Aalto University
School of Science and Technology
Faculty of Electronics, Communications and Automation
Degree Programme of Electronics and Electrical Engineering

Marco Grönberg

User Interface Localization with Design-Time Support by Dynamically Extending User Interface Components

Master's Thesis

Espoo, May 10, 2010

Supervisor: **Professor Lauri Malmi**

Instructor: **M.Sc. Kim Nyberg, Tekla Corporation**

Aalto University
School of Science and Technology
Faculty of Electronics, Communications and Automation
Degree Programme of Electronics and Electrical Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Marco Grönberg		
Title of thesis:	User Interface Localization with Design-Time Support by Dynamically Extending User Interface Components		
Date:	May 10, 2010	**Pages:** 9 + 66	
Professorship:	Software Technology	**Code:** T-106	
Supervisor:	Professor Lauri Malmi		
Instructor:	M.Sc. Kim Nyberg, Tekla Corporation		

Some companies are veterans in internationalized software development and they have developed their own localization technologies, which they want to continue to use with new user interface technologies. However, the new technology may not be compatible with the localization technology of the company and so the company needs to extend the user interface library to support their existing technology. The solution has to be flexible in order to support different component types as they cannot be supported with a single strategy.

In this thesis a new localization support is implemented for Windows Forms user interfaces. The implementation is done by dynamically extending user interface components with new properties at design-time. The properties are used to set translation identifiers for the components. The components are localized in the designer without need to run the project and the solution also allows to add other new functionality such as pseudo-localization.

The presented solution requires that the objects are identifiable either with instance or with property. Most component types fulfill this requirement. However, some components cannot be fully supported as they have child objects that are not identifiable. They require some additional work if they must be supported and the presented solution is not sufficient per se. Still, the used extension technique has potential as its use is not only limited to localization.

Keywords:	software, internationalization, localization, Windows Forms, development environment
Language:	English

Aalto-yliopisto
Teknillinen korkeakoulu
Elektroniikan, tietoliikenteen ja automaation tiedekunta
Elektroniikan ja sähkötekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Marco Grönberg		
Työn nimi:	Käyttöliittymän kotoistus käyttöliittymäkomponentteja suunnitteluvaiheessa dynaamisesti laajentamalla		
Päiväys:	10. toukokuuta 2010	**Sivumäärä:**	9 + 66
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Työn valvoja:	Professori Lauri Malmi		
Työn ohjaaja:	DI Kim Nyberg, Tekla Oyj		

Kansainvälisille markkinoille pitkään ohjelmistoja tehneet yritykset ovat usein kehittäneet omia ratkaisujaan ohjelmistojen kotoistukseen. Uudet käyttöliittymäteknologiat eivät välttämättä ole yhteensopivia entisten ratkaisujen kanssa, joten heidän täytyy laajentaa uutta teknologiaa tukemaan vanhoja ratkaisuja. Laajennusratkaisun pitää olla joustava, koska kaikkia erilaisia käyttöliittymäkomponentteja ei välttämättä kyetä tukemaan yhdellä ainoalla menetelmällä.

Tässä diplomityössä esitellään menetelmä Windows Forms -käyttöliittymäkomponenttien ominaisuuksien laajentamiseen käyttöliittymän suunnitteluvaiheessa. Lisättyjen ominaisuuksien avulla komponenteille asetetaan käännösten tunnisteet, joiden avulla käännöstietokannasta pystytään hakemaan oikeat käännökset. Ratkaisu mahdollistaa myös muunlaisen uuden toiminnallisuuden lisäämisen.

Esitetty ratkaisu edellyttää, että objektit pystytään tunnistamaan joko instanssilla tai nimellä. Suurin osa käyttöliittymäkomponenteista täyttää tämän vaatimuksen. Osalla komponenteista on kuitenkin lapsiobjekteja, joita ei pystytä tunnistamaan luotettavasti, ja näille tuki pystytään toteuttamaan vain osittain. Esitetty menetelmä komponenttien laajentamiseksi on varsin lupaava eikä sen käyttö rajoitu pelkästään kotoistukseen, sillä sitä on helppo soveltaa myös muuhun käyttöön.

Avainsanat:	ohjelmisto, kansainvälistäminen, kotoistus, Windows Forms, ohjelmointiympäristö
Kieli:	englanti

Acknowledgements

I want to thank professor Lauri Malmi for supervising this thesis. His feedback and advices have been really invaluable. I also want to thank Kim Nyberg for being my instructor and for allowing me to do this thesis at Tekla.

Thanks to Kari Sydänmaanlakka for giving me LaTeX templates for master's thesis. They saved me a lot of trouble.

I thank my parents Riitta and Rainer as they have always motivated me to study and to work hard.

Last but not least, I want to thank my wife Regina for giving me support and encouragement while writing this thesis.

This master's thesis is dedicated to my lovely daughters Ellen and Adela.

Espoo, May 10, 2010

Marco Grönberg

Contents

Contents iv

List of Tables vii

List of Figures viii

List of Listings ix

1 Introduction 1

2 Terminology 3

2.1 Localization 3

2.1.1 Linguistic Issues 4

2.1.2 Physical Issues 5

2.1.3 Business and Cultural Issues 6

2.1.4 Technical Issues 7

2.2 Internationalization 8

2.3 Globalization 9

3 The Globalization Process 11

3.1 Project Planning and Preparation 11

3.1.1 Market Evaluation and Globalization Requirements 11

3.1.2 Selection of the Localization Vendor 13

3.2 Product Design and Development 14

3.3 Product Localizability Testing 14
3.4 Product Localization and Testing 15
3.5 Preparing for the Next Project 16

4 Localization Bindings **17**
4.1 Identifying by Integer 17
4.2 Identifying by User Interface Texts 20
4.3 Identifying by Component Name 20
4.4 Identifying by Arbitrary Name 21
4.5 Identifying by Component Instance 22
4.6 Summary 23

5 Existing Technologies **25**
5.1 GNU gettext Library 25
5.2 Localization with Java 27
5.3 Windows Forms Localization 29
5.4 Tekla Technology 33
5.5 Summary 35

6 Improving Localization **37**
6.1 Culture Repositories 37
6.2 Integration of the Cultural Aspects 38
6.3 Referring Resources from the Source Code 39
6.4 Automatically Identifying Localizable Content 40

7 Problem Domain **42**
7.1 Requirements 42
7.1.1 Compatibility with Tekla Technology 43
7.1.2 Usability Requirements 43
7.2 Localizable Component Types 44
7.2.1 Independent Components 44

7.2.2 Components with Child Components 45
7.2.3 Components with Child Objects 45
7.2.4 Component Extenders 46
7.3 Possible Solutions 47
7.3.1 Custom Resource Manager 47
7.3.2 Extensions with Component Extenders 48
7.3.3 Extensions with Metadata Substitution 49
7.4 Summary 50

8 Localizer Implementation 53
8.1 Component Localization 54
8.2 Localizer Component 55
8.2.1 Identifier Storage 55
8.2.2 Localization at Runtime 56
8.3 Type Extensions 56
8.4 System Operation Examined Step by Step 57
8.5 Strongly-Typed Resources 58
8.6 Remaining Problems 58
8.6.1 Unsupported Component Types 59
8.6.2 Compatibility in the Future 60
8.7 Summary 60

9 Conclusions 62

Bibliography 63

List of Tables

7.1 Evaluation of the alternative solutions 51
7.2 Symbol descriptions for the evaluations of the solutions 51

List of Figures

2.1 German labels require more space 4
2.2 Japanese keyboard 5
2.3 Arabic layout is from right to left 8
2.4 Structure of the globalization 10

3.1 Globalization process has a cyclic nature 12
3.2 Localizability can be tested with pseudo-localization 14

8.1 Overview of the Localizer implementation 53

Listings

4.1 Obscurity can be a major problem with integer identifiers 18
4.2 Plain integer identifiers replaced with an enumerated type 19
4.3 Translations defined with the enumerated type 19
4.4 Instance binding is different approach for localization 22
5.1 Snippet of the GCC compiler's Spanish portable object file 26
5.2 Example of the Java's ListResourceBundle implementation 27
5.3 The property definition file has very straightforward syntax 28
5.4 The .resx files used by Windows Forms are human-editable 29
5.5 Strongly-typed resource reference is verifiable by the compiler . . 30
5.6 Generated strongly-typed resource class 31
5.7 Example of the translation definitions in the AIL file 34
8.1 Serialized values of the Localizer component 54

Chapter 1

Introduction

Modern global markets cause additional requirements to the software development as the software must be internationalized, that is, prepared for localization. The localization means that the software is translated and culturally adapted for a specific region. The localization is nowadays considered often self-evident for all bigger market segments, because a localized software product has a clear advantage against non-localized rival products.

Tekla Corporation has done international software development a couple of decades and Tekla has developed its own platform-independent technology to define user interfaces and their translations. The software development has been done mainly in C/C++, but there are also small amounts of code that is written in Java, Visual Basic and some scripting languages. Tekla has used .NET framework in the product development since it was released in 2002. Initially it was used to add extension support to existing native applications and to publish programming interfaces for third-party developers. Lately Tekla's products have started to use the .NET framework more extensively and the user interfaces of the .NET applications have been made with Windows Forms.

The Windows Forms user interfaces should be localized in a similar manner than the old user interfaces and they should be able to use similar translation files as the old technology uses. However, the Windows Forms localization is completely different than the Tekla's localization technology and so it was necessary to investigate how the Tekla style localization could be used with the Win-

dows Forms. After the alternatives were compared, *the most promising alternative was to dynamically extend the existing user interface components of the Windows Forms* and this solution is described in this thesis.

Structure of the Thesis

The structure of the thesis aims to be logically ordered and every chapter prepares the reader for the next chapters. First the reader is familiarized with basic terminology and globalization process. It is vital to know the big picture of the globalization in order to understand that the solution presented in this thesis concerns only a small and very specific problem area. The subsequent chapters are more topic-specific as they concern the translation identifiers and their use in real life software. There is also a chapter about other internationalization related research and it describes some ideas and improvements for the internationalization. Finally the problem domain and possible solutions are described before presenting the chosen method to solve the problem.

Chapter 2

Terminology

The terminology is a bit vague in the language industry, because the basic terms do not have commonly accepted definitions [10]. To avoid confusion one should always define the meaning of words like globalization, internationalization and localization. The following sections will describe the terminology as it is used in this thesis. It also gives some examples about the issues that are related to the localization and internationalization.

The terms are often abbreviated with the numeronyms: all letters between the first and last letter are replaced with the number of the omitted letters. Hence the localization is abbreviated with L10n or l10n, the internationalization with i18n and the globalization with g11n. These abbreviations are not used in this thesis.

2.1 Localization

The process in which the software is adapted to another locale is called localization. The translation of the texts in the user interface and documentation is the main part of the localization. However, the localization includes much more than just the translations. There are four kind of issues that localization commonly addresses: linguistic issues, physical issues, business and cultural issues as well as technical issues [19].

Figure 2.1: German labels are require more space their English counterparts

2.1.1 Linguistic Issues

Linguistic issues concern all applications that are to be sold to individuals who do not speak the language used in the application. For a software project there can be both application and business related resources that need to be adapted to the new locale. The application will require translation for the user interface components, online help, documentation, installer and other similar resources. If the application has spoken audio feedbacks they may require dubbing. The business related resources include for example marketing materials, web pages and training materials.

Linguistic adaptation may also impact directly to the software design. For example, the space requirements of the translated text may differ significantly from the original space requirements and, in the worst case, the whole user interface may have to be redesigned to allow the localization. Of course, this would balloon the cost of the localization process and, as will be explained in Section 2.2, this kind of problems are tried to avoid with internationalization.

The space requirements are visualized in Figure 2.1. The difference is most clearly seen in the buttons that have been resized and relocated in order to accomodate the longer German labels. If the dialogs are examined even closer it is noted that the description labels do not have equal content. The German version does not have the final part of the English sentence after the last comma, because the localizer has not been able to fit it into the available space.

Figure 2.2: Japanese keyboard allows user to change between entry modes

2.1.2 Physical Issues

Software projects are almost completely free of the physical issues. This is quite natural because physical problems are related to concrete things and software is purely virtual. However, there are a few things that one should take into account if the software is embedded in the hardware or if the hardware is referred in the software and documentation.

Probably the most obvious physical issue is the problem caused by the keyboard layouts that differ between countries. Some characters do not exist in all keyboard layouts and that may affect shortcut-key combinations presuming that those are not localizable. There are also keyboards that have multiple different input methods that need to be supported if those languages are targeted. For example, in Figure 2.2 we see a Japanese keyboard containing extra keys that allow the user to toggle between generating phonetic syllables and latin characters. The phonetic syllables are treated specially so the software must be aware of them [24].

Less obvious issues are more hardware related. The line voltage and frequency varies around the world as do the electrical plugs. The voltage can be anything from the 100 volts of Japan to the 240 volts of Australia, the frequency is usually either 50 or 60 Hz and there are 13 different electrical plugs in use. If there is some wireless technology involved, it must obey the local regulations and standards as well.

Though the physical issues rarely affect the software design itself, they do affect the software localization process. Both application and documentation should

refer to correct local information (e.g. line voltage) and the graphical representations should either reflect the particular hardware in the specific markets or be designed to be so general that they can be used everywhere.

2.1.3 Business and Cultural Issues

Maybe the biggest source for technical issues in any project are the business and cultural issues unless they are taken into account from the early stages of the product development. This is because they affect all aspects of the design and most of them require low-level support if they are to be implemented properly.

For the business it is always important to be pleasant and polite towards all clients. A lot of business issues could be described also as political issues, because often politics are the main cause for the business issues. One notorious source for political issues are maps with controversial borders. In worst case the political issues may be so severe that the product will be banned by the government.

Still, not all business issues are political. Offensive symbols, graphics or language are just as bad. The software may work just fine but the customers may seek alternatives from other companies if the content offends or causes negative feelings to them. The French flag to indicate French language may seem intuitive but it may offend people in other french speaking countries like in Canada or Switzerland.

Typical culture issues are related to formatting of the data. Addresses, currencies, dates, numerals, telephone numbers and time must all be formatted according to the local conventions. An application should also display measurements with correct units and support printing to different paper sizes.

Cultural issues may also affect to the colors and graphics of the product, because as much as they should meet the local cultural norms they should also meet the local cultural expectations and be intuitive and easy to understand [18]. Symbols that are not internationally recognizable need to be adapted to the local culture. For example, a rural-style mailbox in the United States is often interpreted as a tunnel by the Europeans [23]. Another example is a check mark that means correct or OK in many countries but in Japan and Finland it means that something is incorrect. Japanese localizers may need to convert the symbol to circle,

which is their symbol for correct, and likewise the Finnish localizers would use the commercial minus symbol [3, 8].

Most of the cultural issues require substantial local knowledge and its importance should not be overlooked. Usually companies have to rely on their local partners to deliver this information unless they have externalized the whole localization process to a third party vendor.

2.1.4 Technical Issues

Formatting issues were mentioned as cultural issues but they are also technical issues. Usually the stored data should work in all versions of the product regardless of the used localization. The software must do conversions between the storage and presentation format every time the data is shown or entered.

The conversions between uppercase and lowercase letters are also troublesome. There is not always a one-to-one mapping between uppercase and lowercase characters. For example, in German the uppercase equivalent of ß is SS. Some languages do not even have the concept of the uppercase and lowercase characters.

Sorting rules for the textual data forms another typical problem, because the alphabets and their ordering varies between languages. In Spanish the letter combination ”ll” is considered as a single letter that is sorted after the letter l. As a result the word llave is sorted after the word luz, which is quite the opposite when compared to the English sorting order.

Even bigger issue is formed by different character encodings. Traditionally most of the character sets have used one byte for each character so one encoding may contain at most 256 different characters. For practical reasons most of the character encodings are compatible with the ASCII[1] by sharing the first 128 characters. However, Chinese, Japanese and Korean require more characters so they have character encodings that use variable number of bytes for each character. Usually programs designed for western markets need to be adjusted afterwards to support the multibyte encodings.

[1] American Standard Code for Information Interchange is a 7-bit character encoding that is based on the ordering of the English alphabet.

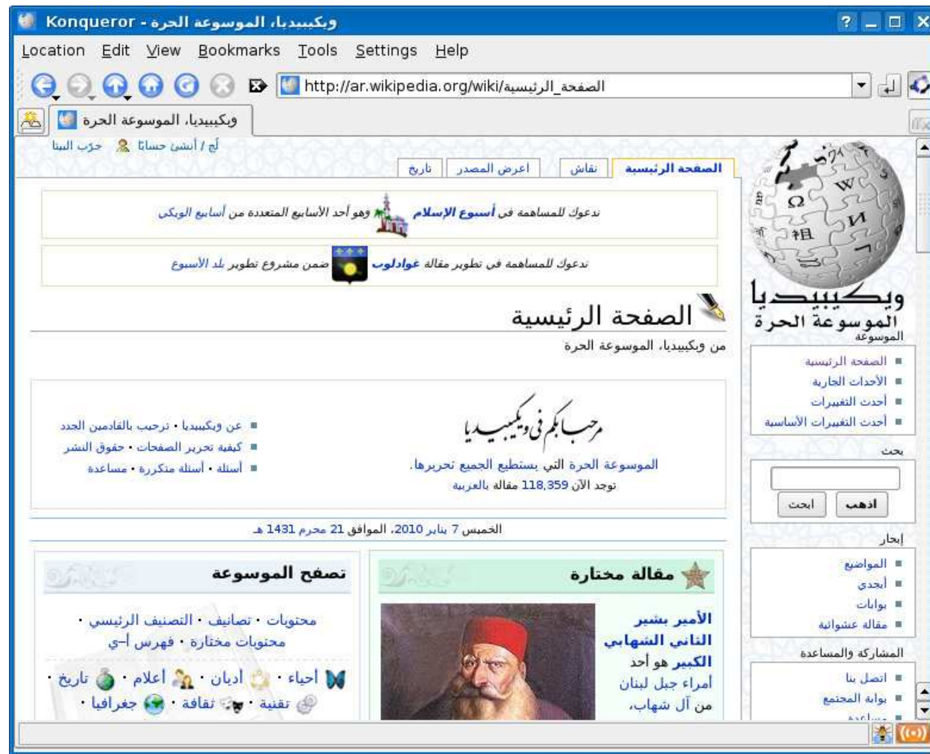

Figure 2.3: Arabic layout is from right to left

Nowadays things are a bit better because of Unicode standard [8]. The main goal of the Unicode is to be able to support all possible character sets simultaneously and currently it covers over 45.000 characters. It mostly removes the need for conversions between different character encodings and makes it easier to process the data.

Finally, not all languages are written from left to right. Languages like Arabic and Hebrew are written from right to left and furthermore the whole user interface must also be laid out from right to left. This is demonstrated in Figure 2.3 that shows the Wikipedia's main page in Arabic.

2.2 Internationalization

In the previous section it is explained how much issues are related to the localization. It would be waste of resources if these issues would be solved again and

again in every new project. Instead it makes sense to solve most of these issues beforehand by planning and preparing the product for the localization. This process is called internationalization.

By LISA's[2] definition, internationalization is the process of enabling a product at a technical level for localization [19]. In practice the internationalization is done by abstracting all the culture, language and market related functionalities that the product has. By doing this the product will be easy to adapt for specific market and it does not require any redesigning when it is localized. With perfect internationalization the localization process should be reduced to be a pure translation task.

There is a general rule that if the internationalization is not done properly the localization will take twice as long and cost twice as much than a similar project with good internationalization. This rule is about localization generally. In the software industry the difference can be even bigger [19].

2.3 Globalization

Some companies use globalization as a synonym for the internationalization and usually they prefer using the term globalization instead of the internationalization. However, we define the globalization to be a two-step process that binds the internationalization and localization processes together. Their interrelations are illustrated in Figure 2.4.

The globalization means all company wide actions that are done to adapt products according to the demands of some specific local market. Typically there are multiple parallel localization processes going on at the same time, each of them targeting their own market area. Their success is largely based on the preceeding internationalization.

In the 1980s, software companies used to have big in-house localization divisions. Nowadays those divisions have disappeared and they have been replaced by localization vendors, which have specialized themselves in the field of localization [13]. To use these service providers the companies must be fully committed

[2] The Localization Industry Standards Association

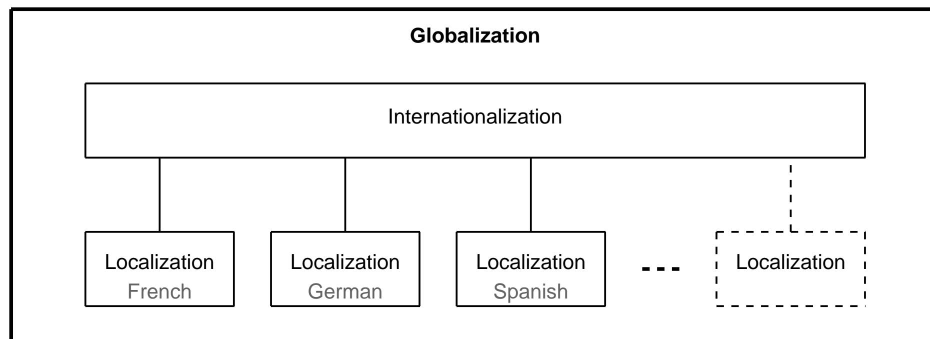

Figure 2.4: Globalization consists of the internationalization process and multiple parallel localization processes

to the whole globalization process. If they want to externalize the localization phase to subcontractors they cannot ignore the internationalization phase, because it would guarantee the failure of the project.

Chapter 3

The Globalization Process

The globalization process is present everywhere in the product development. The project planning and preparation is the key for the successful globalization process. As it was stated in the previous chapter the globalization includes both the internationalization and localization processes. Figure 3.1 on page 12 shows the cycle of the globalization process and the relations between internationalization and localication.

3.1 Project Planning and Preparation

The globalization process requires good planning and preparation or else there will most likely be expensive costs and problems with the deadlines later in the project. Potential market areas must be evaluated to get enough information about the local requirements and related development costs. The internationalization and localization needs are based on both the market analysis and globalization requirements. Another important task is the selection of the localization partner, which has the competencies required by the project. These are the main topics that are discussed in this section.

3.1.1 Market Evaluation and Globalization Requirements

The first step in the globalization process is to decide the required localization level that simply specifies how much translation and customization is considered

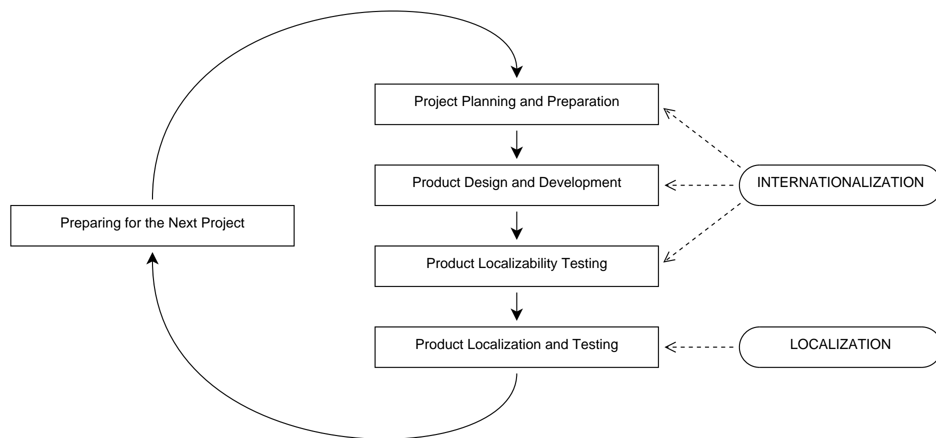

Figure 3.1: Globalization process has a cyclic nature

necessary for different language editions [16]. The levels range from translating nothing to shipping a completely translated product with customized features. The selection of the suitable level for each market area is based on the market analysis.

The market analysis should include both global and local requirements for the product functionality and content, not forgetting the local business processes and regulations. An attempt to predict all the potential difficulties in each market should be made as well as the plans to overcome them. The technical aspects are taken into account when considering the strategic value and estimated revenue of the market. The whole evaluation process should use the expertice of the local representatives and partners in the potential market areas. Based on the results of the market analysis the company decides the degree of internationalization.

When the market areas are evaluated, the company should produce a global product specification that describes everything needed in the product internationalization and localization. The specification divides the requirements to those needed by all market areas and to those needed by specific markets. It defines global content and functionality of the product as well as its local content needs. In addition, it contains the requirements of local regulations, business processes and languages. All the requirements are prioritized on their estimated need and their market value so that their importance can be evaluated from the business point of view.

3.1.2 Selection of the Localization Vendor

In Section 2.3 it was noted that the localization process is often outsourced to localization vendors. The vendors should be chosen very carefully because they differ in pricing, experience, staff and services. If the vendor is chosen in early stages of the project, they can provide help for the cost assessment and support for the product internationalization. This, however, should not be taken for granted because the provided services vary greatly between different localization companies and a poor choice may end in disaster. The localization process of the vendor should be reviewed and its compatibility with the product development process should be estimated [26].

The localization vendor should understand the product and its target audience. If the product is targeted to normal end-users, the translation should be informal and for advanced users the translation should be more technical. The vendor needs to have translators that are able to fulfill the requirements. Some vendors employ full-time translators while others use subcontractors. If the company uses subcontractors they may not be able to hire suitable translators in time and the project schedule may be delayed.

Another question is whether the services of translators or localizers are needed. Translators do only the translation from one language to another and nothing more. Localizers, on the other hand, have much wider spectrum of skills. For example, they may change graphics, set the user interface to use right to left layout and alter the size and position of the components.

It pays off to check the references of the localization vendor. Who are their biggest clients and how they work with them? Their localization process is most likely optimized for those clients and their staff is used to work within certain guidelines. They may find it hard to adjust the localization process to be suitable for other product development processes, which are not similar to the processes of their biggest clients.

Figure 3.2: Localizability can be tested with pseudo-localization

3.2 Product Design and Development

The global product specifications are the basis for the product design as they define global functionality, user interface requirements and local particularities. During the product development the localizability of the application should be periodically tested to ensure that it is free of any design or implementation errors. Small issues require only bug fixes but bigger problems may demand redesigning if they have not been addressed before.

As a part of the the product development a localization kit is created. It will be given to the localizers and its main purpose is to help them to work faster and to improve the localization quality. The localization kit is discussed in more detail in Section 3.4.

3.3 Product Localizability Testing

The localizability testing is done in order to find and correct any possible design flaws and bugs that would prevent the localization of the product. The possible issues include text expansion problems, character encoding problems, hard-coded strings that should be localizable and sorting problems. Usually the bugs are such that they only appear after the localization is done. This is a problem because the localization is done after the product development is finished. However, there are some methods that can be used to verify the product's localizability [22].

Pseudo-localization is a simple method that helps to locate many localizability issues without the need for real localization [12, 16]. It can be implemented in many different ways, but the basic principles are usually the same. In Figure 3.2

there is an example of the pseudo-localization. The letters in English text are replaced with similarly shaped non-English symbols like for example a with à or ä, c with ĉ or ç and n with ń or ñ. This kind of substitutions are chosen to keep the pseudo-localized version readable.

In English the words are relative short and the translations require usually more space. This can be estimated either by adding extra characters to the English strings or by duplicating the vowels so for instance the ”Filename” would become ”Fíílěěñààměě” or ”{Fíílěěñààměě}”. The latter example has simple prefix and suffix characters that help to identify the cases when the string does not have enough space and is clipped from either or both ends. With this method the developer can still understand the meaning of the strings so the user interface remains usable and it can be tested more thoroughly. It is a clear benefit when compared to a method that replaces strings with strings that consist of random special characters.

The character substitution method is good for detecting most types of localizability issues. Still, it is not optimal to detect hard-coded strings. Those are more easily spotted when all characters in the user interface texts are replaced with ”X” or some other easily recognizable character.

These two pseudo-localization methods provide good coverage for the common localizability issues. They help to guarantee that the product internationalization is done correctly. The pseudo-localization is easy to automatize because both methods modify the localizable resources according to very simple rules.

3.4 Product Localization and Testing

Some companies let the third-party contractors and localizers do the whole localization by providing them a package of all necessary files and information that is needed to create a localized version of the product. This package is called a localization kit and it should contain localization guidelines, schedule information, build environments, source files and reference materials [16]. The localizers may also need tools with which they can change the user interface fonts, resize and move user interface components, change the text encoding, replace graphics and modify the keyboard shortcuts.

Other companies prefer to keep the localization process more in-house. There are many possible reasons for that. For example, they may not want to give the source codes and build environments to subcontractors or they may not like the idea that the user interface is changed and redesigned by the localizer. Anyway, they should have guidelines for the good user interface design that minimizes the propability of the user interface changes. The font names, sizes and styles as well as graphics, icons and colors can be defined in the same resource files as the translations. If the product can be customized fully with the resource files, there is no need for the development environment and source code.

Personally I prefer the latter approach because it is able to give better continuity in the product development. The externalization of the whole localization process seems to be a fire-and-forget type solution, because it does not give any support for the localization of the future product versions. In my opinion localizers should be able to change only those things that are explicitly allowed. If the internationalization has been only partial and there is need to change for instance the component layout, it should be done in the main project so that the subsequent product versions will not suffer from the same layout problem. This way also the developers get an instant feedback and they will have a better possibility to improve their skills in the field of the internationalization.

3.5 Preparing for the Next Project

A good globalization process has a cyclic nature. The success, or failure, of finished globalization process is evaluated and the experiences are used to improve the process so that the subsequent projects will benefit from them. Also all localized resources are made available for the next project in order to prevent the same things to be localized all over again. Usually the translations are stored into so called translation memory, which is used to do the initial translation before the files are sent to the localizer.

Chapter 4

Localization Bindings

In previous chapter it has been explained why the localizable content must be separated from the user interface. To do that there must be some kind of mechanism to bind the user interface components with the localized resources. The basic idea is simple. All localizable strings are usually replaced with method calls that identify the substituted string somehow. The method uses this identifier when it gets the correct translation from the translation database.

One exception to the rule is represented in Section 4.5 when the component instance is used as an identifier. It does not need the substitution because the components are referred directly when needed.

Now the main question is what should be used as identifier for the binding. These different binding methods are described in this chapter and their benefits and deficiencies are evaluated. Some of their real life implementations are represented later in Chapter 5.

4.1 Identifying by Integer

In the early ages of the software industry the computers were slow and with scarce memory. At those times the efficiency and frugality were well appreciated. Using integers as identifiers for the localized texts was a perfect solution, because the localized resources could be stored into array and, as a result, they could be indexed very quickly. Even today there are situations when the software has to work with

Listing 4.1: Obscurity can be a major problem with integer identifiers

```
// File: Culture_de_DE.cs
// Defines the German translations for the identifiers.
class Culture_de_DE {
    public static void Initialize() {
        // Translations are specified in string array.
        string[] translations = new string[] {
            "Öffnen",
            "OK",
            "Abbrechen",
            "Durchsuchen...",
        }
        LocalizationManager.SetTranslations(translations);
    }
}

// File: OpenDialog.cs
// Defines the dialog (only relevant content is shown).
class OpenDialog {
    private void Initialize() {
        // Component initialization code omitted.

        // The translations are obscure when they are referred with pure integers.
        this.openLabel.Text = LocalizationManager.GetText(0);
        this.okButton.Text = LocalizationManager.GetText(1);
        this.cancelButton.Text = LocalizationManager.GetText(2);
        this.browseButton.Text = LocalizationManager.GetText(3);
    }
}
```

limited resources so this reasoning is still valid. The embedded devices are typical examples of such cases.

However, there are some very notable problems with the integer identifiers. Obscurity is likely the first one to be noted and it is demonstrated in Listing 4.1. The plain integer does not give any hints about the referred resource. A standard way to solve this issue is to give the integers meaningful names by defining either constants or an enumerated type. This way one does never actually use the integers directly because they are well hidden behind the defined names.

The translations can be defined in one of three ways. All translations can be defined at once in the array initialization, they can be defined one by one in a random order after the array has been first allocated or they can be defined in a resource file. Again they all have their own problems.

The first alternative lacks a straight connection between the identifier names and the array slots. The completely unconnected identifier and array definitions are hard to keep synchronized and a simple mistake in either one can cause the

Listing 4.2: Plain integer identifiers replaced with an enumerated type

```
// File: Identifier.cs
// Defines the identifiers for the localizable resources. The last identifier in the enumeration
// will not have translation. It is used to determine the required array size.
// Note that the enumeration values should be automatically assigned because there are no benefits
// for defining the values explicitly.
enum Identifier {
    OpenLabel,
    OkButton,
    CancelButton,
    BrowseButton,
    LastIdentifier
}

// File: OpenDialog.cs
// Defines the dialog (only relevant content is shown).
class OpenDialog {
    private void Initialize() {
        // Component initialization code omitted.

        // Enumerated type describes the referred translation clearly.
        this.openLabel.Text = LocalizationManager.GetText(Identifier.OpenLabel);
        this.okButton.Text = LocalizationManager.GetText(Identifier.OkButton);
        this.cancelButton.Text = LocalizationManager.GetText(Identifier.CancelButton);
        this.browseButton.Text = LocalizationManager.GetText(Identifier.BrowseButton);
    }
}
```

Listing 4.3: Translations defined with the enumerated type

```
// File: Culture_de_DE.cs
// Defines the German translations for the identifiers.
class Culture_de_DE {
    public static void Initialize() {
        // The array's size is determined with a special enumeration member. In the C# this would not really
        // be necessary because there are other means to find out the number of enumeration members.
        string[] translations = new string[Identifier.LastIdentifier];

        // The translations can be specified in any order.
        translations[Identifier.OpenLabel] = "Öffnen";
        translations[Identifier.OkButton] = "OK";
        translations[Identifier.CancelButton] = "Abbrechen";
        translations[Identifier.BrowseButton] = "Durchsuchen...";

        // For simplicity we use here a static method to activate the translations.
        LocalizationManager.SetTranslations(translations);
    }
}
```

majority of the translations to be broken. Such a mistake can happen by adding or removing an item in either one of the data structures and by forgotting to repeat the action to the other data structure.

The problem with the second alternative is related to the array size. When

the number of identifiers changes the allocation size of the array must also be updated to prevent the code from exceeding the boundaries. This problem can be circumvented easily by defining a special enumeration member that is always kept last in the enumeration definion. Same applies for constants but now the value of the last constant must be updated manually. Listings 4.2 and 4.3 show an example that uses an enumerated type as identifier.

The third alternative suffers from the obscurity because the integers are used as keys in the resource file and all the benefits from the defined constants are lost. This is often compensated by providing the same information with comments in the resource file. This alternative is suitable when the development environment is responsible for the code generation and integer assignments.

4.2 Identifying by User Interface Texts

This binding method is especially well-suited when one needs to implement localization support for an existing application that has not been internationalized. Generally the adaptation is done by surrounding all localizable texts inside a method call and the text is left as a parameter for the method. The localization method uses the supplied string as a key when the correct translation is searched for. One clear benefit of this method is the self-documenting nature of the identifiers. Furthermore the identifiers can be used as substitute when there is no real translation available so the localization method can always return a sensible value.

Sadly the identifiers are not entirely trouble-free. This method requires that each unique text has unique interpretation. Identical strings with different meanings cannot be localized if the target language has distinct translations for them. The most simple solution for this situation is to define an additional context parameter for one of the colliding texts.

4.3 Identifying by Component Name

The component name must not be mixed with the component instance. The binding process for these two methods is completely different. The component name

is used in a similar manner than integers or user interface texts and the component name is simply used to query the translation database. On the other hand, the component instances are used to directly manipulate the components.

The component names must be unique in the whole application. Often this requirement is not fullfilled. The user interface components may form hiearchies that require uniqueness only from the immediate children of the each component. In this case they must be differentiated somehow. One option is to identify the components by concatenating their name with the names of their parent components.

Because the components are identified by their name, they cannot be renamed after the localization has been done. Normally this is not a big issue but it may sometimes complicate the redesign of the user interface. Also all components must be translated separately even if some of them have identical content because the components cannot share identifiers.

4.4 Identifying by Arbitrary Name

Identifying translations with an arbitrary name is a generalized version of all other methods mentioned before. All dependencies to components and component contents are removed. Because there are no dependencies one can bind a single translation with multiple components. Each of the standard buttons, menus, dialogs and messages have only single translation no matter how many times the translations is used. Since the identifiers are freely composed strings, they can also carry some context information: the prefix in the identifier "but_cancel" indicates that it is a cancel label for a button component.

Applications may use both shared and application specific localization files. If the shared localization file is very comprehensive by having translations for all common user interface strings, the developer can create mostly localized application by binding the components with the existing translations. The translation is only needed for the new strings, which do not exist in the shared localization file. Most of these untranslated strings are application specific.

This is a big benefit when compared to the alternative wherein the initial localization is done to all components automatically with the translation memory.

Listing 4.4: Instance binding is totally different approach for localization

```
// File: Culture_de_DE.cs
// Defines the German translations for the identifiers.
class Culture_de_DE {
    // The localization methods are utilizing the overloading.
    public static void Localize(OpenDialog dialog) {
        dialog.OpenLabel.Text = "Öffnen";
        dialog.OkButton.Text = "OK";
        dialog.CancelButton.Text = "Abbrechen";
        dialog.BrowseButton.Text = "Durchsuchen...";
    }

    public static void Localize(SomeOtherDialog dialog) {
        // Implementation omitted.
    }
}

// File: OpenDialog.cs
// Defines the dialog (only relevant content is shown).
class OpenDialog {
    private void Initialize() {
        /// Component initialization code omitted.

        // LocalizationManager forwards this call to the active
        // culture class and calls its Localize() method.
        LocalizationManager.Localize(this);
    }
}
```

After the initial localization the localizer has to check every translation by hand to make sure the translation is correct. Furthermore the developer knows the context of every binding and if the identifiers are clear and intuitive, he is most likely able to do a better job than anyone else. The localizer lacks comparable insight.

This binding method may seem almost identical to the binding with the user interface texts. In fact, because the only difference is the content of the key, their implementations can usually support both binding methods at least partially.

4.5 Identifying by Component Instance

Component instances are a bit different when compared to the other identifying methods, because they do not require any changes to the component definitions of the user interface. However, since the localizations should be separated from the component definitions, the component instances must be exposed for the localization purposes. Because the public access for the internal data structures is

considered as a bad programming style, the programming language should provide means to limit the access for the exposed components

It is easy to alter all localizable aspects of the components, because they are fully exposed and the localizations are done with real program code instead of being defined as plain data. Unfortunately they expose much more than is really necessary for the localization purposes so this method is a poor choice especially when the localization process is outsourced. An example of the instance binding is shown in Listing 4.4.

Usually this binding method requires that each language has its own version of the application or alternatively a language specific dynamic library. Dynamic libraries allow single application to support multiple languages and the languages can be installed and updated when necessary. It is also possible to build a multilingual application in single executable, but its release is susceptible to delays because it depends on all translation projects of the supported languages.

4.6 Summary

In this chapter it has been shown that there are many different binding methods. They each have slightly different properties and none of them is universally superior to others. The selection of the binding method must be based on the requirements of the project. Some possible requirements are runtime performance, simplicity of the implementation, translation database format and identifier type. *Identifying with the integers or component instances is the best alternative if the performance is the most critical requirement.* However, other properties are generally more important and the other identifier types are often more suitable than these two.

If the binding identifiers are managed by the user interface editor, the identifiers can be integers, component names or component instances. The handling of these identifier types is natural for the editor. However, these types usually require specialized editors for the translation files, because the integer identifiers lack proper context and often the same applies to component names and instances too. Editors can also use component texts as identifiers, because this identifier type requires primarily that the editor stores unique identifiers to the translation

database with optional context information.

If the identifiers are directly managed by the developer, component text and arbitrary name are usually the most suitable identifier types, because they both have a human-friendly nature. These two identifier types have much in common and the main difference is that the other uses real user interface strings as identifiers while the other uses abstract strings. The component text identifiers are always bound to the default user interface language while the arbitrary name identifiers remove all dependencies between the default and target languages. This has an important consequence: arbitrary name identifiers can contain additional context information while the component text identifiers are limited to the expression of the default language and the context information must be stored separately.

The easiness of the implementation is more related to other design decisions than to the identifier type. The component instance as identifier is an exception though. It requires always some kind of code generator and parser, because the use of instances requires that the translation database is a source file. If the generated source file can be edited manually, the parser must be sophisticated to be able to handle the content.

The properties of the different identifier types can be summarized in a few sentences. Identifying by component text, component name or arbitrary name are easy to implement and they do not have any limitations for the format of the translation database. The integer identifiers are also easy to store but the abstract nature of the integers may cause some trouble. The component instance is the most inflexible identifier type, but the direct access to the components makes it very fast. All in all none of these identifying methods is indisputably superior to others as the choice is always affected by other design criterias.

Chapter 5

Existing Internationalization Technologies

At this point it is time to investigate the existing internationalization solutions. The main focus is on the translations because the other internationalization aspects such as the date and number formatting are normally standardized and tightly integrated in the framework. After this chapter the reader should have a general understanding of some commonly used localization implementations.

5.1 GNU gettext Library

The GNU gettext library is a part of the GNU Translation Project and it provides a set of tools to help the programmers and translators to produce multi-lingual applications [1]. It consists of a runtime library, some stand-alone programs and a set of rules how programs should be written. The translations are identified with the user-interface strings. In Section 4.2 is an overview of this identification method.

There are two kind of files in the gettext: portable object and machine object files. Portable object files are human-readable and they define the translations for each translatable string as can be seen in Listing 5.1. Each file contains translations only for a single target language and the file is named according to the language. Because the files are human-readable, editing can be done even with a

Listing 5.1: Snippet of the GCC compiler's Spanish portable object file

```
#: lto-wrapper.c:234
#, c-format
msgid "could not write to temporary file %s"
msgstr "no se puede escribir en el fichero temporal %s"

#: lto-wrapper.c:344
#, c-format
msgid "fopen: %s"
msgstr "fopen: %s"

#. What to print when a switch has no documentation.
#: opts.c:341
msgid "This switch lacks documentation"
msgstr "Esta opción carece de documentación"

#: opts.c:1328
#, c-format
msgid " No options with the desired characteristics were found\n"
msgstr " No se encontraron opciones con las características deseadas\n"
```

simple text editor but usually specialized editors are used.

The initial portable object file is created from the source files with a tool that extracts all strings that are marked localizable. The gettext recognizes that programs evolve and therefore the localizations must be easy to synchronize with the new program versions. The gettext provides a tool that is used to merge the extracted portable object file with the existing file that is localized. The merge tool comments out obsolete entries that have been removed, adds new untranslated entries and updates the references to source code in other entries.

The portable object files are intended to be used only in the translation phase. The finished translations must be transformed to machine object files before they can be used with the application. The machine object files are binary because they are meant to be read only by programs and because the retrieval of the translations should be efficient.

There is also a compendium portable object file that is used as translation memory. It contains common translations that have been saved before and the translators may do the initial translation with the compendium. Of course they can also add new and update existing entries in the compendium. The compendium files are meant to be shared between members in the translation team.

5.2 Localization with Java

In Java the translations are accessed through resource bundles [2]. Resource bundles are subclasses of the abstract ResourceBundle class. Java provides two different implementations, ListResourceBundle and PropertyResourceBundle, but new resource bundles can be implemented by extending the base class if those two are not suitable. This may be necessary if the translations need to be fetched from a database or they need to be identified with something else than a string, which is the only supported key type in the default implementations.

The ResourceBundle class has a static method to locate and load the bundles by name. The method combines the given name with the identifier of the default locale of the system. If the method is called with "MyResource" parameter and the default locale is en_US (U.S. English) the method primarily uses "MyResource_en_US" as the bundle name. If the bundle does not exist, the method gradually degrades the name until a matching resource bundle is found. The "MyResource_en_US" degrades first to "MyResource_en" and then to "MyResource" and if none of them is found the method throws an exception. The resource bundles can also share the data if the bundles share the base name, because the requested resources are searched from the bundles in the degrading order until the requested key is found.

The ListResourceBundle is an abstract class with a single abstract method, which the subclass must implement. This method should return translations as an array of key-value pairs as can be seen in Listing 5.2. The key must always be a

Listing 5.2: Example of the Java's ListResourceBundle implementation

```
import java.util.ListResourceBundle;

public class MyResource_de_DE extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "OpenLabel", "Öffnen" },
            { "OkButton", "OK" },
            { "CancelButton", "Abbrechen" },
            { "BrowseButton", "Durchsuchen..." }
        };
    }
}
```

Listing 5.3: The property definition file has very straightforward syntax

```
# The PropertyResourceBundle supports only string type values.
OpenLabel=Öffnen
OkButton=OK
CancelButton=Abbrechen
BrowseButton=Durchsuchen...
```

string but the value can be any object type. This can be utilized by instantiating culture specific user interface components such as a component to show the address information. However, because ListResourceBundle must be defined in the source code, it sets additional skill requirements for the localizers. Often it is better to use some other resource bundle type that does have a better editor support and does not require special skills from the localizers. The ListResourceBundle is still suitable for some specific localization tasks like specifying the culture specific components as mentioned before.

The PropertyResourceBundle differs from the ListResourceBundle by being a concrete class that is not subclassed but instantiated with a parameter that specifies a property file containing the key-value pairs. The syntax of the property file is extremely simple as can be seen in Listing 5.3. The file contains only key-value definitions, comments and empty lines. The simplicity has its price though: only string values are supported. That does not limit its usefulness with translations and even graphics can be referenced by resource name or file path but more advanced usage will be cumbersome.

Inprice, Inc. has implemented a new resource bundle for their JBuilder product. Their implementation that is ArrayResourceBundle uses integers as keys and they index the translations in the array. This method was described in the previous chapter in Section 4.1. It demonstrates quite nicely how much more efficient the integer keys are. ListResourceBundle's time was over 6,5 times and PropertyResourceBundle's time was ten times as much as the ArrayResourceBundle's time [25]. The tests were made by the JBuilder international team so the results should be taken with a grain of salt. Still, the results are pretty clear and they are in line with the intuition.

Listing 5.4: The .resx files used by Windows Forms are human-editable

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!-- There should be a special header here that is too long to be shown -->

  <data name="BrowseButton" xml:space="preserve">
    <value>Durchsuchen...</value>
    <comment>Button opens a file dialog.</comment>
  </data>
  <data name="CancelButton" xml:space="preserve">
    <value>Abbrechen</value>
  </data>
  <data name="OkButton" xml:space="preserve">
    <value>OK</value>
  </data>
  <data name="OpenLabel" xml:space="preserve">
    <value>Öffnen</value>
  </data>
</root>
```

5.3 Windows Forms Localization

Windows Forms is an user interface assembly[1] for the .NET Framework. Its localization support is implemented with resource files and there are two different resource file formats that are differentiated by their file name extensions: .resources and .resx [9].

The .resources file format is a binary format that is embedded within a .NET assembly and accessed with the ResourceManager class. It supports string resources as well as other type resources. Different languages are supported by packaging the localized resources as individual satellite assemblies, which are automatically found at runtime. The satellite assemblies are located in the main assembly's subdirectories, which are named after the satellite assembly's language and region.

The .resx file format is more versatile design-time format for producing .resources files. It is XML so it is also human-editable. Naturally it supports the same resource types as the .resources format and it has an additional support for comments and file references. These properties make it superior to the .resources format for the localization purposes. In Listing 5.4 is a simplified example of the

[1] An assembly is the primary building block of a .NET application and can take the form of a dynamic link library or executable file.

Listing 5.5: Strongly-typed resource reference is verifiable by the compiler [27]

```
// Traditional fragile resource reference.
MessageBox.Show(resourceManager.GetString("InsufficientFunds"));

// Strongly-typed resource reference that is checked by the compiler.
MessageBox.Show(Form1Resources.InsufficientFunds);
```

.resx file format.

Strongly-typed resources are an interesting speciality that was introduced in the .NET Framework 2.0, although there is no technical reasons why it could not have been implemented in the previous .NET Framework versions [27]. The strongly-typed resource is a generated class that contains resource key names as properties and the class is updated always when the .resx file is changed. The strongly-typed resources have essentially the same purpose as the constants or enumerated types have. They replace the fragile name references with references that are validated by the compiler as can be seen in Listing 5.5. The generated class itself can be seen in Listing 5.6 on page 31. Another good feature is the compatibility with code completion so the references can be written quickly and reliably. As a result, the strongly-typed resources provide a nice solution for the problem how the messages in the source code should be localized.

Each dialog window, a form, has a Localizable property that controls whether the form should be localizable or not. Normally the property values of the user interface components are serialized as simple property assignments. When the Localizable property is set to true, the values are stored into .resx resource file and the direct property assignments are replaced with code that gets the values from the resource file. In the resource file each value is identified with the component name that is concatenated with the property name.

The generated code depends on the version of the Visual Studio as the Visual Studio 2003 uses property assignment model and the Visual Studio 2005 uses property reflection model. In the property assignment model the value part of the property assignment is replaced with a method call that gets the correct value from the resource file. In the property reflection model only a single method call is generated. First the method loads the resource file completely and then it traverses the object hierarchy of the form and sets all the values at once.

Listing 5.6: Generated strongly-typed resource class [27]

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
internal class Form1Resources {
    private static global::System.Resources.ResourceManager resourceMan;
    private static global::System.Globalization.CultureInfo resourceCulture;

    [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
            "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
    internal Form1Resources() { }

    [global::System.ComponentModel.EditorBrowsableAttribute(
            global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Resources.ResourceManager ResourceManager {
        get {
            if (object.ReferenceEquals(resourceMan, null)) {
                global::System.Resources.ResourceManager temp =
                        new global::System.Resources.ResourceManager(
                                "WindowsApplication1.Form1Resources",
                                typeof(Form1Resources).Assembly);
                resourceMan = temp;
            }
            return resourceMan;
        }
    }

    [global::System.ComponentModel.EditorBrowsableAttribute(
            global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Globalization.CultureInfo Culture {
        get {
            return resourceCulture;
        }
        set {
            resourceCulture = value;
        }
    }

    internal static string InsufficientFunds {
        get {
            System.Resources.ResourceManager rm = ResourceManager;
            return rm.GetString("InsufficientFunds", resourceCulture);
        }
    }

    internal static System.Drawing.Bitmap NationalFlag {
        get {
            System.Resources.ResourceManager rm = ResourceManager;
            return ((System.Drawing.Bitmap)(rm.GetObject("NationalFlag", resourceCulture)));
        }
    }
}
```

The property assignment model tries to get localized value for every localizable property while the property reflection model assigns values only to localized properties. Usually the number of localizable properties is considerably larger than the number of properties that are localized. The property reflection model is more efficient in this situation.

The localization is done with the IDE's[2] user interface designer or with a dedicated tool like Windows Resource Localization Editor, WinRes. The WinRes is essentially a cut-down version of the Visual Studio Forms Designer, which is used by the Visual Studio IDE. With the WinRes the localizer does not have to install a full .NET development environment and the localizer only needs the resource files of the application and the compiled application itself. The sources are not needed and it is an advantage when the localization is done by an external localization vendor.

The actual localization process is pretty much identical to the normal user interface editing in the IDE's designer. The localizable properties are exposed in the editor and the localizer can change their values as needed. The developer cannot control which values are exposed to the translators, because this choice is already made by the developers of the user interface components. This can be a problem if the developer does not want that the localizer has a possibility to change the user interface layout or some other particular properties.

Some sources [27] list also .txt and .restext files as possible resource file formats. The both formats are actually the same as the only difference is the file name extension. They are plain text files with simple key-value pairs for string resources but they do not have support for comments like the Java's PropertyResourceBundle does. Neither do they have a direct support in the .NET framework and it is the main reason why I do not consider them to be real alternatives for the localization purposes.

[2]Integrated Development Environment is a software application that provides comprehensive facilities to computer programmers for software development.

5.4 Tekla Technology

Tekla Corporation uses a set of libraries that provide cross-platform support to all Tekla's applications. The most important domains supported by the libraries are database management, user interfaces and graphics [5]. With these libraries Tekla's products have been able to adapt themselves to new platforms when necessary.

The user interface library is A-kit [6] and it has supported VMS/OpenVMS, Microsoft Windows and about 15 different unix variants. Current A-kit version supports Windows XP, Windows Server 2003, Windows Vista and Windows 7. The first A-kit version was released 1990 and the following major releases were made 1991, 1995 and 1999. All major releases have incorporated big internal changes in the A-kit implementation.

The A-kit supports typical user interface components such as buttons, text boxes, check boxes, labels, graphics workareas, pull-down and pop-up menus, tool bars and status bars. More advanced features include support for state machines and connection of the dialogs to the application functions and variables. The A-kit dialogs use table layout and the content is resizeable.

The user interfaces are designed with AilEd editor [7] and they are stored into AIL[3] files. The AIL is one sublanguage of MDL[4], which is a resource file description language designed in Tekla. In addition to the user interfaces the MDL is used by other Tekla libraries to define for example databases and database conversions. The MDL based resource files are handled with R-kit function library [4].

The AIL files, like all MDL based languages, are human-readable hierarchical object-attribute based languages. The language definition is a list of available objects and their attributes and child objects. The attribute values can be strings, integers, floating-point numbers, keyword values or tuples, which are compositions of multiple types. The attributes and child objects can be defined to be either obligatory or optional as well as repeatable or non-recurring. The language definition file is read by the MDL compiler and it generates C and C# code that is compiled and linked to the application. The generated code enables the use of

[3]Application Interface Language
[4]Multi-purpose Definition Language

Listing 5.7: Example of the translation definitions in the AIL file

```
string lbl_open
{
    entry = ("enu", "Open");
    entry = ("deu", "Öffnen");
    entry = ("esp", "Abrir");
};

string but_ok
{
    entry = ("enu", "OK");
    entry = ("deu", "OK");
    entry = ("esp", "Aceptar");
};

string but_cancel
{
    entry = ("enu", "Cancel");
    entry = ("deu", "Abbrechen");
    entry = ("esp", "Cancelar");
};

string but_browse
{
    entry = ("enu", "Browse...");
    entry = ("deu", "Durchsuchen...");
    entry = ("esp", "Explorar...");
};
```

R-kit functions with the defined language.

The translations are defined in the AIL resource file. An example of the definitions is shown in Listing 5.7. Each translation is defined in named "string" object and the name of the object is used as identifier for the translation. The object contains one or more "entry" attributes and the attribute values are compounds of two strings: language name and translation. The translations can be defined in multiple files but the storage format dictates that all translations for single identifier must be defined in same file, because the object definitions cannot be splitted. The localizers cannot share one translation file so in practice they each have their own language specific copy of the file and the files are combined into single file afterwards. The translation files are edited either with AilEd editor or with StrEd, which is a specialized editor for the translations.

The AIL file is read by the AIL compiler and translated into an AID binary file that contains descriptions of the user interface elements including the translations for all supported languages. The compiler also produces a source file named

dakbind.c, which contains bindings for callback function pointers and member offsets of the dialog structures. The dialog structures are used to connect the user interface data with the application variables.

From the localization point of view the use of arbitrary names as identifiers reduces the localization costs of the new dialogs, because it is possible to reuse the common identifiers over and over again. If the identifiers were based on the component instance they would not be reusable and the total number of the translations would be considerably larger. This would mean higher localization costs, because the costs correlate with the number of the translations.

All common internationalization solutions use language specific translation files as it makes the management of the translations easy. However, the A-kit considers the translations to be a part of the user interface descriptions and therefore all translations are stored into the compiled AID file. As the AIL format requires that all translations of single identifier must be defined together, the management of the translations is unnecessary awkward. The file format design is made on terms of the A-kit implementation instead of general usability.

Another inconvenience with the AIL format is the lack of Unicode support. The AIL files containing translations are not really pure text files, because the translations are defined in language dependent character encodings. The Western European languages are stored with ISO-8859-1 or ISO-8859-15 encodings but more exotic languages require their own character encodings. If the file format would be designed today, it would surely use UTF-8 or some other Unicode encoding. However, this encoding issue is mainly a problem if someone wants to edit the AIL files manually with a text editor instead of the AilEd or StrEd editors.

5.5 Summary

In this chapter it has been shown that the software industry uses many divergent approaches in the product internationalization. All presented technologies are succesfully used in large-scale applications, so none of them can be considered irrelevant. They are all potential candidates for the software internationalization and the choice between them is usually more dependent on the development platform than on anything else, because the development platform dictates how well

these technologies are supported.

Most of the localization techniques have to rely strongly on machine translation, because the strings in the user interface are not directly linked with existing translation. The tools must identify identical or similar strings in order to ease the workload of the localizer. The string maintenance and the avoidance of unnecessary and costly translations are considered to be the most important problem in the field of the localization [15].

The arbitrary names, which the Tekla technology uses as identifiers, reduce the need for assisted translations, because the user interface developer can reuse the existing identifiers that have already been translated. The identifiers contain hints of the context, so it is more likely that the developer is capable to choose the correct identifier. The reuse of the identifiers applies partly to the GNU gettext too. However, its identifiers do not normally specify the context so the developer has to find out where the existing strings are used before he knows if they have the same context. Because the work is tedious, it is likely that the identifiers are not checked beforehand and their correctness is verified only at the localization testing.

The GNU gettext and Windows Forms try not to stress the developer of the application any more than necessary. Actually this is one of the design goals of the GNU gettext, because the maintainers of the programs already have enough concerns to worry [1]. On the other hand, the Tekla technology has increased the workload of the developers by requiring them to add the identifiers for the components. As a result it is essential that the user interface editors used by the Tekla technology provide efficient means to find the correct identifiers.

Chapter 6

Improving Localization of the Applications

There is a lot of room for improvement in the field of the application internationalization and localization. In this chapter there is an overview of some ideas and improvements, which have been proposed by other people. Some of the ideas have been applied in practice but the majority does not have implementations that would be publicly available. The purpose of this chapter is to be informational as the content is generally not related to the area of the thesis.

6.1 Culture Repositories

The software developers have often difficulties to find information about the properties of different cultures. Either the information does not exist or it is scattered, inconsistent and difficult to find. The developers are unable to use this information, because strict deadlines do not allow them to concentrate on cultural requirements. To support their internationalization work, it has been suggested that all culture-specific information should be stored into a central repository [20, 21].

The repository contains culture information that is stored in an uniform manner for all cultures. The same cultural properties and their implication to the software development are described for every culture in the repository as that makes it easier to compare the cultures with each other. The consistently presented in-

formation is easy to use by the developers.

In order to improve the accessibility of the information even more, the cultural properties need to be easily browsable. The developers must be able to view the properties of single culture as well as they should be able to choose some properties and compare their cultural variations. This could be accomplished by implementing the culture repository as a web service. In addition to pure information, the culture repository can also contain culture-specific resources such as pictures, symbols and other visual data [14].

The repositories can be organized hierarchically as that will help to identify the common aspects and differencies in global context. At the top level the cultures can be grouped by continent [21] or according to cultural history. Each level of the hierarchical model will contain only the information that is common to all underlying cultures. The hierarchy eases the management of the culture information and the developers can utilize the hierarchy when they implement a support for the targeted cultures in their program code.

6.2 Integration of the Cultural Aspects

Generally the user interface components and development environments have limited support for cultural aspects and the user interfaces have very static nature. Individual components may have some cultural properties that can be altered, but there are no means to control properly bigger schemes. Even if the user interface is resizeable and it can be flipped horizontally, the layout is still fixed in the sense of the component positioning in relation to each other. Sometimes it would be necessary to completely rearrange the layout of some components in order to make the layout better suitable for the target culture.

One proposed possibility is to implement the cultural features as customizable options [21]. A profile of default settings is provided for each target culture and the users are free to adjust the settings further to suit their needs. These settings do not only cover the typical internationalization features such as formatting and text direction. The settings may substitute user interface components or even whole dialogs with others.

Another similar alternative is to make the components culture-aware and let

them change their appearance and behaviour according to the current culture [15]. The main difference when compared to the previous approach is the placement of the culture sensitive logic. Here the logic is embodied in the component while the previous approach has the logic implemented outside of the component and the components are interchangeable.

The culture-aware components are easier to use with the current user interface design tools, because they can be used like any other user interface component. The interchangeable components do not have equal support and they may be harder to use with visual user interface editors. Still, the interchangeable components are more simple to implement and they are more flexible in use, because they are specialized to one task only.

6.3 Referring Resources from the Source Code

Normally applications need to output various strings from the source code and these strings must be queried from the localization database. Usually the resources are got from the localization database by using a string identifier directly in the source code even though it is well known that it makes the code fragile. If the resource database supports different value types, the developer must also cast the resources to correct type. The compiler cannot know if the identifier string is misspelled or if the returned resource is casted to incorrect type so these errors will be detected only at runtime.

The string literals should be defined as constants and the resources should be referenced with the constant value. This limits the number of possible runtime errors to single constant definition no matter how many times the resource is referenced. However, the writing of constant definitions is still error-prone and there will still be undetected errors if the resource identifiers are renamed without renaming the corresponding constant values. Neither does this resolve the casting issue.

In order to solve the casting of the resource values, the constant definitions can be replaced with properties that query the database and return a casted value. Now both identifier string and resource type are defined in a single place and the possible misspellings and casting errors elsewhere in the code are detected by

the compiler. Regrettably the maintenance of the properties is still tedious and susceptible to mistakes.

The .NET framework has solved this problem with strongly-typed resources as explained in Section 5.3. In short, the strongly-typed resources are a generated class that contains the resource identifiers as properties and the string identifiers are hidden in the property implementation. Because the identifiers are used as property names, they must conform the naming rules of the properties. This must be taken into account when the naming policy for the resources is defined.

As the resource class is generated, it is immune to human errors and so the chain from the resource file to resource references in the source code is completely verifiable by the compiler. The only remaining possibility for undetected errors is to build the application with one resource file and then deploy it with another. Other possibilities for mistakes have been eliminated.

6.4 Automatically Identifying Localizable Content

It has been suggested that the development platform should be able to distinguish automatically the functionality and the localizable resources [14]. This would ensure that all localizable content would be included in the resource file and the developer could not accidentally forget any localizable resources. This would also make the internationalization process less laborious.

In the Windows Forms the development environment uses attributes to identify the localizable properties of the user interface components. The development environment knows that those properties must be treated differently when the user interface localization support is enabled. As the localizability is specified with attributes, the localizable properties of the component are decided by the component developer and there is no way to customize the behaviour afterwards. This inflexibility may be a problem if the preselected properties are not suitable for the localization needs of the company.

Pattern based source code analysis can be used to find the localizable resources in order to make them localizable. The localizable resources are detected with patterns and then the resources are moved into localization database and the old resources in the source code are replaced with a code that gets the values from the

database. With this method it is possible to make an existing system localizable quickly and inexpensively [11]. It does not address the typical internationalization issues though, so the result is mediocre at best if the method is applied to software that has not been designed with internationalization in mind.

There are additional benefits when the development environment is able to identify the localizable content as it opens possibilities to new helpful features. For instance, the content can be automatically annotated with an additional metainformation such as the type of the component and the component location in the user interface hierarchy. This metainformation helps to improve the quality of the localizations and, when there are several target languages, it produces cost saves by reducing the need for recurring test and correction cycles [17].

Chapter 7

Problem Domain

Tekla has developed user interfaces mainly with A-kit, which is their in-house developed cross-platform user interface library.[1] Nowadays the applications are primarily made for the Microsoft Windows and, when Tekla adopted the .NET technology, there were no cross-platform requirements that would have prevented the use of Windows Forms technology. Soon both old and new technology were used in parallel.

Tekla's products are sold in the global markets, so the globalization is a vital part of the development. The Windows Forms is not compatible with the old localization technology, so it is necessary to investigate how the support for the old technology could be appended to the existing system. In this chapter the problem domain is outlined by defining initial requirements and describing different component types that need to be supported to fulfill the requirements. Then three alternative solutions are introduced and finally one of them is chosen to be implemented. The implementation is described in the next chapter.

7.1 Requirements

The requirements can be divided into two main categories. In the first category are the requirements dictated by the existing Tekla technology. The second category

[1] More information about the A-kit and other related Tekla's technologies can be found in Section 5.4.

contains features that will make the system more user-friendly and reduce the possibilities for errors.

7.1.1 Compatibility with Tekla Technology

The existing Tekla technology defines localization resources with AIL files, which were covered in more detail in Section 5.4. The AIL files identify culture-specific values with arbitrary names and the only supported value type is a string. The implementation of the Windows Forms localization is needed to be compatible with this technology. Because the default Windows Forms localization uses component names as identifiers, it is incompatible with the requirements.

The .NET implementation of the A-kit resource files does not support the old AIL and AID files with multiple character encodings. Instead the only supported encoding is UTF-8. The UTF-8 is reasonable choice, because it is compatible with old null-terminated strings and it can encode all Unicode character. Thereupon the multi-language AIL files are easier to handle as the strings use only single encoding. The upcoming A-kit release will also have support for the UTF-8 encoding in the resource files.

The old technology sets two constraints for the implementation: all localized resources must be strings and the values must be identified with arbitrary names. The changes in the A-kit resource file format do not affect the localization implementation, because the resource files are accessed through a resource library. At first the .NET implementation of the resource library supports only the AIL files, but the support for the AID files is possible to add later.

7.1.2 Usability Requirements

The development environment should help the developer to set valid identifiers to the components in order to prevent accidental misspellings. The .NET framework has value editors that can be either dialogs or drop-down controls, which are instantiated when the property value is edited in the visual designer. The identifier properties should have associated value editors that display a pickable list of the existing identifiers.

Support for strongly-typed resources is another requirement, as it allows the resources to be referred from the source code with compile-time checked names. The system should be able to generate the resource class that contains all the defined identifiers. The benefits of the strongly-typed resources were discussed in Section 6.3.

7.2 Localizable Component Types

The requirements are only the first half of the problem domain. Nowadays the user interfaces have many conceptually different component types and it is important to understand the differencies of these types in order to be able to support them all properly. In the Windows Forms there are four basic cases that must be investigated and their properties are discussed in this section. This information is used in the following section where the possible solutions are introduced and evaluated.

7.2.1 Independent Components

In Windows Forms terminology there are both components and controls. The components are elements that do not directly appear on the dialogs, but they provide some useful functionality to the application. For example, timer component provides a mechanism to call a method at specified intervals. The controls are visual user interface elements such as check boxes, text boxes, labels and buttons. The controls are also components, because they extend the component base class. All components receive a unique name when they are added into the project. The user interface designer uses the component name as a field name when it generates the code and the component instance is stored into the generated field.

The controls form a hierarchy and each control has a Controls property that returns a collection of the child controls. With this property it is possible to traverse control hierarchy at runtime. The components, however, are not normally exposed for public access. Albeit the runtime access is possible with some nasty reflection tricks, there is a better alternative at the design-time. The services of the integrated development environment give access to all component instances.

The independent components and controls are easy to extend with new properties. The .NET framework defines an interface IExtenderProvider that allows components to provide properties for other components. Normally the development environment shows the properties of the components and controls in a property grid, which allows to view and edit the property values of the selected object. The extended properties are automatically shown in the property grid. In Section 7.2.4 the extenders are discussed more as they pose a special case for component localization.

7.2.2 Components with Child Components

Not all components are independent from others. Some components have child components, which partially define appearance and functionality of the parent component. For example, a ListView control has a collection of ColumnHeader components that specify the column properties such as column name and content alignment. Because the columns are components, their instances are stored into the fields of the class and they can be accessed directly from the user code.

The child components are not as easy to extend as the normal components. The child component collection is usually edited with a collection-specific editor that may ignore the extended properties. As a result the extended properties are shown in the property grid if the component instance is selected directly, but the extended properties are ignored if the component is edited through the collection editor of the parent component. Even though this behaviour could be described as a bug, it must be circumvented, because it exists in the official framework components.

7.2.3 Components with Child Objects

When the component has child objects, which are not components, there will be some additional issues when compared to component childs. The components have always an unique name and they are stored into class fields, but the same does not apply to the the non-component objects. They may not have any useful property for the identification and they are likely to be defined as local variables in

the generated code. The object instances must either be captured in the generated code or they must be accessed through the parent component.

The TreeView is typical control in this category. The tree node hierarchy is made of TreeNode objects, which are not components. The tree nodes are an example of the worst case, because the TreeNode class does not have any good property to be used as identifier. There are Name and Text properties but neither is guaranteed to be unique.

7.2.4 Component Extenders

One important property in the .NET is the possibility to extend the existing user interface components with new properties at design-time. The extenders are implemented as components and they provide functionality that is common to many different components. A good example is a tooltip component that extends all controls with a tooltip text property. At runtime the tooltip component is responsible for showing the set tooltip values when the mouse hovers over the control for certain time.

Error provider component is another good example of the shared functionality. It provides a simple mechanism for indicating to the end user that there is an error associated with a control in the dialog. In case of an error the component shows an optionally flashing icon next to the control. The reason for the error is shown in a tooltip when the mouse hovers over the icon. The error provider has a method that sets the error description string and shows the error icon for the specified control. Normally this method is called when the value of the control is validated.

The main idea behind the component extenders is to separate the common functionality of the user interface elements into reusable components. The separation allows the extenders to have common setting properties that specify how the extender should work with all extended components. The tooltip extender has properties to control visual style and appearance delay. The error provider has properties to set the error icon and its blink rate.

The values of the extended properties are stored by the extender component. Usually the extender puts the values into a hash table that is indexed with the instances of the extended components. The designer generates automatically the

necessary code to preserve the values. In order to provide localization support for the extended properties, one must be able to identify both the extender and the extended component. There is no direct support in the framework to extend the component extenders.

7.3 Possible Solutions

In order to be compatible with the Tekla technology, the Windows Forms localization support in the visual designer must be extended. The localization engine needs to be able to use localization data from the AIL files and the resources must be identified with arbitrary names. The goal is to enable the support of the Tekla technology in all component types that are described in Section 7.2. In this section alternative extension mechanisms are evaluated.

7.3.1 Custom Resource Manager

The Windows Forms uses normally .resx resources when dialogs are localized. The development environment generates necessary code to get the resource values by using concatenated component and property names as a key. The resource values are got by using a resource manager, which loads the necessary resource files and returns the set values.

By implementing a custom resource manager, it would be possible to intervene the localization process. Instead of defining translations for each culture, the developer would specify the AIL identifiers as the values for the default culture. The custom resource manager would use the original resource manager to get the identifiers from the resource file and then it would find and return the corresponding translations from the AIL file. So the official localization system would be utilized as storage for the identifiers, which are resolved to real translations at runtime.

With this approach all components would be supported in a same way as they are normally supported by the development environment. This does pose a problem though. If a component has a collection of child objects, the whole collection may be serialized and stored into resource file as a single value. Luckily this spe-

cial case is possible to detect. After the object collection is get from the default resource manager, it can be traversed and the localizable properties of each object can be translated.

With the custom resource manager there is no way to provide help to the developer when he sets the identifiers. This makes it harder to reuse the existing translations, because their identifiers must be found by other means. However, it should be possible to report the identifiers that are missing from the AIL file so they can be corrected if they are misspelled. It might also be possible to implement editing support for the translations in the AIL file.

Presumably the biggest problem with the custom resource manager is the integration with the development environment. The visual designer of the Windows Forms does not support custom resource managers. The generated code has always the same hard coded resource manager and there is no easy way to replace it with the custom resource manager. The simplest alternative is to overwrite the default resource manager by using a component with custom serialization code. This is not a robust solution though, because it does only work if the component is first to be serialized.

7.3.2 Extensions with Component Extenders

The component extenders would seem to be a natural way to extend the existing components with new properties. With extenders it is possible to introduce additional properties for the components so the Tekla technology could be supported by adding new identifier properties, which would coexist with the real properties that are supposed to be localized. For example, the Text property of the Button control would have additional TextIdentifier property. At runtime the localization engine would set the value of the Text property according to the current locale and the value of the TextIdentifier property. The properties provided by other extenders can be supported by adding a corresponding localization extender for each existing extender.

Though the idea of the extenders is useful and well reasoned the Windows Forms framework does not support it as well as it could. As it was explained in Section 7.2.4 only components can be extended. However, if the component has

child objects that do not extend the component class, the extenders will not work with them. The .NET Framework 2.0 introduced several components that were not compatible with the exteders anymore. The new components are improved replacements for some existing components.

Another problem with the extenders is their static nature. One extender can only provide one set of properties. The extender decides by the component type if the component is supported or not and all properties of the extender are applied if the extender chooses to extend the component. In order to extend the components for the localization purposes one would need to implement one extender for each different component type, because different components have different properties that need to be localized.

Third problem is being caused by the editors of the component collections. If the property value of the component is a collection of components, it is edited with a value editor of the collection. The value editor may not have support for the component extenders and, as a result, the components in the collection do not have the extended properties when they are shown in the value editor. This is a big flaw, because the editing of the child components is almost always done in the value editor.

All in all, the component extenders cannot support non-component types and the design-time support of the child components is only partial. Because of these deficiencies the component extenders do not provide a proper solution for this problem. The component extenders could be a real alternative if they were able to support all component types and the value editors were aware of the extenders. In addition, it would be beneficial if the component extenders would have more dynamic nature so that they could provide a selection of properties based on the extended type.

7.3.3 Extensions with Metadata Substitution

As the component extenders seem to be insufficient to extend the existing user interface components, the logical next step is to use a lower-level extension mechanism that applies to all object types. In the .NET framework static class information can be viewed with reflection. However, there is a runtime layer on top of

the reflection and the layer enhances the capabilities of the reflection. This layer is accessed through TypeDescriptor class and the runtime metadata substitution can be accomplished with it. The only condition is that the system gets the class information with the TypeDescriptor instead of the reflection. Fortunately the TypeDescriptor class is heavily used in the .NET framework and the development environments rely on it when the class information is needed.

The TypeDescriptor can extend both types and distinct instances. The type information is extended by registering a type description provider, which is used by the framework to get the extended type information. The type description providers are chained so several providers can contribute information simultaneously. The TypeDescriptor supports both events and properties but methods are not supported.

The metadata substitution offers interesting possibilities, because it makes possible to extend the existing types with completely new properties. The new properties can also replace the existing properties and the extended properties are fully customizable. The design-time support for identifier selection can be implemented by defining an editor attribute for the extended properties. The editor attribute specifies the editor component that the visual designer uses when the developer chooses to change the property value.

7.4 Summary

The alternatives to implement Tekla compatible localization support for the Windows Forms are evaluated and summarized in Table 7.1 and the used symbols are explained in Table 7.2. If all requirements are to be implemented, only one of the presented solutions may be able to accomplish them. The component extenders lack support for pure object types and some value editors do not support them properly. The custom resource manager does not support custom value editors and its integration with the code generator is questionable. Only metadata substitution is able to provide the extended properties with custom value editors. Still, it is unclear how well the existing Windows Forms components can be extended to enable their localization with arbitrary named identifiers. This will be the main evaluation criteria of the design.

	Supports independent components	Supports components with child components	Supports components with child objects	Supports component extenders	Value editors for extended properties
Custom Resource Manager	++	++	+	++	no
Component Extenders	+++	++	-	+	yes
Metadata Substitution	+++	+++	+	+++	yes

Table 7.1: Evaluation of the alternative solutions to support the Tekla's localization technology in the Windows Forms user interfaces

Symbol	Description
+++	Full support. The object type is supported completely and no additional code is required.
++	Partial support. Some features are missing or the development environment support is imperfect.
+	Minimal support. Only basic functionality is supported or the development environment support is missing. Improvements require substantial amount of additional code.
-	No support. The method is incompatible with the object type.
yes	The feature is available for the supported object types.
no	The feature is not supported by any object type.

Table 7.2: Symbol descriptions for the evaluations of the solutions

The value editors themselves are not very interesting subject, as they can be implemented by following the normal value editor design guidelines. The value editors do not depend on the other design choices. They only need access to the translation database in order to display available localization identifiers. Likewise the class generator of the strongly-typed resources depends only on the translation database. The implementation of the component extensions and the related design choices do not affect these two features in any way.

Chapter 8

Localizer Implementation

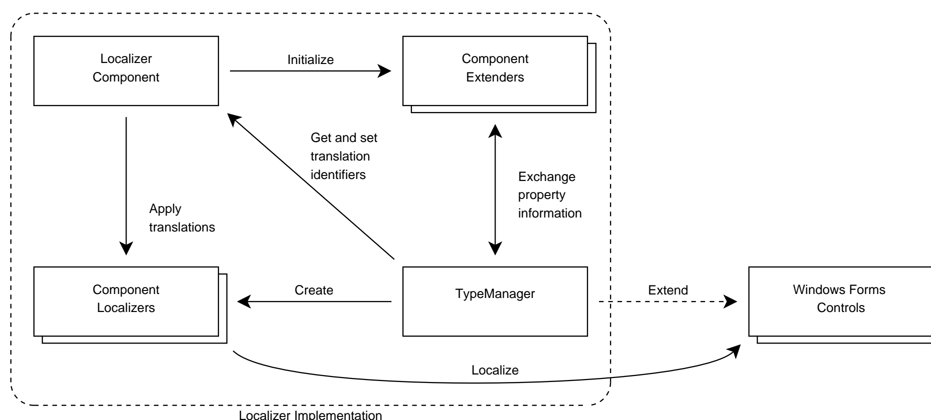

Figure 8.1: Overview of the Localizer implementation

The original localization support of the Visual Studio is not compatible with Tekla technology and so the support for Tekla technology is implemented with a Localizer component that dynamically extends existing user interface objects with new properties. An overview of the implementation is shown in Figure 8.1. The implementation is not Visual Studio specific so it works also with other integrated development environments.

The implementation contains three main parts in addition to the Localizer component: a TypeManager class to extend types with new properties and collections of component extender and component localizer classes. Each component extender class knows how certain component types need to be extended for

Listing 8.1: Serialized values of the Localizer component

```
// Visual Studio does not use using statements when the code is generated,
// but here we use one to improve the readability of the listing.
using ComponentLocalizers = Tekla.Technology.Localizer.ComponentLocalizers;

// Following snippet is generated into InitializeComponent() method of the form.
this.localizer1.Add(new ComponentLocalizers.PropertyLocalizer(this.okButton, "Text"), "but_ok");
this.localizer1.Add(
        new ComponentLocalizers.ListControlLocalizer(this.textStyleComboBox, "Items"),
        new string[] {
            "style_regular",
            "style_bold",
            "style_italic",
            "style_underline"
        });
this.localizer1.Add(new ComponentLocalizers.ToolTipLocalizer(this.toolTip, this.textStyleComboBox), "tip_text_style");
this.localizer1.Add(new ComponentLocalizers.PropertyLocalizer(this.fileMenuItem, "Text"), "menu_file");
this.localizer1.Add(new ComponentLocalizers.PropertyLocalizer(this.quitMenuItem, "Text"), "menu_quit");
```

localization. The TypeManager class is a helper class for the component extender classes to extend existing types with new properties. The extended properties enable developer to edit the translation identifiers of the components. The identifiers are persisted by the Localizer component when it is serialized. The Localizer component is also responsible for localizing the user interface at runtime. The component localizer classes are used to perform the localization of the components.

8.1 Component Localization

Different components have different properties that need to be localized. Simple components need localization typically for one string property, but some components have much more complex requirements like string arrays, enumeration values or image objects. These varying needs of the components mean that the localization system must be flexible and able to adapt itself to different requirements. This is especially true when the translation database does support only string data, but the localizable content includes images and other binary data. Sometimes the data can simply be encoded as string values, but it can as well be referenced with file path or resource name.

Another requirement is to be able to persist the translation identifiers of the

properties. For simple components the identifiers must be associated with the property name and the component instance, but for example the properties of component extenders require that the identifiers are associated also with the component extender instance. In short, different properties require different means in order to be identified uniquely, because the properties are internally different.

The Localizer component uses component localizers to perform the localization in a component specific manner. They are also used as keys when the translation identifiers are stored into identifier collection. The type of the component localizer is dictated by component extenders, as the component extenders specify a component localizer factory for each property that they add with the TypeManager class. The TypeManager uses the factory to produce a component localizer instance, which is used as a key when the property value is either get or set. The component localizer instance is based on a component instance that is given as a parameter to the factory method. In Listing 8.1 the component localizers and their associated translation identifiers are presented in their serialized form.

8.2 Localizer Component

The main part of the implementation is the Localizer component, as it is the center piece that binds everything together. Components are able to add functionality to other components as well as to the user interface design infrastructure. In addition, the development environment stores the state information of the component automatically by serializing it into the generated source code.

8.2.1 Identifier Storage

The Localizer component has a collection of key-value pairs that map the properties of the object instances to arbitrary names used as translation identifiers. The key must be serializable or else it would not be possible to store the identifier collection. Initially the key was defined as a simple struct with fields for the component and property names. There was also a context field, as the two fields are not enough for all identification needs. For example, the properties provided by component extenders cannot be identified with only two fields, because the

extender component itself must also be identified in addition to the extended component. Fourth field defined a handler that is able to localize the component by interpreting the defined fields correctly.

However, the struct keys were abandoned after it was found out that the serializer of the .NET framework is able to serialize all types that can be converted to InstanceDescriptor objects. The instance descriptor provides the information needed to recreate an instance of the object: a constructor and its parameters. In order to convert an instance to instance descriptor object, the type must have a type converter specified with a class attribute.

There are several component localizers and they all are instantiated differently. The associated type converter must know how the component localizer should be instantiated for being able to create a corresponding instance descriptor. Instead of defining a new type converter for each component localizer, all component localizers share a single type converter and implement an interface with which the instance descriptor of the type can be created. When the shared type converter needs to convert a component localizer instance to an instance descriptor, it uses this interface to get the right instance descriptor.

8.2.2 Localization at Runtime

Both runtime and design-time localization of the user interface components is done by the Localizer component. The localization process is very simple as the localization of the components is abstracted by the component localizers. The Localizer component traverses the key-value pairs in the collection of the translation identifiers: the keys are component localizer instances and the values are translation identifiers. For each key-value pair the translation identifier is used to get the translation from the translation database and the component localizer instance is used to set the the translated value into corresponding component.

8.3 Type Extensions

The .NET framework has ability to extend the types at runtime with TypeDescriptor class, which is especially utilized by the design-time infrastructure. The type

extension framework of the TypeDescriptor is unnecessary complex for the needs of the component extenders, so the original framework is hidden with a TypeManager class. The TypeManager provides a simple interface to add and remove properties for existing types.

The new properties are described with property info objects, which contain all necessary information to create a new property. The information includes property name, property type and attributes such as type editor attribute. The type editor attribute defines the type editor that is used when the property value is edited in the visual user interface designer.

The property information contains also a factory instance to produce the component localizer instances, which are used as identifiers for the values of the extended properties. The factory method takes a component instance as parameter and creates a new component localizer instance, which is capable to set the localized value for the component instance.

8.4 System Operation Examined Step by Step

The operation of the Localizer component is most easy to understand by examining how the system really works.

- Localizer component initializes component extenders. Each component extender describes new properties for specific component types. The property description includes name, value type, value editor and key factory. The key factory is used to produce a key that uniquely identifies the component instance and property. All keys are component localizer objects that are able to apply the localized value for the component.
- When a component is edited in the user interface designer, the TypeManager administers that the extended properties of the component are shown in the property grid. The values for the properties are get from the Localizer component.
- When user edits a value of the extended property, TypeManager stores it to the Localizer component as a key-value pair. The key is produced by the key factory of the property.

- To localize the user interface, the Localizer component processes all stored key-value pairs and for each pair it gets the translation of the value and uses the key, which is a component localizer instance, to set the translation.

8.5 Strongly-Typed Resources

In the .NET framework there are two straightforward methods to generate source code for the strongly-typed resource. Both methods use CodeDOM types to describe the source code model. The CodeDOM is a collection of types that represent many common types of source code elements and the CodeDOM object graph can be rendered as source code with a CodeDOM code generator. With the code generators it is possible to generate source code for different languages from single CodeDOM object graph.

The first method is to simply write a code that generates the CodeDOM object graph from the used resource file. However, the .NET framework has already a class that does this for the .resx resource files and the second method is to reuse this class to generate the object graph. This latter method is suitable when IResourceReader interface has been implemented for the used resource file format. Even if the interface has been implemented, the generated object graph requires some modifications before it can be used with other resource file formats.

There is not much difference in the workload of these two methods. The first alternative requires more code to be written, but it is also more easy to adapt for the custom needs. The second alternative is more limited, but there is available a ready-made example of its implementation [27]. When implemented both approaches can be used in a similar manner to generate the source code. In the Localizer component the generation of the strongly-typed resource is done with the second approach.

8.6 Remaining Problems

The Localizer implementation is not entirely trouble-free. There are some problems that could not be solved and in the future there may also be additional com-

patibility issues. In this chapter these problems are described, their severity is estimated and possible solutions are proposed.

8.6.1 Unsupported Component Types

Components with child objects caused some problems. For example, the child nodes of the TreeView control cannot be localized, because the tree node objects do not have any property that could be used as unique identifier. The lack of identifiers would not be a problem if the tree node instances could be tracked, but that is not possible because the value editor of the tree node collection may replace instances without warning.

There are some alternatives to solve this problem. One alternative is to extend the original TreeView and TreeNode classes by inheriting them in order to append the localization support. Presumably the simplest way to add the localization support is to replace the value editor of the TreeView's node collection. This solution would only require implementation of the new collection editor and a small change in the extended TreeView class.

Another alternative is to implement only runtime support for the TreeView class. In practice the TreeView class is extended with a boolean property that specifies whether the TreeView instance should be localized. If the property value is set true, the child nodes of the TreeView are traversed and property values of the nodes are assumed as translation identifiers and they are substituted with respective translations. The tree node objects are not extended with new properties, but the value editor of the localizable properties should be changed to the value editor of the translation identifiers.

The first solution is more laborious as the new value editor requires quite a bit coding. Still, it provides more complete support than the second alternative, which does not provide support for design-time localization. The second alternative is basically a kludge that avoids the need to track the instances of the tree nodes by storing the translation identifiers into values of the localizable properties.

8.6.2 Compatibility in the Future

The implementation of the Localizer component is now strongly dependent on the type extension support of the TypeDescriptor class. The TypeDescriptor works only if the integrated development environment supports it. There are no guarantees that this mechanism is supported in the future versions of the development environments. It is important to evaluate the possible consequences that will follow if the current implementation does not work in the future.

The old translation identifiers are stored through normal component serialization so they are not affected by the extension mechanism. Even if the extensions would not work, the old data in the project continues to be loaded and saved correctly. However, the old translation identifiers cannot be edited and new identifiers cannot be added through the visual designer. They must be accessed through the generated source code if they have to be changed.

The extension mechanism does neither affect to the user interface localization at runtime. The component localizers use only the normal runtime information of the components, which is provided by the reflection feature of the .NET framework. As a result no data is lost and the existing localizations will work even if the extension mechanism stops working with some development environments.

8.7 Summary

The implemented dynamic component extension mechanism appears to be a working concept. It is possible to implement completely new localization framework by appending new properties to existing components at design-time. The implementation supports arbitrary named identifiers, which are used in the existing Tekla technology, and the extended properties are edited with custom value editors, which help user interface developer to assign properties with existing identifiers. Other new functionality such as pseudo-localization, described in Section 3.3, can also be implemented and it does help the developer to ensure that the user interface layout is compatible with long strings.

Still, this solution is not perfect as the components with child objects could not be supported properly. This problem case was described in Section 7.2.3. The

main problem with these child objects is the lack of means to track the object instances. The objects do not have any property that could be used as unique identifier and the value editors do not preserve the instances when the object collections are edited. Consequently these objects cannot be localized.

Even though the components with child objects are an unresolved issue, it does not completely prevent the use of this localization implementation. There are only a few components that are completely or partly unsupported because of this limitation and the majority of the Windows Forms components are fully supported. Most notable unsupported component is the TreeView control and some features of the ListView control are also unsupported. If the functionality of the unsupported components is needed, the components have to be reimplemented. At least the collection editors for the localizable objects must be replaced with new editors that enable the tracking of the object instances.

Chapter 9

Conclusions

Windows Forms localization support was incompatible with Tekla's localization technology and so a new localization support was implemented by dynamically extending objects at design-time. The used technique requires that either the object instances are trackable or the objects have some property, which can be used as unique identifier. In practice, the technique worked for most of the user interface elements, but some objects could not be supported properly as they did not fulfill the requirements. There are some alternatives with which the unsupported objects can have at least partial localization support, but these methods must be considered separately for each unsupported object type.

The presented object extension method is not Windows Forms specific and it can be applied to other user interface component libraries as well. Neither is the usage of the method limited to the localization, because the same approach can be used to implement also other kind of functionality. Despite the problems that were encountered, the dynamic design-time object extensions have much potentiality as they fulfilled almost all requirements.

In a perfect world there would not have been any need for the presented solution in the first place. The user interface libraries should provide means to extend and override the default localization system, including the design-time support. *The pattern for flexible and extendable localization system could be a subject for further research.*

Bibliography

[1] GNU gettext manual. Available from World Wide Web: http://www.gnu.org/software/gettext/manual/gettext.html [cited 2010-05-10].

[2] Java[TM] platform, standard edition 6 API specification. Available from World Wide Web: http://java.sun.com/javase/6/docs/api/ [cited 2010-05-10].

[3] W3C: Internationalization. Available from World Wide Web: http://www.w3.org/standards/webdesign/i18n [cited 2010-05-10].

[4] *R-kit Reference Guide, Version 1.20*. Tekla Corporation, 2006.

[5] *Tekla Technology Product Description, Version 1.20*. Tekla Corporation, 2006.

[6] *A-kit Reference Guide, Version 4.65*. Tekla Corporation, October 2009.

[7] *AilEd User's Guide, Version 2.0*. Tekla Corporation, 2009.

[8] *The Unicode Standard, Version 5.2*. Unicode Consortium, California, December 2009. Available from World Wide Web: http://www.unicode.org/versions/Unicode5.2.0/ [cited 2010-05-10].

[9] Joseph Albahari and Ben Albahari. *C# 3.0 in a Nutshell*. O'Reilly Media, Inc., California, third edition, September 2007.

[10] Pierre Cadieux and Bert Esselink. Gilt: Globalization, internationalization, localization, translation. *The Globalization Insider*, 1(5):1–5, March

2002. Available from World Wide Web: `http://www.lisa.org/globalizationinsider/2002/03/gilt_globalizat.html` [cited 2010-05-10].

[11] Jesús Cardeñosa, Carolina Gallardo, and Álvaro Martín. Internationalization and localization after system development: a practical case. 4th International Conference on Information Research, Applications and Education (ITECH2006), 2006.

[12] Jim Compton. Pseudo-localization techniques, December 2009. Available from World Wide Web: `http://blog.lionbridge.com/translation-and-localization/bid/22953/Pseudo-localization-Techniques` [cited 2010-05-10].

[13] Bert Esselink. *A Practical Guide to Localization*. John Benjamins Publishing Company, Amsterdam/Philadelphia, fourth edition, 2000.

[14] Steffen Gross. Internationalization and localization of software. Master's thesis, Eastern Michigan University, Ypsilanti, Michigan, June 2006. Available from World Wide Web: `http://www.emich.edu/compsci/projects/steffen_gross_thesis.pdf` [cited 2010-05-10].

[15] James Hogan, Christopher Ho-Stuart, and Binh Pham. Key challenges in software internationalisation. In *ACSW Frontiers 2004 - Proceedings of the Australasian Workshop on Software Internationalisation (AWSI2004)*, Dunedin, New Zealand, January 2004. Available from World Wide Web: `http://www.acs.org.au/documents/public/crpit/CRPITV32Hogan.pdf` [cited 2010-05-10].

[16] Dr. International. *Developing International Software*. Microsoft Press, Washington, second edition, 2002.

[17] Tatjana Jevsikova. *Internet Software Localization*. Summary of PhD dissertation, Bytautas Magnus University, Vilnius, Lithuania, 2009.

[18] Yves Lang. Using symbols and icons in localization, November 2005. Available from World Wide Web: `http://www.translate.com/`

Language_Tech_Center/Articles/Using_Symbols_and_Icons_in_Localization.aspx [cited 2010-05-10].

[19] Arle Lommel. *The Globalization Industry Primer*. The Localization Industry Standards Association, Switzerland, January 2007. Available from World Wide Web: http://www.acclaro.com/assets/files/downloads/whitepapers/lisa_globalization_primer.pdf [cited 2010-05-10].

[20] M. J. Mahemoff and L. J. Johnston. Software internationalisation: Implications for requirements engineering. In *Proceedings of the Third Australian Workshop on Requirements Engineering*, pages 83–90, Geelong, Australia, October 1998. Deaking University. Available from World Wide Web: http://mahemoff.com/paper/reqsi18n.ps [cited 2010-05-10].

[21] M. J. Mahemoff and L. J. Johnston. The planet pattern language for software internationalisation. In *Pattern Languages of Program Design (PLOP) 1999*, Monticello, Illinois, September 1999. Available from World Wide Web: http://www.hillside.net/plop/plop99/proceedings/Mahemoff/planet.pdf [cited 2010-05-10].

[22] Microsoft Corporation. *Globalization Step-by-Step: Localizability Testing*, 2010. Available from World Wide Web: http://msdn.microsoft.com/en-us/goglobal/bb688150.aspx [cited 2010-05-10].

[23] Microsoft Corporation. *MSDN: User Interface Issues*, 2010. Available from World Wide Web: http://msdn.microsoft.com/en-us/library/aa291877%28VS.71%29.aspx [cited 2010-05-10].

[24] Yukiko Nishimura. Linguistic innovations and interactional features of casual online communication in japanese. *Journal of Computer-Mediated Communication*, 9(1), November 2003. Available from World Wide Web: http://jcmc.indiana.edu/vol9/issue1/nishimura.html [cited 2010-05-10].

[25] John O'Conner. Java internationalization: Localization with resourcebundles, 1998. Available from World Wide Web: `http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/` [cited 2010-05-10].

[26] Phebe Packer and Lelanie Hellmer. A beginner's guide to managing a localization project, 1998. Available from World Wide Web: `http://www.stc.org/confproceed/1998/PDFs/00017.PDF` [cited 2010-05-10].

[27] Guy Smith-Ferrier. *NET Internationalization: The Developer's Guide to Building Global Windows and Web Applications*. Addison Wesley Professional, Indiana, August 2006.