1. はじめに --- 概要

1. はじめに
2. 定義
3. 意図的な重複 (必要な重複)
4. 意図的な重複を意識せずに翻訳を進めるとどうなるか
5. 複数の訳文を登録する (属性を使わない、Trados 標準の方法)
6. 意図しない重複 (不要な重複)
7. 不要な重複が蓄積した翻訳メモリの例
8. メモリの掃除
9. 属性を使った重複
10. 実用面での問題
11. まとめ

2. 定義

重複する翻訳メモリとは

同じ原文に対して複数の訳文が登録されているメモリのことで、**翻訳者が意図した必要な重複**と、**意図しない／不要な重複**に分けられます。

3. 意図的な重複 (必要な重複)

定義: 状況に応じて訳文を使い分けるために翻訳者が意図的に複数の訳文を登録している。

ハンドアウトの 1 ページを見てください。

CASE1. クライアントから指定されたスタイルの仕様上、1 つの用語に何通りかの訳出が必要

例えば、ソフトウェアはローカライズせず、マニュアルのみを翻訳するプロジェクトがあるとします。画面上の UI は英語のままなので、対象製品の UI の訳出は [English (日本語訳)] というスタイルで訳出するようクライアントから指定されています (これが A です)。

マニュアルの中で、製品のインストール手順などで Windows の UI が登場しました。翻訳の使用者は日本語版の Windows を使用するはずなので、Windows の UI は訳出が必要です。「Finish」ボタンなどのような一般的な UI で、たまたま対象製品の UI と同じ原語のものがあったという場合、Windows の文脈に従ってこれは訳出することになります (これが B です)。

このようにして、同じ原文に対して A と B の二通りの訳を使い分ける必要が発生します。

CASE2. 原文が FrameMaker 形式で、索引マーカテキストの訳出が必要

FrameMaker では、索引や目次ファイルは専用のコマンドを使って自動生成されます。索引を持つマニュアルでは、各章のファイルの本文の要所要所に索引マーカという、テキストを含んだタグが挿入されており、索引はこれらのマーカのテキストをソートして生成されます。日本語の場合、索引マーカテキストには読み仮名を入力しておかないと、漢字などを含む索引は正しくソートされません。このため、Trados では<:so> (ソートオーダータグ) という専用のタグが用意されています。

このケースでは、原文はまったく同じでも、A の原文は章の見出し、つまり本文用のテキストなので、そのまま訳出する必要があるのに対し、B は索引用のテキストであるため、<:so>タグを使って読み仮名を入れる必要があります。

意図的な重複が必要となるケースとしては、このケースが大半を占めています。私が大型のマニュアルの翻訳メモリの管理を意識するようになったのは、このような、索引とそうでないテキストの使い分けが

大きな理由です。

4. 意図的な重複を意識せずに翻訳を進めるとどうなるか

以上のような場合に、Trados で普通に翻訳を進めてしまうとどうなるかを見てみます。ハンドアウトの 6 ページを見てください。

状況に応じて 2 通りの訳出をしていたとしても、訳文をそのまま登録していたのでは、後からメモリに登録した訳文によって最初に入力した訳文が上書きされてしまいます。この状態で、後でこのメモリを再利用して自動翻訳した場合、この例のように、原文は<heading 2>というタグが付いていることからわかるように本文の章見出しであるのに、<:so>タグと読み仮名の付いた索引マーカ用の訳文が 100%一致として取得されてしまいます。索引は<imk>と</imk>タグで囲まれて識別されており、索引以外の場所で<:so>タグを使用すると、S-Tagger でタグ検証を行った際、エラーが出ます。逆に、索引マーカテキストに、本来は本文テキスト用に訳出していた読み仮名のない訳文を取得すると、FrameMaker に変換した際に、索引が正しく生成されなくなります。

5. 複数の訳文を登録する (属性を使わない、Trados 標準の方法)

もちろん、ご存知の方も多いと思いますが、Trados では 1 つの原文に複数の訳文を登録する機能がサポートされています。方法としては、メモリの作成時に、設定ダイアログで「1 つの原文分節に複数の訳文を許可／Allow multiple translations for identical source seguments」というオプションを選択しておき、既存の訳と違う訳を追加する際には、通常の登録を行う代わりに、このように (*スライドに図挿入*) 「新しい翻訳を追加」コマンドを選択して翻訳を追加登録します。

この方法でも、問題なく複数の訳文を使い分けることは可能です。Trados のマニュアルやセミナー、出版物でも、単一の原文に複数の訳文を登録する方法としてこの手順が説明されています。

ですが、翻訳メモリの再利用性の面で、この方法にはいくつかの問題点があります。これらの問題点について触れる前に、先ほど定義したもう 1 つの重複、意図しない重複 (不要な重複) について見ていきたいと思います。

6. 意図しない重複 (不要な重複)

ワード数の大きいプロジェクトを短期間で仕上げるため、複数の翻訳者による分業を支援するのも Trados が提供する利点の一つとしてうたわれています。外注の翻訳者同士で分業する場合、互いに翻訳の一貫性を保つため、メモリのスワップを行います。メモリのスワップとは、各翻訳者が定期的に、前回のメモリスワップから現在までの作業分の翻訳メモリをエクスポートし、お互いに提供し合うことを言います。他の翻訳者から受け取ったエクスポートファイルは、自分の作業メモリにインポートして使います。このとき、知らないうちに必要なメモリが上書きされてしまうのを防ぐため、「Merge／結合」インポートオプションを選んでインポートするのが通例です。意図しない重複は、こうした作業の弊害として発生します。ハンドアウトの 2 ページを見てください。

CASE 1. 翻訳作業を開始してしまってから、当初指定されていた訳語に変更があった場合

時間的な余裕のないローカライズのプロジェクトでは、チームの一部が UI を翻訳している間に、他のメンバーはマニュアル本文の翻訳を前倒しで開始する場合があります。このような場合、UI の翻訳がまだ確定していないうちは、マニュアル本文中の UI は仮に英語のままにしておくように指示があったりします (A)。

プロジェクトの中盤になり、翻訳された UI がクライアントのレビューを経て確定され、UI のリストが翻訳者に配布されました。英語のままにしてあったマニュアル本文中の UI を、翻訳者各自で確定した訳に置き換える作業が発生します (B)。

自分の担当部分の修正に関しては、既存の訳を修正後の訳で上書きするだけなので問題はないので

すが、例えば、この例文の修正が自分の担当部分以外で行われていて、A の訳の間にメモリスワップが行われ、B に修正した後再びメモリスワップを行ったとします。あなたは修正前のメモリと修正後のメモリをいずれも Merge／結合インポートしているため、手持ちのメモリでは A も B も保持され、複数候補が発生します。

CASE 2.　細かいスタイルの指定が事前になかったために、同じ原文に訳者によって異なる訳出が発生

「～については、～を参照してください。」といった定型文は、3 人の翻訳者が任意に訳出すれば 3 通りの訳ができてしまいます。後で気が付いて表現を統一したとしても、すでにメモリスワップを重ね、Merge／結合を行っていれば、TM の使用者が明示的に削除しない限り、修正前の訳文もそのまま残り、複数候補が蓄積されていきます。

CASE 3.　他の翻訳者の訳文に入力ミスがあったが、後になってから本人が気づいて修正している

修正した本人のメモリでは、同じ訳文に修正が加わりますが、修正する前にすでに 1 回メモリスワップを行っていれば、Merge／結合オプションでインポートしている他の翻訳者のメモリでは、修正前の訳文も削除されないまま残ります。

7. 不要な重複が蓄積した翻訳メモリの例

ハンドアウトの 3 ページと 4 ページは、こうした不要な重複が蓄積した翻訳メモリを表しています。まず 3 ページを見てください。

CASE1.　他の翻訳者の訳文に入力ミスがあったため、修正前後のメモリが混在している

最初の例は、先ほどの「CASE 3.他の翻訳者の訳文に入力ミスがあったため、修正前後のメモリが混在している」を再現したものです。

実際にはこの原文に対応する訳文は 1 つしか存在しないはずなのに、入力ミスを修正する前のメモリがそのまま残っているために複数の訳文が存在しています。複数の訳文の存在はメモリオプションのデフォルトで 1%のペナルティが付くので、後でこのメモリを使って自動翻訳をしても、この訳文は完全一致（100%）と見なされず、翻訳者が手動で取得する作業が必要になり、生産性ツールとしての Trados の機能性が生かされないということになります。

CASE 2.　細かいスタイルの指定が事前になかったために、同じ原文に訳者によって異なる訳出が発生

4 ページを見てください。カタカナ複合語や、長音引きは、事前に細かく指定しておかないと、3 人三様の訳語ができてしまいます。この例では、user interface というごく一般的な用語を、

A).　翻訳者 A は「ユーザー（半角スペース）インターフェイス」、

B).　翻訳者 B は「ユーザインタフェース」、

C).　翻訳者 C は「ユーザー（半角スペース）インターフェース」と訳出しています。

最終的に、カタカナ複合語には中黒（・）を使用することになり、user interface は「ユーザ・インターフェイス」で統一することに決定しました。修正後の訳が D です。

このような場合、異なる訳出を含んだメモリのスワップ、修正後のメモリのスワップを重ねた結果、この図に示すように、実際には必要な訳文は 1 つだけなのに、同じ原文に 4 種類の訳文が蓄積されることになります。このメモリを再利用するとなると、いちいちボタンで候補を順にスクロールしなければならないなど、訳文を取得する作業も煩雑になります。また、後になってから、この翻訳作業には関っていなかった第三者がこのメモリを使用することになったとき、どれが正しい訳なのか、メモリを見ただけではわかりにくく、新たな訳出ミスのリスクも発生します。

8. メモリの掃除

以上のようにして蓄積してしまった不要な重複を排除し、再使用可能な翻訳資産としての翻訳メモリの価値を高めるためには、プロジェクトのキリの良いとき、あるいはプロジェクトが完了し、翻訳が確定してから、メモリを掃除するという作業を行います。元のメモリと同じ設定を用いて空のメモリを新たに作成し、ワークベンチの「Translate／翻訳」オプションで「Update TM／TM を更新」を選んで最終版の Unclean ファイルに走らせます。これで、実際のファイルで使われている翻訳のみが原文文節と対になってメモリに入ります。

この作業は他にも、プロジェクトの途中にクライアントから用語やスタイルの一括変更の要請が入り、対象の変更箇所があまりにも多いといった場合に有効です。たとえば、先ほどの「ユーザ・インターフェイス」のようなカタカナ複合語の中黒を、やはり MS スタイルの半角スペースにして欲しい、といった修正が入った場合、中黒をワード画面で検索し、ヒットしたセグメントをワークベンチで開いて修正、翻訳を登録、といった作業をただひたすらに繰り返す代わりに、まずワードの一括置換で中黒をスペースに変換してしまい、このファイルに前述の手順で空のメモリを走らせてメモリを作り直せば、短時間でメモリとファイルの両方に修正を反映できます。

ただし、ここで 1 つ大きな問題が生じます。メモリの中に、意図していなかった不要な重複だけでなく、翻訳者が意図的に複数登録しておいた必要な重複も含まれている場合、このメモリ掃除プロセスによって、意図していた重複までも削除され、ハンドアウトの 6 ページに示した例のような状況に逆戻りてしまうということです。これが、先ほど「意図的な重複」の最後に述べた問題点です。

9. 属性を使った重複

この問題を回避するために、私は属性を使って複数の翻訳を登録しています。属性は、この図のように、翻訳メモリを任意のカテゴリに分類します。設定方法については、時間の都合上割愛しますが、ハンドアウトの最後のページに簡単な説明を付録として付けてありますので、そちらを参考にしてください。詳しい手順は、Trados のマニュアルを参照してください。

属性、属性値の名前はユーザーが指定できます。この例では、ND、MainText、IndexMaker の 3 つの属性値を設定しています。ND は、No-Duplicate の略で、重複がなく訳文候補が 1 つしかない分節に使用します。MainText は重複する訳文候補で本文テキストに使われる訳、IndexMaker は索引に使われる訳です。索引については、訳文に原文にはないタグが挿入されるという特別なケースであるため、重複していてもしていなくても必ず IndexMaker 属性を付けるようにしています。

このようにして翻訳メモリに属性を設定しておき、さらにワークベンチの「プロジェクトとフィルタの設定」でフィルタを設定し、登録した訳文分節のすべてに自動的に属性値 ND が付き、また既存の訳文については、属性値 ND が付いているもののみを 100%一致とみなし、それ以外の属性値にはペナルティが付くようにしておいて、翻訳作業を進めます。

このようにして作成した翻訳メモリがハンドアウトの 7〜8 ページの例です。

CASE 1. [English (日本語訳)] というスタイルが指定された UI と、完全に日本語化されている Windows UI が混在 (I-CASE1)

属性のペナルティ (2%) が付くことで訳文は 98%一致となり、自動翻訳をしたとしてもこの原文は処理対象にはなりません。必ず翻訳者の手を経なければ翻訳は行われないので、翻訳者がコンテキストを見ながら適切な訳文を手動で取得できます。また、属性を設定するとき、テキストフィールドにメモを入れておけば、後で第三者が見た場合でも、それぞれの訳文がどのような状況を意図しているものかがわかりやすくなります。

CASE 2. 同じ原文が索引マーカテキストと本文の両方で使われている (I-CASE2)

属性のペナルティ (2%) が付くことで訳文は 98%一致となり、自動翻訳をしたとしてもこの原文は処理対

象にはなりません。また、第三者がこのメモリを使った場合でも、属性の名前から、それぞれの訳文がどのような状況を意図しているものかがわかりやすくなります。

10. 実用面での問題

このように、属性は便利なのですが、実際には、すべての Trados 関連の仕事でこの設定を活用することはできないのが実情です。

個人の翻訳者としては:

- TM を共有している翻訳者全員が同じ設定を使わないと属性は意味がない。

- 1 人のサブコントラクタとしての立場で、同じ立場の他の翻訳者の人たちに、自分のやり方を指示するわけにはいかない。

エージェンシーの指示で実行するとしたら ：

- 課金体系はどうする? 100%、98% (再利用) メモリは現在の Traods 料金体系ではほとんど料金が発生しない → なのにキーストロークは余分に手間がかかる。

- 最終的にはエージェンシーの利益となるメモリ資産の管理作業の一部を翻訳者の責任ですることに →これについての料金は？

エージェンシーとしては:

- 日本語、翻訳、Trados の知識が豊富なインハウスの担当者、プロジェクトリーダーが指揮を執る必要がある

- 翻訳者のサポート、ミスの管理はどこまでできる？

- Trados に熟練した翻訳者の確保

11. まとめ

- Trados は生産性ツール。細かい機能をマスターして、Trados を味方に。

- 直接クライアントを開拓し、大きな仕事を翻訳者が直接取ってこなすことも可能

- エージェンシーも、「大きい仕事が速く安く上がる手段」として捉えるのではなく、Trados を使った作業に隠されている以上の問題点を認識して欲しい。

- Trados は技能、という認識が浸透し、エージェンシーがもっと高い水準での Trados 活用を要求し、同時に、それについて相応の料金が支払われると良い