Kapitel 8

Begriffliche Grundlagen der Softwarelokalisierung

Alexander Behrens
Universität Leipzig

Softwarelokalisierung war zu Beginn der ersten Globalisierungswelle in den achtziger Jahren eine Tätigkeit, die technisches Verständnis voraussetzte und entsprechend von nur wenigen Übersetzungsdienstleistern erbracht wurde, alternativ von Tandems, die sich aus Übersetzungsdienstleistern und Entwicklern zusammensetzten. Dass Softwarelokalisierung zu einem Massenmarkt werden konnte, liegt weniger an einer gestiegenen IT-Expertise unter Translatoren denn an der Tatsache, dass die Softwareentwicklung Werkzeuge und Dienstleister (namentlich Lokalisierungsingenieure) hervorgebracht hat, die dem Translator die extralinguistische Seite der Softwarelokalisierung abnehmen. Diese Entwicklung hat den Begriff der Softwarelokalisierung zunehmend verzerrt. Eine begriffliche Bestandsaufnahme wird umso nötiger, je leistungsfähiger und paternalistischer die Werkzeuge werden. Gegenstand vorliegenden Beitrags soll die am Rohprodukt Software erbrachte menschliche Dienstleistung sein.

1 Globalisierung (G11N)

Der Terminus Globalisierung, abgekürzt auch mit dem Numeronym G11N für *globalization*, bezeichnet im GILT-Paradigma alle auf die Erschließung von Auslandsmärkten abzielenden unternehmerischen Aktivitäten.[1] Aufgabenfelder der Globalisierung sind u. a. Standortentwicklung, Finanzplanung, Entwicklungsplanung, Marktforschung, Vertriebs- und Lieferantenmanagement. Dieser Globalisierungsbegriff hat keinen fachlichen Bezug zum Produkt Software und existiert

[1]Die Abkürzung GILT umfasst die vier Begriffe Globalisierung – Internationalisierung – Lokalisierung – Translation (siehe Fry & Lommel 2003: 6).

Alexander Behrens. 2026. Begriffliche Grundlagen der Softwarelokalisierung. In Oliver Czulo, Martin Kappus & Felix Hoberg (Hrsg.), *Digitale Translatologie*, 115–130. Berlin: Language Science Press. DOI: 10.5281/zenodo.17523044

losgelöst von den übrigen drei Begriffen des GILT-Paradigmas. Dass er trotzdem Eingang ins Begriffssystem gefunden hat, wird vornehmlich terminographischen Überlegungen geschuldet gewesen sein, denn bis dahin wurden die Ausdrücke *Globalisierung* und *Internationalisierung* uneinheitlich verwendet, oft genug synonymisch. Eine normative Gegenüberstellung der zwei war aus damaliger Sicht also durchaus sinnvoll.

2 Internationalisierung (I18N)

Der Terminus Internationalisierung, abgekürzt I18N für *internationalization*, wird in der Literatur bis heute uneinheitlich verwendet und entzieht sich deswegen einer allgemeingültigen Definition (siehe hierzu Behrens 2016: 45–64). Dies mag nicht zuletzt der Tatsache geschuldet sein, dass unterschiedliche Produkte unterschiedlich internationalisiert werden müssen. Bei der Betrachtung materieller wie immaterieller Produkte lassen sich grob zwei Internationalisierungstypen unterscheiden: Lokalisierbarkeit und Universalität.

2.1 Internationalisierungstypen

2.1.1 Internationalisierungstyp Lokalisierbarkeit

Der am stärksten beanspruchte Internationalisierungsbegriff beinhaltet jenen Entwicklungsschritt, der ein Produkt für die Lokalisierung (zum Begriff siehe Abschnitt 3) vorbereiten – es also lokalisierbar machen – soll. Vertreter dieses Internationalisierungsbegriffs sind u. a. Schmitz & Wahle (2000) sowie Fry & Lommel (2003: 14) . Göpferich (2002: 343–366) bespricht diesen Begriff unter der Bezeichnung *Internationalisierung mit dem Ziel einer anschließenden Lokalisierung (Fall b)*. Diese Form der Internationalisierung wird vorzugsweise durch eine modulare Gestaltung des Produkts erreicht: Ein marktneutraler Produktkern wird von marktabhängigen Komponenten technologisch getrennt. Ein auf diese Weise modularisiertes Produkt kann ohne Interferenzen zwischen den einzelnen Gewerken – zwischen jenen, die am Produktkern arbeiten und jenen, die für dessen marktabhängige Komponenten zuständig sind – entwickelt, verändert und produziert werden.

2.1.2 Internationalisierungstyp Universalität

Ein anderer, in der Literatur etwas vernachlässigter Internationalisierungsbegriff beinhaltet das Überflüssigmachen des Lokalisierungsschritts. Göpferich (2002:

340–342) bespricht diesen Begriff unter der Bezeichnung *Internationalisierung ohne anschließende Lokalisierung (Fall a)*. Diese Form der Internationalisierung wird durch Eliminierung marktabhängiger Merkmale im Produkt erreicht. Universalität entsteht etwa durch den Verzicht auf verbale Informationen, wie wir es von Icons und Piktogrammen kennen.

2.2 Internationalisierung von natürlicher Sprache

2.2.1 Internationalisierungstyp Lokalisierbarkeit

Internationalisierung von natürlicher Sprache bedeutet im Lokalisierungskontext zunächst allgemein die Herstellung von Übersetzungsgerechtheit durch den Einsatz kontrollierter Sprache. Je nach Regelwerk mag Übersetzungsgerechtheit etwa die Absenkung des Präsuppositionsniveaus, die Vermeidung von Implikationen, Realia, Regionalismen, Idiolektismen und Suprasegmentalia bedeuten, ferner Konnotationsfreiheit, die Einhaltung lexikalischer und syntaktischer Verbote und Konsistenz im Fachlichkeitsgrad. Internationalisierung von natürlicher Sprache ist schließlich auch die Sicherstellung terminologischer Eineindeutigkeit und Konsistenz in der Bedienoberfläche und in der Beziehung zwischen der Bedienoberfläche und den Hilfeseiten eines Programms.

Zur Verwendung von kontrollierter Sprache in Softwareoberflächen äußern sich Beste (2006: 53–55) und O'Brien (2019).

2.2.2 Internationalisierungstyp Universalität

Das Prinzip der Universalität kommt zunächst überall dort zur Anwendung, wo auf eine Lingua franca ausgewichen wird; es kann aber ebenso auf Nomina propria angewandt werden. Weil der Glaubenssatz „Nomen est omen" auch für zufällige Assoziationen gilt, sollten Produktnamen möglichst universal sein (*culturally sensitive branding*), was u. a. die Freiheit von lautlicher Ähnlichkeit mit ungewollten Botschaften einschließt. Manche Hersteller unterziehen ihre Produkte deswegen vor der internationalen Vermarktung einer sprachlichen Unbedenklichkeitsprüfung.

2.3 Internationalisierung von Software

2.3.1 Nicht internationalisierte Software

Bis in die siebziger Jahre hinein wurden Ausgabestrings noch explizit in der Quelltextdatei hinterlegt – sie wurden hartkodiert. In nicht internationalisier-

ter Software finden Datenspeicherung und Datenverarbeitung also in einer und derselben Datei statt wie in Listing 8.1 illustriert:

```
write("Hallo, Welt!");
```

Listing 8.1: Beispiel für Hartkodierung in Pseudocode

In diesem Beispiel ist `write()` die Ausgabefunktion und deren Argument `Hallo, Welt!` der Ausgabestring. Hartkodierung war in der Entwicklungsphase gewiss komfortabel, machte die Wartung und das Projektmanagement im Zuge der Globalisierung aber zunehmend unbeherrschbar und teuer, weil nach jeder Modifikation das Programm neu lokalisiert werden musste.

2.3.2 Internationalisierte Software

Aus solchen Erfahrungen, die keineswegs nur, aber eben auch die Lokalisierung betrafen, begann man, Software modellhaft in Schichten zu organisieren (Internationalisierungstyp Lokalisierbarkeit, siehe Abschnitt 2.1.1). In den nun folgenden Überlegungen soll von einer dreischichtigen Architektur ausgegangen werden mit …

1. einer Präsentationsschicht, umfassend jene Dateien, die für die Darstellung am Ausgabegerät zuständig sind (Typographie- und Layoutinformationen);
2. einer Geschäftslogik-Schicht, umfassend jene Dateien, die für die logische Verarbeitung zuständig sind (Quelltextdateien) und
3. einer Datenschicht, umfassend die von der Software benötigten persistenten Daten (Ressourcen).

Zu den Ressourcen und damit zur Datenschicht gehören auch die Ausgabestrings, mithin das, woran hauptsächlich Translatoren arbeiten. Ressourcen können Datenbanken oder Dateien sein. Im letzteren Fall spricht man von Resource-Dateien oder, wenn es um regionsabhängige Dateien geht, auch von *Lokalisierungsdateien*. Da, wie in Abschnitt 4.1 noch auszuführen sein wird, in der Softwarelokalisierung unterschiedliche Gewerke und Denktraditionen aufeinandertreffen, darf man sich hier allerdings auf terminologische Abenteuer gefasst machen. Bei Trados Studio heißen die Resource-Dateien *Projektdateien*, bei memoQ *zu übersetzende Dokumente*. (Das Wort „Ressource“ ist bei diesen Werkzeugen Dateien und Technologien vorbehalten, die den Translator bei seiner Arbeit unterstützen: TMs, Termbanken, MÜ, Autokorrektur-Tools etc.). Auch in der Entwicklung haben die Projekte jeweils eigene Hausterminologien. Bei GNU gettext

heißen die Resource-Dateien *message catalogs*, bei Apple werden sie je nach Format *string catalogs* oder *strings dictionaries* genannt, bei KDE *dictionary files*, bei Qt *translation files* oder *translation sources*, bei Mozilla Fluent *translation lists*.

2.3.2.1 Datenspeicherung (Resource-Datei) Die Ausgabestrings verbleiben bei internationalisierter Software also nicht in der Quelltextdatei, sondern werden in Resource-Dateien ausgelagert. Dies geschieht meist getrennt nach Kultur, im Falle eines Dateisystems sinnvollerweise eine Kultur pro Ordner oder Datei, seltener mehrere Kulturen in einer Datei. Die locale-neutrale Identifikation eines Ausgabestrings erfolgt hier über einen Bezeichner, der Schlüssel genannt wird.[2] Der unter diesem Schlüssel in der Resource-Datei gespeicherte String heißt Wert. Entsprechend nennt man den Datensatz einer so strukturierten Resource-Datei Key-Value-Paar (Listings 8.2 und 8.3) und die Datei Key-Value-Datei. Als Resource-Dateien kommen flache Formate wie Java Properties, Objective-C Strings oder gettext Portable Object, aber auch hierarchische Formate wie XML, JSON oder YAML zum Einsatz, die nach der translatorischen Bearbeitung zum Teil noch in ein Binärformat übersetzt oder in eine Programmbibliothek eingebunden werden.

```
string_1 = Hallo, Welt!
```

Listing 8.2: Key-Value-Paar in einer deutschen Ressource

```
string_1 = Hello, world!
```

Listing 8.3: Key-Value-Paar in einer englischen Ressource

In diesen Beispielen ist `string_1` der Schlüssel, das Gleichheitszeichen der Zuweisungsoperator und `Hallo, Welt!` resp. `Hello, world!` der Wert.

2.3.2.2 Datenaufruf (Quelltextdatei) In der Quelltextdatei eines internationalisierten Programms steht anstelle des Ausgabestrings eine Lookup-Funktion, die diesen String in der Resource-Datei des aktuell eingestellten Locale zunächst nachschlagen muss. Die Identifikation des Strings erfolgt, wie oben gesagt, über einen Schlüssel, der der Funktion mitgegeben werden muss (Listing 8.4). Der Rückgabewert der Lookup-Funktion ist der aufgelöste Ausgabestring.

```
write(lookup("string_1"))
```

Listing 8.4: Aufruf in einer internationalisierten Quelle (Pseudocode)

[2] Zum Locale-Begriff siehe Abschnitt 3.1.

In diesem Aufruf ist `write()` die Ausgabefunktion, deren Argument `lookup()` die Lookup-Funktion und wiederum deren Argument `string_1` der Schlüssel, unter dem der Ausgabestring in der Resource-Datei gespeichert ist.

3 Lokalisierung (L10N)

Der Terminus Lokalisierung, abgekürzt auch L10N für *localization*, bezeichnet im GILT-Paradigma die Anpassung eines Produkts an einen regionalen Markt. Softwarelokalisierung ist entsprechend die Anpassung der Benutzerschnittstelle (englisch user interface, UI) eines Computerprogramms.

3.1 Locale

Das Wort *localization* leitet sich von englisch *locale* ab, deutsch auch *Gebietsschema*. Der Terminus Locale bezeichnet das Einstellungsprofil eines Absatzmarkts. In einem solchen Einstellungsprofil wird marktabhängiger Content zusammengefasst. Für die Identifikation solcher Locales greifen die einzelnen Entwicklungsumgebungen auf teilweise generische, teilweise proprietäre Nomenklaturen zurück, z. B. `de_AT` für Deutsch / Region Österreich oder `de-AT` für Deutsch / Subsprache österreichisches Deutsch.

3.2 Zu lokalisierende Daten

3.2.1 Manuelle Ersetzungen

Manuell zu lokalisierende verbale Informationen sind beim Produkt Software vor allem Ausgabestrings und Zugriffstasten. Diese Eingriffe übernimmt im GILT-Paradigma der Translator (mehr hierzu Abschnitt 4). Manuell zu lokalisierende nonverbale Informationen können zunächst Schriftauszeichnungen sein. Das mag selbstverständlich klingen, ist es aber nicht, denn nach dem Prinzip der Trennung von Content und Format (Goldfarb & Rubinsky 1990: 567) gehören Stile in die Präsentationsschicht, sind also außerhalb der Reichweite des Translators definiert. Eine solche Trennung ist zweifellos gut gedacht, doch etwas zu kurz, beruht sie doch auf der irrigen Annahme, alle in einer Kultur üblichen typographischen Gestaltungsmittel – Laufweite, Kursivdruck, Unterstreichung, Versalierung usw. – seien in einer anderen Kultur oder gar in einem anderen Schriftsystem in derselben Weise gebräuchlich oder auch nur vorhanden. Aus diesem Grund werden Schriftauszeichnungen zuweilen mit in die Ausgabestrings eingebettet und damit für die translatorische Bearbeitung zugänglich gemacht.

Bei unglücklich internationalisierter Software müssen u. U. auch Symbole, Farben und Schallereignisse manuell lokalisiert werden, nämlich immer dann, wenn diese nicht universal sind. Diese Aufgabe obliegt im GILT-Paradigma dem Lokalisierungsingenieur; fehlt ein solcher, so erfolgt deren Erledigung kollaborativ, je nach Projekt unter Einbeziehung des Translators, des Entwicklers und des Auftraggebers. Einen Eindruck über die Bandbreite kulturabhängiger Größen vermittelt Heimgärtner (2017: 17-19).

Dem Werkzeugcharakter von Software ist geschuldet, dass Benutzerschnittstellen gesetzlich stärker geregelt sind als Druckmedien, besonders etwa dann, wenn Fragen der Arbeits-, Verkehrs-, Patienten- oder Datensicherheit berührt sind. Die Anpassung von Software an unterschiedliche rechtliche und korporative Erfordernisse geschieht i. d. R. ebenfalls kollaborativ.

3.2.2 Automatische Ersetzungen

Es fällt aber auch Content an, der automatisch lokalisiert werden kann. Die hierfür benötigten Daten findet das Programm in einer Bibliothek. Eine solche Bibliothek mag die Lokalisierung u. a. der folgenden Größen übernehmen:

- wenn eine Single-Byte-Kodierung verwendet wird: Wahl der Codepage
- Sprachregeln
  - Schreibrichtung
  - Sortier- und Silbentrennungsregeln
  - Numerusregeln (mehr hierzu in Abschnitt 3.3.4)
  - Wahl der Wörterbücher für Rechtschreibprüfung, Autovervollständigung und Autokorrektur
- Layout
  - Tastaturbelegung
  - Bedien-Gesten (Bildschirmgesten, Raumgesten)
  - bei logographischen Schriftarten: Eingabeschema
  - im Desktop-Publishing (DTP): Papierformate
- Formatierungen
  - Grund- und Ordnungszahlen

  - Aufzählungszeichen
  - Telefonnummern
  - Zeitangaben
  - Datumsangaben
  - Preisangaben
  - Adressangaben
- Kalenderfunktionen
  - Default-Zeitzone
  - Sommerzeitregelung
  - Festlegung des ersten Tags der Woche
  - Wochenend- und Feiertagsregeln
- Maßsysteme
  - Währungseinheiten
  - Temperatureinheiten
  - Geschwindigkeitseinheiten
  - Gewichtseinheiten
  - Längeneinheiten
  - Raumeinheiten

3.2.3 Hinzufügung und Unterdrückung von Content

Der Vollständigkeit halber ist darauf hinzuweisen, dass Lokalisierung nicht zwangsläufig die Ersetzung von Content sein muss; je nach regionalen Vorschriften müssen besonders in stark regulierten Anwendungen wie Luftfahrt und Medizintechnik auch Pflichtinformationen bzw. -features hinzutreten oder verbotener bzw. unüblicher Content unterdrückt bzw. aus dem Default genommen werden.

3.3 Organisation der Oberflächenstrings

3.3.1 Normalisierung

Komplexe Strings (Sätze oder Syntagmen) werden mitunter in einfachere (Wörter oder Wortverbindungen) zerlegt, wodurch zunächst Redundanzen entstehen, die in einem anschließenden, *Konsolidierung* genannten Schritt durch Zusammenlegen von Datensätzen eliminiert werden können. Die auf diese Weise entstehenden String-Fragmente können zur Laufzeit dann nach Bedarf zu Syntagmen bzw. Sätzen verkettet (*konkateniert*) werden. Das hilft Speicherplatz sparen, die lexikalische Konsistenz verbessern, kombinatorische Explosion vermeiden und das Volumen für den anschließenden Arbeitsschritt (Translation) senken. Eine mögliche Anwendung für die Normalisierung von Stringbeständen wäre eine Navigations-App, die eine praktisch unbegrenzte und vor allem nicht vorhersagbare Auswahl möglicher Anweisungssequenzen generieren kann, wohingegen die Anzahl möglicher Anweisungsbausteine – „links/rechts halten/abbiegen“ etc. – durchaus begrenzt ist.

Die Konkatenation von Teilstrings ist eine logische Operation, erfolgt also in der Geschäftslogik eines Programms. Dies geschieht i. d. R. nach den Bedürfnissen der Ausgangssprache der Programmoberfläche, meistens Englisch. So sinnvoll die Normalisierung für die Datenhaltung sein mag – für den Lokalisierer bedeutet sie einen Verlust an Kontrolle über die Syntax und Morphologie in der Zielsprache. In den Abschnitten 3.3.2, 3.3.3 und 3.3.4 sollen Techniken vorgestellt werden, die dem Lokalisierer helfen können, einen Teil der Kontrolle zurückzugewinnen.

3.3.2 Interpolation von Teilstrings

Damit der Lokalisierer einen besseren Zugriff auf die Wortfolge in der Zielsprache bekommt, verzichten Entwickler zuweilen auf die Konkatenation von Strings und greifen zu einer alternativen Technik – der Interpolation von Strings. Bei dieser Technik werden Strings in andere Strings „eingeschoben“. Die Stelle, an der dies passieren soll, wird im aufnehmenden String durch eine als Platzhalter dienende Variable reserviert. Für die Notation solcher Platzhalter existieren in unterschiedlichen Sprachen unterschiedliche Konventionen. Oft ist es eine Indexzahl (z. B. `{0}`), ein sprechender Variablenname (*self-documenting identifier*, z. B. `%{username}`) oder ein sprechender Buchstabe für den jeweiligen Datentyp (*format specifier*, z. B. `%s` für *String*). Gemeinsam ist solchen Platzhalterausdrücken, dass sie durch Delimiter – geschweifte Klammern, Prozentzeichen, @-Zeichen, Ausrufezeichen oder eine Kombination daraus – ausgezeichnet sind.

3.3.3 Translation Scripting

Mit der Interpolation von Teilstrings bekommt der Translator Zugriff auf die Wortfolge, nicht jedoch auf die Genus-, Kasus- und Numeruskongruenz im Satz. Nach der Jahrtausendwende entstanden deswegen Überlegungen, wie man die Datenschicht befähigen kann, selbst Teile der logischen Verarbeitung zu übernehmen, namentlich dort, wo logische Beziehungen sprachspezifisch sind. Ein Beispiel ist (Illich 2003), dort noch unter der Bezeichnung *programmable UI translations*. Im Jahr 2008 schließlich lag für das KDE Framework 5 eine erste praxistaugliche, auf JavaScript basierende Architektur zum Skripten von Übersetzungen vor.[3] Ausführlich wird dieses Konzept in (Behrens 2016: 137–152) vorgestellt, dort unter der auf das KDE Framework 5 zurückgehenden Bezeichnung Translation Scripting. Beispiele für jüngere Scripting-Lösungen sind Mozilla Fluent und ICU MessageFormat.

3.3.4 Numerusregeln

Häufig sind in Platzhaltern Zahlen gespeichert. Wo eine erst zur Laufzeit entstehende Zahl andere Konstitutenten im Satz regiert, ergeben sich Übersetzbarkeits-Probleme, da unterschiedliche Sprachen unterschiedlichen Rektionsregeln folgen. Im Gegensatz zur Technologie des Translation-Scripting, die sich in der Breitenanwendung bis heute nicht so recht durchsetzen konnte, sind Mechanismen für die Abbildung des Rektionsverhaltens von Zahlen, meist unter Schlagworten wie *pluralization*, *plural rules* oder *plural handling* diskutiert, in den gängigen Bibliotheken schon seit längerem Standard.[4] Systematisch kategorisiert findet man die in den jeweiligen Sprachen anzutreffenden Rektionsmuster im CLDR (Common Locale Data Repository, siehe Unicode-Konsortium 2023). Hauptwerk des CLDR sind in LDML (Locale Data Markup Language, einem XML-Format) geschriebene Dateien mit Locale-Informationen, darunter eben auch solchen zum Rektionsverhalten von Zahlen.

4 Translation (T9N)

Der Terminus Translation, abgekürzt auch T9N für *translation*, bezeichnet im GILT-Paradigma den sprachmittlerischen Teil des Arbeitsschritts Lokalisierung.

[3] Private Kommunikation mit Chusslove Illich ab dem 09.12.2013.

[4] Das im Deutschen gebräuchliche Wort *Pluralregeln* trifft das Ansinnen nur schlecht und soll hier gemieden werden, denn es ist mit der Dichotomie Singular ↔ Plural eben nicht getan, wo eine Sprache Numerus-Kategorien aufweist, die in die genannte Dichotomie nicht so recht hineinpassen wollen (Paukal, Dual, Trial).

4.1 Arbeitsumgebungen

4.1.1 TMS (Translation-Memory-Systeme)

Die der Textverarbeitungs-Tradition verpflichteten TMS verarbeiten ein Dokument i. d. R. bei jeder Iteration *in toto*. Namensgebende Funktion eines TMS ist dessen Fähigkeit, sich Strings zu „merken". Damit das TMS dies kann, muss es wertbasiert arbeiten. (Zum Begriff des Werts siehe die Ausführungen in Abschnitt 2.3.2.1.)

4.1.2 Non-TMS

Die aus der Welt des Software-Engineerings hervorgegangenen proprietären Ressourcen-Editoren setzen tendenziell auf eine inkrementelle Pflege des Datenbestands. Sie unterscheiden Oberflächenstrings nicht (wie TMS dies tun) nach bekannten und unbekannten, sondern nach erledigten und unerledigten. Damit können diese Arbeitsumgebungen in gewisser Weise als Gegenentwurf zum TMS verstanden werden, denn sie legen den Fokus eher aufs Vergessen denn aufs Merken. Hierfür müssen sie ein entsprechendes Veränderungsmanagement mitbringen. Ein solches Veränderungsmanagement wird schon aus Gründen der Ressourcenschonung nicht über Werte umgesetzt werden, sondern über Metadaten.

Eine vielleicht nicht mehr ganz zeitgemäße, aber immer noch praktizierte Vorgehensweise besteht darin, neue Strings zunächst in eine Template-Datei zu schreiben oder mit einem einfachen „Diff-Tool" dorthin zu extrahieren. Eine solche Template-Datei kann nach der translatorischen Bearbeitung dann zurück in die Produktiv-Ressource gemergt werden.

4.1.3 Mischkonzepte

Jeder dieser zwei Wege hat seine Berechtigung. Vorteil des Übersetzungsspeichers ist, dass er projektübergreifend eingesetzt werden kann (eine Anforderung, auf die Softwareentwickler nicht unbedingt von selbst kommen) und überall dort, wo im Vorfeld der translatorischen Bearbeitung kein Veränderungsmanagement stattgefunden hat. Inkrementell – i. d. R. ohne TM – arbeitende proprietäre Umgebungen bewähren sich dagegen besser im Umfeld einer kontinuierlichen Entwicklung (eine Anforderung, an die zur Entstehung klassischer TMS noch nicht zu denken war). Unter Umständen bietet sich an, das eine zu tun, ohne das andere zu lassen, also inkrementell und mit TM zu arbeiten. Der Übersetzungseditor Poedit (mit dem Arbeitsformat POT ein klassisches Beispiel für den Weg über

eine Template-Datei) verfügt mittlerweile auch über ein TM. Eine andere Strategie verkörpert das Lokalisierungs-Tool Rigi (Abbildung 1). Rigi selbst arbeitet ID-basiert (also nicht wertbasiert) und gehört damit zur Klasse der Non-TMS. Aber es bietet Plugins für Trados und memoQ und erlaubt so eine Koexistenz beider Konzepte.

Dass die genannten zwei Konzepte in der Datenorganisation unterschiedliche Wege gehen, soll nicht über bestehende Gemeinsamkeiten in der Datenverarbeitung hinwegtäuschen. Immer wird im Translationsschritt eine ausgangssprachliche Resource-Datei sprachmittlerisch bearbeitet. Dies geschieht praktischerweise in einem vorgeparsten Dokument, ganz gleich, ob mit oder ohne TM gearbeitet wird.

4.2 Parsen der Datei

4.2.1 Dateistruktur definieren

Wie in Abschnitt 4.1 festgestellt, arbeitet ein TMS rein wertbasiert, also so, wie der mit ihm arbeitende Translator es auch tut. Schlüssel und andere Strukturdaten sind für die Datenorganisation in einem TMS überflüssig; für das menschliche Auge sind sie Rauschen.[5] Entwickelte TMS haben Werkzeuge, mit denen sich Parser – im vorliegenden Kontext wird meist von Filtern gesprochen – individuell herstellen und konfigurieren lassen. Der Filter sorgt dafür, dass ausschließlich Translatables ausgewertet werden und im Übersetzungseditor erscheinen, also Strukturdaten ausgeblendet werden und bei der Berechnung der Match-Raten unberücksichtigt bleiben.

Die schier unbegrenzt anmutende Vielfalt möglicher Resource-Formate mag auf den ersten Blick einschüchtern; tatsächlich aber ist das Spektrum der in solchen Dateien verwendeten Sprachen gar nicht so groß, und entsprechend gering die Anzahl der für die Formulierung von Filterregeln infrage kommenden Methoden:

1. Feldtrennzeichen-strukturierte Dateien (CSV, TSV etc.) werden je nach Tool graphisch oder mit regulären Ausdrücken definiert.

2. Flache Key-Value-Dateien (Properties, Strings etc.) und Quelltextdateien (PHP, JS etc.) werden sinnvollerweise mit regulären Ausdrücken definiert.

[5] Der für Matches > 100 % verwendete ID-basierte Kontext in memoQ stellt eine über das Konzept von TMS hinausgehende Beziehung dar und ändert nichts an der Wertbasiertheit von TMS.

3. XML-Dateien werden sinnvollerweise mit XPath definiert.

4. JSON-Dateien werden sinnvollerweise mit JSONPath definiert.

Es existieren auch Formate, die sich in den graphischen Dialogen von Filterassistenten nicht gut beschreiben lassen. Zu diesen Formaten gehören Plain-Text-Formate wie PO oder RTF, mittlerweile selten auch Binärformate wie DLL oder EXE. Bei ihnen wird man auf werkseitig ausgelieferte Filter zurückgreifen oder auf die Möglichkeit, Filter als Plugin einzubinden.

4.2.2 Eingebettete Strukturdaten definieren

In den Ausgabestring eingebettete Non-Translatables wie Platzhalter (s. o. Abschnitt 3.3.2) oder Auszeichnungs-Tags können in entwickelten CAT-Tools maskiert werden. Hierfür wird an den Filter für die Dateistruktur ein weiterer für eingebettete Elemente angehängt. Eine solche Maskierung erhöht den Lesekomfort bei der Bearbeitung im CAT-Tool und mindert das Risiko der Verletzung von Strukturdaten, die gerade bei XML recht ausladend sein können, ist aber natürlich nur sinnvoll, wenn solche Strukturdaten tatsächlich nicht bearbeitet werden müssen.[6]

4.3 Graphischen Kontext vorrendern

Die persistente Datenrepräsentation in den Resource-Dateien von Programmen liegt i. d. R. dekontextualisiert – weil ohne syntaktischen und graphischen Kontext – vor (Abschnitt 3.3). Dies macht Softwarelokalisierung inhaltlich wie sprachlich anfällig und ergonomisch aufwändig.

Früher, als Anwendungen nicht selten erst kompiliert und dann lokalisiert wurden, konnte man mit *visuellen Lokalisierungstools* (die bekanntesten unter ihnen RWS Passolo und Alchemy CATALYST) versuchen, die in der Ressourcensektion der ausgelieferten Binary immer noch im Klartext vorliegenden Strings (Accelerators, Dialoge, Menüs, Stringtables) und Layoutinformationen zu extrahieren und aus ihnen eine WYSIWYG-Vorschau zu erzeugen.[7] Das erlaubte es dem Lokalisierer, im graphischen Kontext zu arbeiten.

[6] URLs werden selten zu übersetzen sein; sie sind es aber dort, wo eine zielsprachliche Version einer Webseite mit eigener URL existiert. Zur Bearbeitungsbedürftigkeit von Schriftauszeichnungen siehe Abschnitt 3.2.1.

[7] WYSIWYG ist die Abkürzung für *What You See Is What You Get* und bezeichnet vor allem in der Textverarbeitung eine Technologie zum Vorrendern des graphischen Endbildes eines Texts.

Aus heutiger Sicht mag verwundern, wie die Praxis, ein Produkt erst zu finalisieren und dann doch noch zu bearbeiten, sich derart bewähren konnte, dass sie eine eigene Werkzeugklasse hervorgebracht hat. Historisch lässt sich diese Tatsache aber erklären: Zum einen verstand man den Lokalisierungsschritt nicht von Beginn an als Teil des Entwicklungsprozesses (siehe hier beispielhaft Esselink 2003), weswegen die Lokalisierung als Anschlussgewerk lange Zeit selbst zusehen musste, wie sie Zugang zu den Ressourcen bekam: Es blieb oft nur der über das fertige Produkt. Zum anderen kamen die klassischen CAT-Tools noch zur Jahrtausendwende für Lokalisierungsaufgaben nicht in Frage, weil sie kaum mehr als Office-Dateien verarbeiten konnten. Das ließ den Ruf nach einem dedizierten „Loctool" laut werden. Offensichtlich war bei alledem ein Vorteil, der in den fertigen Binarys selbst lag: Diese hatten häufig alle zum Rekontextualisieren der fragmentierten Stringbestände nötigen Daten beisammen.

Heute, wo auch klassische CAT-Tools für Ressourcenformate der Softwareentwicklung bestens gerüstet sind, wo immer seltener statische Binarys zum Einsatz kommen, wo mehr interpretiert und mehr mit verteilten Anwendungen gearbeitet wird, müssen die überkommenen visuellen Lokalisierungstools als überholt gelten. Für Webanwendungen gibt es heute die Technologie des In-Context-Editing (ICE). Beim ICE arbeitet der Translator auf einem separaten Lokalisierungsserver, also nicht auf dem Produktivsystem, sehr wohl aber live in der laufenden Anwendung, und immer mit graphischem Kontext. Beispiele für ICE-Editoren sind Lokalise, Phrase oder Smartling. Diese Editoren integrieren zwei Systeme: eine allein dem Preview dienende Kopie der laufenden Anwendung (*Staging-Kopie*) und ein TM-System. Zwischen beiden vermittelt der ICE-Layer.

Mit ICE lassen sich freilich nur Anwendungen lokalisieren, die auf einem Server laufen können, also hauptsächlich Webanwendungen. Desktop-Anwendungen muss man deswegen aber noch nicht abschreiben. Einen interessanten Mittelweg geht nämlich XTM mit dem Visualisierungstool XTM Rigi. Im Unterschied zu den früheren visuellen Lokalisierungstools ist das in der Lage, aus Laufzeitdaten eine graphische Vorschau zu rendern, und im Unterschied zu den modernen ICE-Werkzeugen kann es WYSIWYG-Ansichten auch für Desktopanwendungen erzeugen. Rigi lässt zunächst die Anwendung in einem nativen Umfeld laufen und erzeugt dabei eine statische Zwischenschicht (*Capturing*). Beim Capturing entstehen die für die Previews erforderlichen Screenshots und die String-Ressourcen. Letztere sind XML-Dateien mit der Dateiendung RIF (*Rigi Intermediate Format*). Die RIF-Ressourcen können (anders als beim ICE) lokal mit dem eigenen CAT-Tool bearbeitet werden, sinnvollerweise freilich bevorzugt mit Trados Studio oder memoQ, weil Rigi neben dem hauseigenen Webeditor auch

Plugins für diese Tools anbietet. Im Falle eines Einsatzes von Trados oder memoQ findet der graphische Preview nicht im CAT-Tool selbst statt, sondern in einem separaten Client, dem Rigi Viewer. Der Rigi Viewer lädt die benötigten Screenshots vom Rigi-Server, dies entweder einmalig als Gesamtprojekt oder dynamisch in Echtzeit (im letzteren Fall wie beim ICE über einen *Staging-Server*), und gibt im Übersetzungsprozess den für die jeweilige Translation Unit benötigten Screenshot aus. Ein Beispiel für so eine Ausgabe zeigt Abbildung 1.

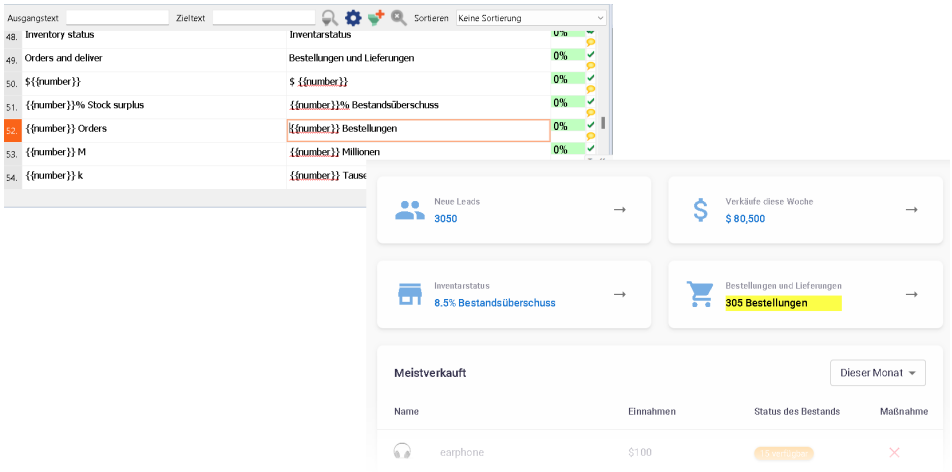

Abbildung 1: Oberflächenstrings im Editor von memoQ; vorne: graphischer Preview in Rigi Viewer (Ausschnitt)

Beide Lösungen – ICE und der statische Zwischenschritt von Rigi – sind im Einsatz natürlich etwas aufwändiger, weil der Anbieter zunächst die Anwendung in einem nativen Umfeld zum Laufen bringen und sich in sie einklinken muss. Sie sind in diesem Sinne nicht nur ein Softwareprodukt, sondern auch eine Dienstleistung.

Literatur

Behrens, Alexander. 2016. *Lokalisierbarkeit von user-interface-strings. Übersetzerische Aspekte der Internationalisierung und Lokalisierung von Software, untersucht anhand der Übersetzungsrichtungen englisch-deutsch und englisch-russisch* (Leipziger Studien zur Angewandten Linguistik und Translatologie 16). Frankfurt: Lang.

Beste, Kai. 2006. *Softwarelokalisierung und Übersetzung* (Heidelberger Studien zur Übersetzungswissenschaft 8). Trier: WVT, Wissenschaftlicher Verlag.

Esselink, Bert. 2003. *The evolution of localization*. https://multilingual.com/downloads/screenSupp57.pdf.

Fry, Deborah & Arle R. Lommel. 2003. *LISA. The globalization industry primer*.

Goldfarb, Charles F. & Yuri Rubinsky. 1990. *The SGML handbook*. Oxford: Oxford University Press.

Göpferich, Susanne. 2002. *Textproduktion im Zeitalter der Globalisierung: Entwicklung einer Didaktik des Wissenstransfers*. 3. Aufl (Studien zur Translation 15). Tübingen: Stauffenburg-Verlag.

Heimgärtner, Rüdiger. 2017. *Interkulturelles user interface design: Von der Idee bis zum erfolgreichen Produkt*. Berlin: Springer Vieweg.

Illich, Chusslove. 2003. *Programmable UI translations*. http://nedohodnik.net/misc/cotras-intro.html.

O'Brien, Sharon. 2019. Controlled language and writing for an international audience. In Bruce Maylath & Kirk St.Amant (Hrsg.), *Translation and localization: A guide for technical and professional communicators*, 65–88. New York: Routledge. DOI: 10.4324/9780429453670-4.

Schmitz, Klaus-Dirk & Kirsten Wahle (Hrsg.). 2000. *Softwarelokalisierung*. Tübingen: Stauffenburg-Verlag.

Unicode-Konsortium. 2023. *Language plural rules*. https://www.unicode.org/cldr/charts/43/supplemental/language_plural_rules.html.