Patterns for Internationalization and Cross-Cultural Usability

PHILIPP BACHMANN, iRIX Software Engineering AG

Enabling an application to be used in different cultures can drastically increase its market. The patterns in this catalog lay the foundation to later localize the application. This catalog consists of the following patterns: MULTICODED INSTRUCTION, MESSAGE CATLOG, CULTURALLY AND ENVIRONMENTALLY NEUTRAL PERSISTENCE, CULTURE AND ENVIRONMENT AWARE PERSISTENCE, INPUT EXAMPLE, and HIERARCHY DRIVEN BY TARGET CULTURE. Most of these patterns abstract from the implementation in a concrete programming language. Rather, they suppose proven solutions to higher-level issues. So they primarily address people in an architecture or user experience role.

Another source for patterns on this topic is the “Internationalization” category of the Portland Pattern Repository’s Wiki.

The presentation of the patterns follows the style well known from [Alexander et al. 1977]. In part, these patterns are based upon other patterns. Typographic conventions for references to other patterns are similar to [Alexander et al. 1977]. A Glossary provides thumbnails of many of these patterns.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques—User interfaces; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Version control; D.2.11 [Software Engineering]: Software Architectures—Patterns; H.1.2 [Models and Principles]: User/Machine Systems—Software Psychology; H.5.2 [Information Interfaces and Presentation (e.g. HCI)]: User Interfaces—Ergonomics

General Terms: Design

Additional Key Words and Phrases: Culture, Date, I18N, Internationalization, L10N, Language, Localization, Patterns, Time, User Experience

ACM Reference Format:
Bachmann, Ph. 2013. Patterns for Internationalization and Cross-Cultural Usability in V, N, Article 1 (October 2013), 13 pages.

1. OVERVIEW

Enabling an application to be used in different cultures can drastically increase its market. The patterns in this catalog lay the foundation to later localize the application.

Table I lists the patterns proposed.

Many of these patterns share a common scheme: To adapt an application to more than one culture it needs to abstract from a single culture first. This implies generalizing any code that is somehow culture dependent. This step of building in the capability that an application can be adapted to other cultures at all is referred to as [Internationalization] abbreviated as I18N. If your application has been built from several components, its Internationalization should be partitioned along the boundaries of these components both to allow for piecemeal Localization (see below) and to let each originator independently localize his or her component. Internationalization also implies the usage of a character set that contains the union of all characters valid in the cultures to support. Providing respective encodings is the goal of the Unicode standard, one example encoding from this standard is UTF-8.
The second step after internationalizing the application is the specialization to single cultures. This second step has to be repeated for every culture the application is intended to be rolled out to. This is referred to as Localization, abbreviated as L10N. Localization will best be carried out by people from the respective culture or at least by educated experts in the respective culture.

Another source for patterns on this topic is the “Internationalization” category of the Portland Pattern Repository’s Wiki: [http://www.c2.com/cgi/wiki?CategoryInternationalization](http://www.c2.com/cgi/wiki?CategoryInternationalization). Further sources of information are [International 2002](#) and [int 2003](#).

2. MULTI-CODED INSTRUCTION

A graphical user interface offers a toolset to design interaction with the user. Some is driven by metaphors, some by convention.

Different people need different media to comprehend best—some are more textually oriented, some more graphically. A user interface that presents only text to the user makes interaction hard for those who prefer graphics, an interface that solely consists of graphics is hard to understand for those who prefer textual labels. How to maximize comprehensibility of a graphical user interface?

Information can be represented in different ways, e.g. graphically, textually etc. Each single representation alone might not provide the desired distinctness for a fraction of the audience: 8% of the male (female to a much lesser extent) are color blind in one way or the other, so color alone will not work, at least 77% of the world
population have no command of English [Mydans 2007], so text must either be localized or supplemented with another representation of the same information. Even people who are able to interpret two specific representations have a penchant for one over the other. As an application designer, you have to take into account all of these users, because it does not make sense to limit the reach of the application by stratifying the users into groups of different preferences and decide to support only one of them, because this stratification is likely rather artificial to the functionality of the application.

Pictograms annotated with tooltips still give a clear preference to those users that are good at memorizing the pictograms and acting accordingly. The other group of users has to move the mouse to get the same information from the tooltips the other group gets at a glance.

Therefore:

Use two representations for every widget that will be perceived by the same kind of sensory organ of the user, e.g. a pictogram and a textual label. Make both visible at the same time. To further increase comprehensibility base both codings on a respective standard or de facto standard. You can go even further and ensure that the cultures the two codings come from are not completely congruent, so the reach of the application extends to people who understand only one of the two.

If the information will be represented in two different ways that are to be perceived by the same sense this is called a multi-coded presentation—if both representations are to be perceived by different senses, this is called a multimodal presentation. Both multi-coded and multimodal presentations intend to embrace different learning styles, in this pattern especially e.g. regarding the model [Fleming and Mills 1992] the Visual and Read / Write “perceptual modes”. This pattern proposes multi-coded instruction and not multimodal instruction, because the latter can easily cause overload and interferences of human perception finally resulting in distracting split attention, if the signals that stimulate different senses have not been carefully coordinated and synchronized. [Krummeck 2008, p 102] Therefore multimodal are much more controversial than multi-coded presentations.

Traffic signs are a good example for carefully designed multi-coded symbols: One coding is given by color (Portland Orange and white with U.S. pedestrian traffic lights or red for prohibition signs and blue—not green—for mandatory-signs in those countries that ratified the Vienna Convention on Road Signs and Signals), the other one by shape.

With Graphical User Interfaces, in the former OPEN LOOK interface by Sun Microsystems and AT&T Corporation iconified applications always have born both a pictogram and a textual label, see e.g. Figure 1a. Not all graphical user interfaces have followed this tradition—note that while the Dock in Apple OS X shows pictograms, but labels only in terms of tooltips, the smartphone operating system and interface Apple iOS shows both pictograms and labels permanently, thus follows the OPEN LOOK style again, see Figure 1b.

This pattern has similar implications as the Dual-Coding Theory by Allan Urho Paivio [Paivio 1971], but is more general: It suggests at least two codings and does not restrict one of them to text. Figure 1c shows an x mark and a check mark to indicate “no” and “yes”. Because these shapes have no unique meaning in different cultures, color is used as one orthogonal dimension—in this case, the colors of prohibitive and mandatory traffic signs as standardized by the Vienna Convention on Road Signs and Signals have been used not to place a burden to people with red-green color blindness. Further, the filling is another orthogonal dimension, so in total three codings are used. The filling, however, does not provide any predefined meaning, so its primary purpose is to enhance distinctness especially in black-and-white media. Pattern publications themselves are examples for this pattern: Most are illustrated by means of diagrams or photos to allow for easier access to them.

A limited number of pictograms seems to work even cross-culturally. Among those are the 50 pictograms issued by the U.S. Department of Transportation (DOT), which have been cross-culturally tested; the DOT pictograms are in the public domain. [Cook and Shanosky 1979] Figure 2 shows two examples of DOT pictograms that might be useful in the context of graphical user interfaces. Arrows seem to work cross-culturally, too, if taking the reading direction into account. [Fernandes 1995] pp 43–44]
(a) OPEN LOOK Sun Microsystems DeskSet Mailtool icon showing pictogram and label

(b) Apple iOS Email. The presentation follows the OPEN LOOK tradition.

(c) Rejection and confirmation by character, color, and filling

Fig. 1: Three examples for multi-coding

(a) Mail

(b) Litter Disposal

Fig. 2: Two examples from the set of DOT pictograms

1974, 1979, U.S. Department of Transportation [Cook and Shanosky 1979]

❖ ❖ ❖

If one of the codings is textual, you may want to build in a MESSAGE CATALOG to supply translations to several natural languages. See Section 3 for details.

3. MESSAGE CATALOG

Language is the picture and counterpart of thought.

MARK HOPKINS

Still much of the user interface of an application makes use of natural language. Language is culture dependent.

❖ ❖ ❖

Patterns for Internationalization and Cross-Cultural Usability — Page 4
Applications put a burden on users if the native language of the user is not the language of the user interface. This increases the likelihood of making more mistakes in using the application and might ultimately result in abandoning this application. So what to do about the user interface when rolling out an application to different countries?

It is hard enough to design a user interface to be usable within a single culture. There are nearly 7000 natural languages in the world—it is impossible to support all of them at once. So realistically only a subset will be supported. This subset might change—probably increase—over time. Language is not only vocabulary. Sentence structure also differs in different languages. The best translation is always to and not from his or her native language. It is not enough for an interface to follow a certain natural language; it should also adhere to the conventions set by the body of all applications in that language that predate the respective application. So a translation that is linguistically correct does not necessarily match the interaction expectations of the user he or she has acquired from having used other software—in English, for example, it was quite confusing if a menu entry to terminate an application was labeled “File / Terminate” instead of “File / Exit”.

In general, natural language skills and programming skills are distributed among several persons. The application probably evolves over time, so its interface changes—then what about its several instances in foreign languages? Every non-trivial application is structured in terms of modules or libraries. Some of them probably come from external sources. Those of them that generate output or process user input show through the user interface of the final application. Responsiveness is important for user acceptance. Automatically looking up translations into foreign languages must not drain performance too much.

Therefore:

Substitute a call to a lookup function into a culture-specific translation table for all occurrences of output strings in a natural language. These translation tables are commonly referred to as MESSAGE CATALOGS or Resource Bundles. Keep the substituted string as the key for the respective lookup. This is an instance of [internationalization] as described in Section 1. Then hand over the task of creating concrete instances of these translation tables to specific cultures to native speakers in the respective natural language. This is an example for [Localization] as described in the section referred to. You will end up with one MESSAGE CATALOG per supported natural language and component of the application.

Because text sizes in different translations differ quite a lot, first the application layout needs to be generalized such that it can adapt to different sizes of textual labels afterwards. So instead of specifying positions and sizes of textual interaction elements like buttons in terms of a fixed distance unit the layout should be delegated to a layout manager that arranges them relatively to each other taking their individual sizes into account. Most widget sets provide such functionality; a Java example for a layout manager is java.awt.FlowLayout.

One well established Internationalization library is GNU gettext, which does not interpret the translation tables, but uses tables compiled before to speed up lookup. Gettext can use different tables for different components of the application. Gettext allows Localizations to evolve more or less independently from the internationalized application itself by means of so-called fuzzy entries, thus can be used to relax baselining as explained by pattern COMPONENT BASELINE by Ralph Thim and Lise B. Hvatum[Thim and Hvatum 2012, pp 8–11].

When an application is going to be localized to cultures which share the same natural languages or dialects thereof, it might be a valid simplification to start with not to consider the regional dialects. The most important point here is to decide on a single variety to support—i.e. not to mix up e.g. British and American English. To non-native speakers given the task to localize the application this might pose a challenge. This consistency requirement also includes e.g. typography. When it then turned out that there is a significant market in regions sharing the same natural language, but in different varieties, then it is time to refine the Localizations to consider these varieties to let more users feel really at home.

Be aware that the realm of the natural language might not be congruent with the realm of other aspects that have to be considered when adapting an application to other cultures. All of these aspects are referred to as Cultural Facets besides natural language e.g. a unit system for quantities. Consider e.g. the different medical
units used in different regions where German is spoken: In Northern and Central Switzerland the native language is German or a regional dialect thereof, and as e.g. in former Eastern Germany the medical units are in accordance to the International System of Units (SI), e.g. mmol l$^{-1}$ for blood sugar. In (former Western) Germany, however, many clinical laboratories and devices use units that predate the SI system, e.g. mg dl$^{-1}$.

When implementing MESSAGE CATALOG it has to be somehow decided how the application should figure out the culture the user belongs to. This is not trivial, so it pays off to negotiate this topic with user experience experts. The most automated way is probably to use the cultural settings of the operating system, which in turn are bound to the currently logged in user.

Also processing of local time, numbers etc. are affected as well. Some of these cases can be treated using the built-in facilities of the respective standard library, e.g. stream facets in C++ to change formatting of numbers, dates etc. and date / time classes or functions for dealing with different time zones. Quantities measured in a certain culture-specific unit, currencies etc. are also affected. For quantities generalization implies multiplying with a culture-specific factor and exchanging the respective unit accordingly. This is further elaborated on in patterns CULTURALLY AND ENVIRONMENTALLY NEUTRAL PERSISTENCE (see Section 4) and CULTURE AND ENVIRONMENT AWARE PERSISTENCE (see Section 5).

4. CULTURALLY AND ENVIRONMENTALLY NEUTRAL PERSISTENCE

Data is being passed around the world. Client / Server applications may have clients localized by means of MESSAGE CATALOG (see Section 3) and central servers with data repositories that span all cultures of the clients. The units of the quantities stored in these repositories differ from culture to culture. It is not important to keep track of the unit a quantity has originally been entered in. The losses in precision multiple conversions back and forth may yield are tolerable.

Say, data from different cultures is not just stored in a central place, but also aggregated, e.g. compared with each other or summed up. Facing the cultural differences: How should the data from different cultures be persisted?

Travellers know that date and time are different around the globe; these differences must be taken care of not only on a global scale, but already within large countries like the U.S. and even within some single states: The U.S. state Indiana, for example, mainly observes Eastern Time (UTC−5), but shares the Chicago, IL, metropolitan area in its northwest with Illinois and Wisconsin and the Evansville, IN, metropolitan area in its southwest with Illinois and Kentucky, both where Indiana observes Central Time (UTC−6). Storing quantities in their local unit only works until the need arises to e.g. compare them across cultural boundaries. A date and time stored without reference to its time zone is called local time according to ISO 8601—and is probably the worst way to store it because no judgement can be made about the permissibility of an operation involving a second local time, which may or may not come from the same time zone.

Different from numbers strings are not subject to arithmetical operations. But even those might pose challenges when storing them in a central place. Different character sets make it difficult to exchange them crossing cultural boundaries. Given the strings are intended to be interpreted as e.g. numbers or dates, this interpretation likely does not work from another culture than those that persisted the number or date.

Therefore:

Store such data in a culturally and environmentally neutral way and do the conversion to and from this representation upon storing and reading it.

This is a simple solution, even though it involves two conversions, one on writing, the other one on reading and presenting the data, even if the presentation takes place in the same culture the data was entered in. Note that
conversions involving only integral number representations and addition are lossless back and forth aside from the risk of arithmetic overflow. And even potential losses resulting from conversions of floating point numbers or involving multiplications may happen at a scale that can well be tolerated.

Decide an internally used unit of a quantity first. Always keep in mind that this convention is an implementation detail. In case of date and time use e.g. Coordinated Universal Time (UTC).

The same considerations apply to the way the libraries used adapt to cultural differences. Operating systems and libraries group the aspects that define a culture, the Cultural Facets, e.g. what character is used as a decimal point, which is the currency, etc., into a so-called Locale. Often, an application implicitly depends on a Locale fed into the application by means of an environment variable. While this is quite useful for end-user applications that manage persistence on their own, it is a bad idea for centralized server applications that have to deal with input from multiple cultures and store it. Most libraries apply Localization to serialization and deserialization regardless of whether the sink or source is the screen or a file, i.e. regardless of whether the presentation or persistence layer is affected. So from within the code explicitly set the Locale right after application start to a fixed one or use those serialization and deserialization routines that explicitly take a Locale parameter instead of magically fetching this information from the environment. In case of UNIX Locales prefer the “C” Locale, because it is the only Locale available on every system.

Every modern operating system provides routines in its standard libraries to convert date and time to and from UTC considering Daylight Saving Time, if any, or convert instances of intrinsic data types into strings and back using the formatting rules given by the “C” Locale.

Regarding date and time there is another point to consider: Often, some data is associated to fixed time intervals, e.g. a day. In any other time zone this data will span two days, however. It might make sense to discretize the day to the smallest difference between all time zones in the world, which currently is 15 min. Then for every location in the world it requires the same calculations to display the data relative to the respective local time zone.

For physical quantities a culture-specific table can be maintained that maps the respective unit to the arithmetic operation required to express the same quantity in the standard unit the quantity is stored in internally. Note that depending on the unit a factor is not enough as sometimes the conversion is not multiplicative, but additive—or even logarithmic.

When implementing Culturally and Environmentally Neutral Persistence it has to be somehow decided how the application should figure out the culture the user belongs to. This is not trivial, so it pays off to negotiate this topic with user experience experts. The most automated way is probably to use the cultural settings of the operating system, which in turn are bound to the currently logged in user; in case of date and time in web applications note that HTTP does not transmit the time zone of the client to the server—therefore the only automatic solution is via client-side ECMA Script given you do not intend to personalize the application and bind that information to the user, who would be quite confused then when he or she travels, however. Sometimes, the users might want to have more explicit choice regarding certain Cultural Facets, e.g. they know better than the application designer whether they prefer to see time in their home time zone or always converted to the time zone they are currently in.

This is a similar solution as the one described in the pattern Canonical Data Model [Hohpe and Woolf 2004a]; the context of the pattern referred to is messaging between several enterprise applications, however, whereas Culturally and Environmentally Neutral Persistence affects persistence across multiple cultures within a single application.

\[\text{Note that the constructor of the standard Date class in ECMA Script taking a date string, which runs the same parser as Date.parse() does, exhibits inconsistent behavior in different implementations of ECMA Script, especially among different web browsers. Only a subset of ISO 8601 is supported by most. So it requires either some experimentation to find the intersection of formats accepted by the engines to support or engine specific modification of the input string before creating an instance of Date.} \]
5. CULTURE AND ENVIRONMENT AWARE PERSISTENCE

Data is being passed around the world. Client / Server applications may have clients localized by means of MESSAGE CATALOG (see Section 3) and central servers with data repositories that span all cultures. The units of the quantities stored in these repositories differ from culture to culture. The application is expected to keep track of the unit a quantity has originally been entered in or has to reduce losses in precision due to conversions of a quantity from one unit to another one to the least possible.

❖ ❖ ❖

Say, data from different cultures is not just stored in a central place, but also aggregated, e.g. compared with each other or summed up. Facing the cultural differences: How should the data from different cultures be persisted?

Travellers know that currencies vary in different countries. The same applies to other units as well. Even if the unit does not seem to differ, there might be a “hidden” point the unit actually refers to that differs from country to country: Elevation above sea level in Europe is given in m, but the details differ: Somewhat simplified, the reference might be the Amsterdam Ordnance Datum as e.g. for Germany, the tide gauge at Marseille as e.g. for mainland France and Switzerland, the Sartorio mole in the Port of Trieste as for Austria, but not for Italy, which uses the Genoa tide gauge.

Both money and elevation above sea level are examples for quantities which original units likely are to be kept by the application for retrieval later on.

Therefore:

Make persistence culture and environment aware: Store both the quantity in a format that is local to a culture and also store an unambiguously interpretable reference to that culture in close association with the quantity, i.e. store the amount of the quantity in a culture specific unit and store this unit in a way that is not culture specific.

Never simply store data in a central place in a format that is local to a culture without additionally storing an unambiguously interpretable reference to that cultural differentiator or Cultural Facet—otherwise you would not know later on how to interpret those amounts. The key of the Cultural Facet is the remaining small part of the information that needs to be persisted in a culturally and environmentally neutral way; see Section 4 for details.

In the case of e.g. money upon first entry record both the amount and the currency, the latter probably in terms of the currency code according to ISO 4217, e.g. USD for United States dollar. If this quantity is later requested from within the same culture, simply present the persisted amount and the currency. Otherwise call a conversion routine to express the quantity in terms of the currency requested. When a user changes the amount in his or her local currency different from the original currency, convert the quantity back to the currency the quantity has been created in.

The same considerations apply to the way the libraries used adapt to cultural differences. Operating systems and libraries group the aspects that define a culture, the Cultural Facets—e.g. what character is used as a decimal point, which is the currency, etc.—, into a so-called Locale. If the application needs to store strings entered by the user that contain e.g. numbers, then the key of the Locales valid for this user needs to be stored along with the string. Otherwise it is quite unlikely that the application can interpret the string as a number in the future. Again, the Locales key itself needs to be persisted in a culture and environment neutral way.

When implementing CULTURE AND ENVIRONMENT AWARE PERSISTENCE it has to be somehow decided how the application should figure out the culture the user belongs to. This is not trivial, so it pays off to negotiate this topic with user experience experts. The most automated way is probably to use the cultural settings of the operating system, which in turn are bound to the currently logged in user. The most automated way is probably to use the cultural settings of the operating system, which in turn are bound to the currently logged in user. Sometimes, however, the users might want to have more explicit choice regarding certain Cultural Facets, e.g.
they know better than the application designer whether they prefer monetary values in the their (probably foreign) currency or always converted to their own currency.

6. INPUT EXAMPLE

Computer applications are not television sets—output is one thing, but at some point the user will input some data thus closing the interaction cycle.

❖ ❖ ❖

Sometimes a single text field allows for a more efficient data entry than a set of fields or special widget even if the data to be entered is expected to have a certain kind of structure, because the user can stick with her or his keyboard and does not have to alternate between keyboard and mouse. If data can be entered by means of a widget specially designed for data structured this way, e.g. a calendar widget to enter dates, the user experience expert might also provide a text field to allow an alternative way for the user to enter the data—this would also enable him or her to copy and paste all data at once. How does the user know the expected format of the data to enter that second way?

Many users have a history with computers. They have experienced a multitude of design decisions: Sometimes they have to always adapt to the U.S. way to format numbers and dates, sometimes formatting and even programming languages have been localized, e.g. Microsoft Visual Basic for Applications for Excel 95. So they are uncertain of what format a system expects as input. The conventions differ even between countries with the same natural language.

Therefore:

For each text field provide example input. This can be done by means of a default entry, if there exists some plausible value at all, or by text in a dedicated column in close proximity to the respective text field. If the anticipated screen size allows so, do not use tooltips but text permanently visible.

If examples are presented separately from the field, i.e. not as default entries, then present them to the user in a way he or she can copy them from the example column and paste them into the respective text field.

Interpreting user input is always harder than outputting some value, because it is important to allow for some variability in formatting by the user as long as his or her input is not ambiguous.

Besides number and date formats another important area of application of this pattern are conversion factors, where the user might not instantly know whether to type in a certain number or its reciprocal or negative counterpart. Examples include currency exchange rates and conversion factors for different systems of physical units. Errors in entry of such data can cause severe failures—and are hard to detect because such not round numbers are suggestive of precision, and precision has persuasive power, be the numbers correct or not.

Microsoft Office 2010 Outlook provides two ways to enter a calendar date: One via the text field printed on top of this pattern and the other one by means of the calendar representation shown in Figure 3 which follows the CALENDAR PICKER pattern described in [van 2009]. The text field shows today's date as the default and can be edited by the user providing an alternative way to change the date to using the CALENDAR PICKER.

7. HIERARCHY DRIVEN BY TARGET CULTURE

Data that consist of distinct information like parts of names, phone numbers, geographical location etc. is entered and stored in a fielded form to let the computer more easily grasp that structure than when letting the user enter it as unstructured full text.

❖ ❖ ❖
Applications often force the user to enter data with an implicit structure in a structured way because they store them preserving structure. Sometimes the structure is specific to the culture of the content, however. Telephone numbering plans differ around the world, postal addressing schemes also do because postal systems differ. The quality of data will suffer for the majority of cultures that do not share the way to structure the data with the culture the application was originally developed for. How to design data entry to ensure a high quality of data entered for all cultures anticipated?

Figure 4 shows the dialog Microsoft Office 2010 Outlook presents on entering a new address of a contact. Even though it is open to more than one culture by mentioning more than one term in three of its five labels, both the choice of fields and their order clearly stresses its origin. For addresses in Germany, for example, apart from domains where state legislation is important like education and construction it is quite unusual to maintain the state in address records, and the city comes after the ZIP code.

Figure 5 shows the dialog Microsoft Office 2010 Outlook presents on entering a new phone number of a contact. For Germany extensions do not need to be treated in a special way. In Switzerland there are no extensions at all, and because since 2002 full-number dialing has been implemented and phone numbers are portable throughout the country, there is no real need to distinguish between a city code and a local number any more there.

Therefore:
Present different forms to the user depending on the culture of the data to be entered and store them accordingly. Because there are many cultures you cannot support in the first place, provide a default to handle even them. Later you can substitute specific implementations for the default piece by piece.

Apply this pattern recursively if you also want to precisely model subcultures, e.g. in the case of small villages where houses are uniquely identified just by a number you can drop the street name field given there has been a differentiation between street name and house number.

In the case of addresses the choice of a key could be e.g. the ISO 3166-1 alpha-2 country code or the name of the country in either French, the official language of the Universal Postal Union (UPU), or English, the other language accepted by the UPU. In case of phone numbers taking the country calling code as a key is a natural choice.

Implementation-wise, the most obvious way were switch statements. More advanced implementations probably will combine the design patterns [FACTORY METHOD][Gamma et al. 1996c] and [BRIDGE][Gamma et al. 1996a] instead. In F# and Scala give the programmer discriminated unions to express this idea in a even conciser way, taking the target culture as the tag.

Using this pattern has the liability that search cannot be implemented in a unified way any more. One refinement therefore is to consider not just the input, but also formatted output and search as being culture specific.


APPENDIX

REFERENCES


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1996b. Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, deutsche Übersetzung von DIRK RIEHLE (dritter, unveränderter nachdruck ed.). Addison-Wesley-Longman, Bonn · Reading,
This section contains pattern-thumbnails and definitions of the most important terms used in this paper.

**Bridge.** The **Bridge** pattern separates an abstraction from its implementation. Behind the same abstraction different implementations of its functionality can coexist. Each of them can evolve independently. [Gamma et al. 1996a]

**Component Baseline.** The **Component Baseline** pattern proposes to decouple development teams by splitting a single, monolithic baseline into multiple baselines. The software artifacts of each single team are managed as a separate baseline and are allowed to evolve more independent from one another than when managing the whole application as a single baseline. [Thiim and Hvatum 2012]

**Cultural Facet.** A Cultural Facet denotes a certain aspect that joins a group of people on the one hand and distinguishes this group from other groups on the other. Examples are natural language, system of units for quantities, timezone, currency, number format, numbering scheme for calendar weeks, sorting etc. The extent of a Cultural Facet is subject to definition in many cases; it has to be e.g. decided whether each language dialect should constitute its own Cultural Facet or not. Be aware that the realm of a single Cultural Facet, e.g. a natural language, might not be congruent with the realm of another one, e.g. a system of units for quantities.

**Factory Method.** The **Factory Method** pattern delegates creation of objects to a method. This method encapsulates the knowledge of the exact type of the object to create. One example is a method with a parameter that allows the caller to tell the method which type to instantiate. [Gamma et al. 1996c]

**Internationalization.** Internationalization refers to the cultural generalization of an application to prepare it for the forthcoming process of Localizations to several cultures. Internationalization is an all-encompassing endeavor that has to consider all Cultural Facets relevant for the respective application. Internationalization is often abbreviated as I18N.

**Locale.** A Locale denotes a set of several Cultural Facets. A Locale is specific to a certain region, e.g. a country, and refers to all information to localize text, format numbers, dates, currencies etc. To identify a
Locale, it often has a key unique among all Locales available, e.g. the tuple of a language code according to ISO 639-1 and an ISO 3166-1 alpha-2 country code, e.g. en_GB for British English.

**Localization.** Localization refers to the specialization of an application to a specific culture considering all relevant Cultural Facets. The result is a new or modified Locale. To prepare an application for the task of Localization it must be internationalized first once. Localization is often abbreviated as L10N.

**ACKNOWLEDGMENTS**

Without the invaluable feedback of Jeffrey L. Overbey, who was the PLoP shepherd of this work, this paper would not have been the way it is now.

Last but not least my thanks and love go to Cornelia Kneser, my wife, for her constant support throughout the writing of the paper.

Received May 2013; revised October 2013; accepted August 2013