design an application that solves the issues for all stakeholders, regardless of their own linguistic barriers.

In addition, the industry must play around the double-edged sword of automated translation. While automated translation can be an easier means of localization, it tends to ignore how actual speakers of those languages speak. In other words, good localization should be automated, but translated into the language as a native speaker would use it (with high accuracy, of course). Communication and cooperation with native speakers of the language you are translating to are vital in that regard. We cannot rely on machines to inherently understand the cultural nuances of high-context communication or localized idioms.

Finally, the decision regarding which languages to support cannot be driven solely by immediate profit margins. The choice of what languages we localise into should be context-dependent, with an understanding of who is likely to use the software. Simultaneously, organizations should try to cover as much of the potential user base as it would be cost-effective to do so. Balancing economic viability with the social responsibility of digital accessibility ensures that technology does not unintentionally erase the cultures it is meant to serve.

To create software that is truly accessible to people around the world, we must localise in a way that is aware of the language, culture and norms of the people we are localising for; must

build flexible, changeable user interfaces that can adapt to the complexities of human communication, and design the software to be able to cater to the diverse populations we are building our products for.

WORKS CITED

- Bass, M. (2004). Global Software Development Process Research at Siemens. “*Third International Workshop on Global Software Development (GSD 2004)*” *W12S Workshop - 26th International Conference on Software Engineering, 2004*, 8–11.
<https://doi.org/10.1049/ic:20040304>
- Beckett, G. H., & McGivern, L. (1999). *Dilemmas in designing multimedia software for learners of English as a second or foreign language*. International Society for Technology in Education.
- Cabezas, C., Dorr, B., Resnik, P., & Maryland univ college park inst for advanced computer studies. (2001). *Spanish Language Processing at University of Maryland: Building Infrastructure for Multilingual Applications*. Maryland univ college park inst for advanced computer studies.
- Hau, E., & Aparício, M. (2008). Software internationalization and localization in web based ERP. *Proceedings of the 26th Annual ACM International Conference on Design of Communication*, 175–180. <https://doi.org/10.1145/1456536.1456570>
- Keniston, K. (1997). *Software localization: Notes on technology and culture*. Program in Science, Technology, and Society, Massachusetts Institute of Technology.
- Matthew. (2024, November 25). *Creating multi-language support in full-stack development* | by Matthew | *Medium*. Creating Multi-Language Support in Full-Stack Development.
<https://medium.com/@mdburkee/creating-multi-language-support-in-full-stack-development-3066d6abf833>
- Maumevičienė, D. (2017). Lokalizacija Kaip Komunikacijos Aktas. *Vertimo Studijos*, 4(4), 95.
<https://doi.org/10.15388/vertstud.2011.4.10576>
- Mick, G., & Pym, A. (2015). *The role of revision in English-spanish software localization*. Universitat Rovira i Virgili.
- Ressin, M. (2015). *An empirical examination of interdisciplinary collaboration within the practice of localisation and development of international software*. University of West London.
- Rivas-Ginel, M. I., Gautier, L., Corpas Pastor, G., Froeliger, N., Seghiri, M., Monti, J., & Faria Pires, L. de. (2023). *The ergonomics of Cat Tools for video game localisation*.
- Taylor, D. (1992). *Global software: Developing applications for the International Market*. Springer-Verlag.

YAZICI, Y., & HAYTA, P. (2021). Investigation of the relationship between computer programs and foreign language used in graphic design process. *Yıldız Journal of Art and Design*, 8(1), 43–52. <https://doi.org/10.47481/yjad.811071>