# ScreenMatch: Providing Context to Software Translators by Displaying Screenshots

**Geza Kovacs**
MIT CSAIL
32 Vassar St, Cambridge MA
02139 USA
gkovacs@mit.edu

Copyright is held by the author/owner(s).
*CHI 2012*, May 5–10, 2012, Austin, TX, USA.
ACM 978-1-4503-1016-1/12/05.

## Abstract
Translators often encounter ambiguous messages while translating software. To resolve ambiguity, the translator needs to understand the context in which the message appears. Currently, context is provided via textual descriptions, or not at all. This paper describes ScreenMatch, a system which provides translators with visual context for each translatable message. It does so by matching each message with a corresponding screenshot of the application. ScreenMatch consists of a tool to gather screenshots, an algorithm to match messages to screenshots, and an interface that presents translators with screenshots alongside messages. We evaluated the system by gathering screenshots for 3 applications, using the algorithm to match messages to screenshots, and comparing results to manual matches. We found that hard-to-reproduce error messages make it difficult to gather all the screenshots. The algorithm correctly matched messages to screenshots 80% of the time when a corresponding screenshot had been gathered.

## Keywords
internationalization, translation, ocr, text matching

## ACM Classification Keywords
H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work.

## Introduction

The software translation process begins with the developer marking strings in the source code as being translatable. These strings are called *messages* – translatable units of text, such as labels or tooltips. The developer then runs a tool that generates, for each language, a *message file* – a file containing a list of messages and placeholders for corresponding translations. The message files are then sent to the translators, who use a tool such as Gtranslator to view the messages, and fill out the translations.

User interfaces are full of ambiguous messages which can have different meanings depending on the context they are being used in. For example, "Open Door" can be either a command (Please open the door) or status notification (The door is open). When translating to a foreign language, these different meanings may correspond to different phrases – hence, the translator will need to know the context the message was used in, in order to determine the correct translation.

Translation tools currently provide little context to translators – they display only the message itself, and any comments that are present in the message file. Comments are often automatically added during the process of generating the message file, to indicate the source of a message. This sometimes provides some context for the translator – for example, if the message source is "cc-printers-panel.c", the message is presumably related to printers. Developers sometimes write additional comments to clarify ambiguous messages, as we can see in Figure 1. Unfortunately, writing such notes for translators is a burden on developers, and is only rarely done.

```
#. Translators: One or more doors on the printer are open
#: ../panels/printers/cc-printers-panel.c:536
msgid "Open door"
msgstr "扉が開いています"
```

**Figure 1:** Part of the message file for the Japanese translation of Gnome Control Center. Contains the message "Open Door", comments for translators, and a translation.

Screenshots of the user interface are a source of context that might be useful to translators. For example, if we see the message "Open Door" used in a screenshot, appearing as an informational dialog in the notifications area, it would be clear that the message is a status notification.

This paper describes ScreenMatch, which is used to provide context to translators by showing screenshots. To use the ScreenMatch system, the developer first gathers screenshots for the application. Each message is then algorithmically matched to a screenshot illustrating its usage. When the translator translates the application, the corresponding screenshot will be shown alongside each message.

## Related Work

Facebook Translations [1] and Qt Linguist [2] both provide translators with visual context – however, their approaches differ from ScreenMatch's screenshot-based approach, and are less generally applicable.

Facebook Translations uses the approach of allowing users to translate the site while using it. Because the translators are translating the website in the process of actually using it, they are able to see the context in which the text is being used. Having users translate software while using it,

[1] http://www.facebook.com/apps/application.php?id=4329892722
[2] http://developer.qt.nokia.com/doc/qt-4.8/linguist-translators.html

however, is a radical departure from the traditional model of software translation, and does not fit in well with the message-by-message translation workflow used by professional translators. Furthermore, translators need to make a conscious effort to discover all the translatable text, and it is not apparent which parts of the application still need to be translated. Additionally, this approach requires the ability to determine the widget that was clicked and its associated text – information that is readily available via the Document Object Model (DOM) for web applications, but which cannot be obtained from desktop or mobile applications.

For software whose GUI was constructed using the Qt Designer interface builder (which creates a *UI file*, a file that declaratively describes the layout and forms in a window), Qt Linguist will show translators a preview generated from the UI file, for messages which originated from a UI file. One limitation of this approach is that the visual context can only be shown if the message originated from a UI file built using Qt Designer. Another limitation is that because the preview of the UI is generated from the UI file rather than from the running application, it may bear little resemblance to how it actually looks like in the application – particularly if there is much programmatic manipulation of the interface.

Thus, unlike the approach used by Facebook Translations, ScreenMatch's message-file-centric approach fits well with existing translation workflows. Unlike Qt Linguist, the visual context is generated from the running application, and hence is also available for interface components built outside the interface builder. Finally, unlike both Qt Linguist and Facebook Translations, ScreenMatch is toolkit and platform-independent, because screenshots can be taken of any user interface.

Similarly to our work, Prefab (Dixon 2011) also extracts translatable text from the pixel representation of user interfaces. However, we use the text extracted from the user interface for a different purpose – whereas Prefab sends it to a machine translation service to translate the application in-place, ScreenMatch uses it to assist software translators.

## Implementation

ScreenMatch consists of 3 components:

1. A tool to gather screenshots (Screenshot Gatherer)
2. A program to match messages to screenshots (Matcher)
3. An interface for translators that presents screenshots alongside messages (Translation Editor)

To use ScreenMatch, the developer first obtains all the screenshots using the Screenshot Gatherer. These screenshots, and the message file, are then supplied to the Matcher, which outputs an annotated message file, which contains, for each message, the screenshot which best matched the message. This annotated message file is then supplied to the Translation Editor, which displays the best-matching screenshot for each message, with the matching region highlighted.

*Screenshot Gatherer*

The Screenshot Gatherer assists a developer in gathering screenshots for the application. The developer supplies the Screenshot Gatherer with a message file, and it takes screenshots while the developer uses an application, attempting to ensure that all translatable messages are included in at least one screenshot. The Screenshot Gatherer is not necessary to use ScreenMatch – one could gather the screenshots manually – however, it is useful

because it automates the process of taking and saving screenshots, and provides feedback on the progress by indicating which messages still need a matching screenshot.

The Screenshot Gatherer constantly takes screenshots of the current active window. Each time a screenshot is taken, it uses the Optical Character Recognition (OCR) engine from Sikuli (Yeh 2009) to determine what text appears in the screenshot. Then, it matches the messages to the OCR-ed text, to see which messages appear in the screenshot. If the current screenshot contains a new message which wasn't present in any existing screenshots, it will be saved. As shown in Figure 2, messages which haven't been matched are listed, grouped according to the source indicated in the message file (which roughly correspond to dialogs). This allows the user to easily determine when they have gathered all screenshots for a dialog (or if they missed part of a dialog).

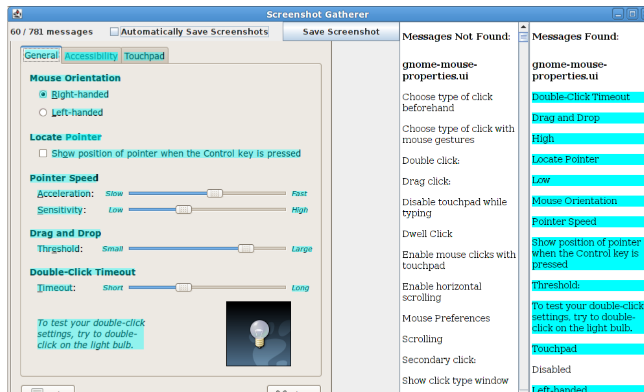



**Figure 2:** Screenshot Gatherer – screenshot of active window is on left, matched messages are on the rightmost column, messages that need to be found are in the center column

*Message-Screenshot Matcher*

The Matcher program is supplied with a message file, and a list of screenshot files. It annotates the message file, matching each message to a screenshot.

The Matcher first uses OCR to extract a list of words in each screenshot. Then, for each message, the longest common subsequence is computed between the message and every subsequence of words in every screenshot. The match score is the length of the longest common subsequence, divided by the length of the longer of the two strings.

It is necessary to match against substrings of the OCR-ed text, because it is difficult to accurately determine how the words should be grouped together. For example, if the screenshot contains "Updates are ready to be installed." followed by "Install now?" on the next line, this could either correspond to one message that is wrapped across two lines, or two separate messages on separate lines – we need to try matching against both possibilities.

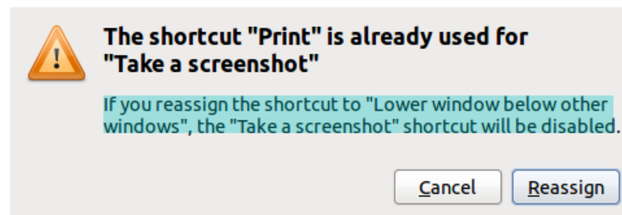
If there is little whitespace separating the matched substring from nearby text in the screenshot, a penalty is applied to the match score. This ensures that individual words don't get picked out and matched from a larger body of text. For example, if we are attempting to match the message "Web" and there is one screenshot containing "Web" in isolation, and another screenshot containing "Web Browser", then "Web" in isolation will be preferred.

Some messages contain placeholders which may be substituted by different text at runtime. For example, the message "Install %s now?" might appear in the application as "Install Firefox now?'. Because the substituted text can be arbitrarily long, we use a modified version of the longest common subsequence algorithm,

which does not count substituted text in the total length, on such messages.

Once match scores have been calculated for a given message, the highest one is selected, and if it is above the threshold, then the message is matched to the screenshot.

*Translation Editor*
Translation Editor takes the annotated message file generated by the Matcher, and displays the matched screenshot alongside each message (see Figure 3). We envision that this functionality will eventually be integrated into existing translation tools like Gtranslator.




**If you reassign the shortcut to "#{Lower window below other windows}", the "#{Take a screenshot}" shortcut will be disabled.**

Si reasigna la combinación a «%s» se desactivará la combinación «%s».

#: ../capplets/keybindings/gnome-keybinding-properties.c:1207
msgid "If you reassign the shortcut to \"%s\", the \"%s\" shortcut will be disabled."

**Figure 3:** Translation Editor – Screenshot is shown above, with the matching portion highlighted.

## Evaluation

To evaluate the accuracy of the Matcher, we gathered screenshots for three different applications, and performed the matching between messages and screenshots manually. These screenshots and the message file were then provided to the Matcher, and the results of the manual matching and algorithmic matching were compared.

The three applications used for evaluation were the Gnome Control Center, the XFCE Terminal Emulator, and the PCManFM file manager. These particular applications were chosen because they represent a diverse set of application types, and their message files are readily available.

We first gathered the screenshots for the three applications, using the Screenshot Gatherer. The screenshot gathering procedure took about two hours for the Gnome Control Center, and about one hour for each of the other two applications.

Not all of the screenshots for the applications could be gathered. Of the messages which did not appear in any screenshots, many were hard-to-reproduce error messages. Other reasons for missing messages were related to the machine configuration. For example, about 100 of the missing messages for Gnome Control Center corresponded to a fingerprint reader configuration window, which was inaccessible as we did not have access to a fingerprint reader.

Surprisingly, some of the messages we could not find screenshots for turned out to be generated from dead code – for example, a tab had been removed from the Gnome Control Center, but the patch which removed it had forgotten to remove associated lists of options.

| Application | Total Messages | Total Messages in Gathered Screenshots | Number of Screenshots |
|---|---|---|---|
| Gnome Control Center | 783 | 397 (51%) | 155 |
| XFCE Terminal Emulator | 275 | 222 (81%) | 94 |
| PCManFM File Manager | 153 | 124 (81%) | 36 |

**Figure 4:** Messages for which screenshots could be gathered.

We then performed matching between the messages and screenshots manually. For each message, we listed which screenshots, if any, illustrated the message in use. The Matcher was then used to match each message to a screenshot. The match could be one of the following:

- Correctly matched: The message was matched to a screenshot which had been manually listed.
- False positive: The message was matched to a screenshot which had not been manually listed.
- False negative: The message was not matched to any screenshot, though it had appeared in a screenshot.

Among the messages that were represented in the screenshots, 80% overall were correctly matched. Many of the false negatives corresponded to bold text, or text on backgrounds (window decorations or buttons), which the OCR system wasn't able to read correctly. These types of errors could be reduced by retraining the OCR system on computer screenshots. Many of the false positives resulted from similar messages. For example, the XFCE Terminal Emulator message file contains both "_Copy" and "Copy". However, both appear as "Copy" in the interface – the underscore in "_Copy" indicates that the shortcut is Ctrl-C, and is only rendered as an underline under the C, which is ignored by the OCR system.

| Application | Total Messages in Gathered Screenshots | Correctly Matched | False Positives | False Negatives |
|---|---|---|---|---|
| Gnome Control Center | 397 | 341 (86%) | 16 (4%) | 40 (10%) |
| XFCE Terminal Emulator | 222 | 159 (72%) | 37 (17%) | 26 (12%) |
| PCManFM File Manager | 124 | 99 (80%) | 9 (7%) | 16 (13%) |

**Figure 5:** Breakdown of matched messages.

## Conclusion

We have built ScreenMatch, a system which provides translators with a screenshot illustrating how the message being translated appears in the software. As we have seen, matching messages to screenshots using only the English text extracted from the screenshots via OCR works reasonably accurately. The primary difficulty encountered is reliably gathering all the screenshots for the application.

As seen from the inaccessible fingerprint reader configuration window in Gnome Control Center, and unreproducible error messages, one may need to perform screenshot gathering on various machine configurations to get all the screenshots. Thus, crowd-sourcing the screenshot gathering process to users may be more effective in achieving complete coverage of screenshots.

In addition to the translation application we have described, this system could also be used for identifying dead code. Namely, if a message cannot be found in any screenshot, this would suggest that the code it was generated from is never executed. Identifying and removing this dead code could make the codebase more robust, and reduce the burden on translators, as they would no longer need to translate these messages.

## Acknowledgements

This work was made possible by Tsung-Hsiang Chang's assistance with using Sikuli, and Professor Robert Miller's mentorship.

## References

[1] Dixon, M., Leventhal, D., and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. CHI 2011, 969-978.

[2] Yeh T., Chang, T., and Miller, R. Sikuli: Using GUI Screenshots for Search and Automation. UIST 2009, 183-192.