The Planet Pattern Language for Software Internationalisation

Michael J. Mahemoff
Department of C.S.& S.E.
The University of Melbourne
Parkville, 3052, Australia
m.mahemoff@csse.unimelb.edu.au
Tel: +61 3 9344-9100
Fax: +61 3 9348-1184

Lorraine J. Johnston
School of I.T.
Swinburne University of Technology
Hawthorn, 3122, Australia
ljohnston@swin.edu.au
Tel: +61 3 9214-8742
Fax: +61 3 9214-5501

KEYWORDS Pattern Languages, Design Reuse, Internationalisation, Usability.

0 Planet: An Overview

Software is becoming an increasingly international affair. The internet has brought users closer together, and there is more outsourcing to overseas firms than ever before [16]. Since good software takes into account the characteristics of typical users, internationalisation produces challenges because of user diversity. The Planet pattern language aids in the specification of such systems, with a goal of making them usable and useful to people from different locations and backgrounds.

The overall process for producing software which fits the needs of particular cultures is termed *culturalisation* (after [2]). It is not enough to provide one global version of software with the intention of satisfying everyone. Some software features, such as images and text, must be developed with specific cultures in mind. To this end, the culturalisation process entails two phases. In the *internationalisation* phase (also called *localisation-enabling*), databases and other structures are set up within the core system. These structures are then populated in the *localisation* phase, when programmers, translators, graphic artists, and others decide what is appropriate for a particular user community.

Figure 1 provides an overview of the fourteen patterns and indicates the seven patterns we have detailed in this paper. The remaining patterns are summarised in Section 8. The cycle between `Flexible Function(4)` and `Elastic User-Interface(5)` reflects the idea that functionality and user-interface should be iterated until satisfactory.

You begin applying the language when you have been presented with a project that delivers to multiple cultures, or may do so in the future. In this case, begin with `Export Schedule(1)`. If there are no such projects, but you anticipate they will come later on, you can get started by developing some `Culture Models(2)`.

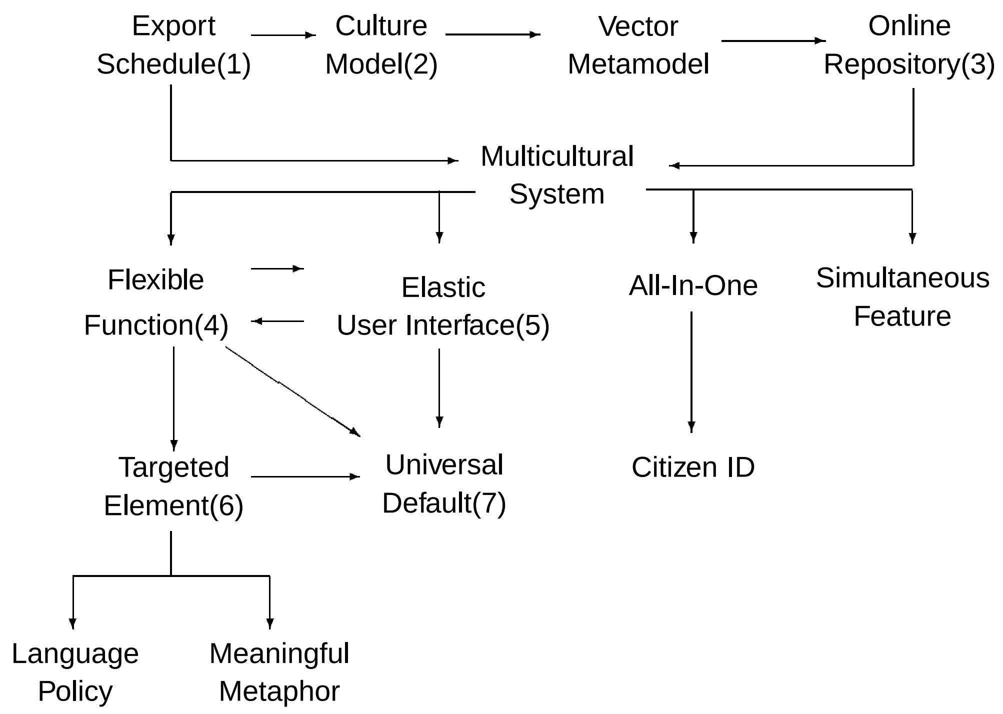

Figure 1: A map of the Planet language. The number after some patterns represents its position in this document. Section 8 outlines the remaining patterns.

1 Export Schedule

Context: You are beginning a project where users will come from a wide range of cultural backgrounds.

Problem: Culturalisation is a resource-intensive exercise.

How can you allocate resource to cultures at a level befitting their individual needs, and at the same time ensure that no more attention than necessary is devoted to any particular culture?

Forces:

- You should know which cultures will use your software, otherwise you cannot provide features which meet their needs.
- It is often too expensive to produce for all audiences at the same time. Sometimes, a local version is prepared first and overseas users are considered later on.
- Introducing a new culture may not be a simple localisation procedure. The new culture may necessitate changes to the internationalisation framework and this can be painful. As one example, different cultures use different rules for sorting words. An English-language implementation can sort using the "<" and ">" operators, but what if a culture with a different alphabet order is introduced? If the comparisons are scattered across the code, then a programmer will have the unenviable task of replacing each one with a generic comparison function.

Solution: Produce a schedule which indicates *when* each target culture will be supported, and *how important* it is to support each target culture. Some cultures will not use the system until version 3.0 is released, but these cultures should still be considered when version 1.0 is developed. The repository can be consulted to determine which factors vary across the supported cultures. The localisation phase can then address this variation.

In most businesses, interaction with the marketing department will be essential. When considering attractiveness of foreign markets, consider size and value of customer demand, competitive importance of the market, and availability of expertise [19].

At this stage, establish the importance each culture places on different aspects of software. For instance, Evers and Day [7] found that Indonesians, compared with Chinese, are more attracted towards usability than certain other qualities. Since you cannot maximise every quality, this kind of information will tell you which areas to focus on for a given culture.

Rationale: Identification of cultures in advance ensures that the functional core and internationalisation framework are flexible enough to make changes relatively straightforward.

Examples: Searching for target markets, choosing how intensive the operation will be, and planning future expansion is a well-established international marketing practice [14].

Uren et al. [23] suggest that marketing staff should develop a list of target countries, even if translation is not immediately required. Luong et al. [15] suggest that developers need to decide between a full localisation and a partial localisation, and also consider whether to ship overseas simultaneously with the local release.

Resulting Context: If your organisation has not already done so, start creating `Culture Models(2)`. You may need to update existing `Culture Models(2)` to reflect any new cultures being considered or to account for aspects of the cultures which are relevant to the new project. Once you have collected `Culture Models(2)` into an `Online Repository(3)`, you will be able to create a `Multicultural System` suitable for cultures mentioned in the schedule.

Note: For the remainder of this paper, the cultures mentioned in the `Export Schedule(1)` are referred to as *Target Cultures*.

2 Culture Model

Context: You are working on software projects with `Export Schedules(1)` identifying which cultures will use the system.

Problem: Like beauty, usability and utility are in the eye of the beholder. You have to understand the user before you can begin working towards these qualities. **How do you support development decisions which depend on information about specific cultures?**

Forces:
- Extensive information may be required to support development decisions, necessitating potentially expensive research activity. This effort will have to be repeated on new projects involving the same cultures unless explicit attempts are made to reuse it.
- It is difficult—sometimes impossible—to reverse-engineer details about cultures just by looking at the resultant product. The Walkman has been successful in America largely due to the fact that people do not want to be disturbed by the outside world, but it was originally conceived by Sony's co-founder to help listeners avoid disturbing others [17].
- Even if you can access process documentation such as meeting minutes, information about cultures will be scattered. This will be difficult to find specific information when it is needed during internationalisation and localisation.

Solution: Construct models of cultures which are relevant to your projects. When you discover new information about a culture, add it to the culture model. To improve opportunities for reuse, these models can be held in common by the entire organisation, rather than pertaining to a specific project. They are an asset which is refined over time, just like a software library or an estimation technique.

The culture model can also include issues regarding the development process. Luong et al. [15] discuss the English-language ability of the quality assurance engineers they deal with in Japan. Another culture-dependent process is user testing; some cultures are more reluctant than others to criticise a product (e.g. [11]).

Rationale: Consistent culture models provide a central point for culture-specific details and also help developers compare and contrast cultures. If a developer notices that all of the cultures for a particular program use Latin characters, then the internationalisation phase need not consider the possibility of non-Latin character sets. Localisation will also be more efficient if similarities between cultures can be exploited.

Examples: There are several well-known culture models in anthropological and marketing literature. In "Understanding Cultural Differences" [10], Hall and Hall explain several culture-dependent variables, e.g. the level of detail people desire when presented with information, attitudes to personal space, whether or not people like to perform tasks in parallel.

Luong et al. [15] provide details on localising for Asia. Numerous business texts describe the intricacies of doing business with, or creating products for, a particular country [12] or region [6].

Resulting Context: Once you've decided to create a culture model, you'll need a way to structure it. A culture model can be conceptualised in different ways; an extremely simple form is just an arbitrarily-ordered collection of relevant documentation gathered from various projects. In this pattern language, we focus on one means of characterising a culture: the `Vector Metamodel` form.

3 Online Repository

Context: You have begun to maintain `Culture Models(2)` according to the same `Vector Metamodel` (i.e. same factors for each culture).

Problem: As you start accumulating `Culture Models(2)`, you will realise the need to organise them together. **How can a collection of culture models be organised in a manner which is useful for software projects?**

Forces:

- Culture Models should be organisation-wide to avoid duplication; it is feasible and desirable to transfer information learnt from one project into other projects.
- Information about cultures should be shared, but is often discovered in physically-distant locations.
- If developers can't access the models quickly and easily, the information will be ignored.
- If developers can't update the models easily, the repository will lose accuracy over time.
- Cultures have relationships with one another. We should be able to capture associations, such as one group being a sub-culture of another group.
- It is useful to look up a specific `Culture Model(2)`, but there are many other ways people might like to access information during the culturalisation process. Depending on the task at hand, developers may wish to explore the information in unanticipated ways, e.g. comparing two cultures, considering a single factor across numerous cultures.

Solution: Create an online repository, accessible to the entire organisation. Compose it of `Culture Models(2)` all based on the same `Vector Metamodel`.

The following guidelines make it easy to *access* information in the repository:

- Provide browsing facilities which present each culture and factor. Let the user select a factor (and show how each culture varies on it), a culture (and show all of its factors), or a combination of both.
- Provide facilities to search the `Culture Models(2)`.
- Link from one model to another if it helps to demonstrate a point of similarity or difference.
- Link to the original artefacts if they are online, or identify the source if they are not.

The following guidelines make it easy to *update* the repository:

- Facilitate discussion among contributors, e.g. via a mailing list or within the repository system.
- For each `Culture Model(2)`, make one or more people responsible for maintaining it.

Large repositories can group `Culture Models(2)` together, e.g. according to continent. This approach can follow the `Composite` pattern [9], e.g. a model of Europe contains its own information and also contains child models (France, Italy, etc.). When a user looks up a model, information about its ancestor models are also shown.

Rationale: Like pattern languages, repositories of this nature help people reuse existing, tested, knowledge. Reuse of proven concepts involving human-computer interaction are particularly helpful, because people's reactions can be difficult to predict. In the case of interaction with international systems, the case for reuse is even stronger, because more work is required to obtain original knowledge (e.g. travelling overseas, establishing partnerships with foreign consultants).

Examples: Fernandes [8] contains some tables showing factors versus culture. However, the text stops short of exhaustively listing this information. One table might show cultures A, B, and C, and a table on other factors might show A, D, and E.

Ito and Nakakoji have prototyped a system for retrieving culture-specific details [13].

The authors are currently undertaking a project to build a web-based repository. The intention is that developers and users from around the world will use and contribute to the database.

Resulting Context: The repository enables developers to easily access a corpus of culture-specific information. You can use this information to specify a `Multicultural System`.

4 Flexible Function

Context: You are producing a `Multicultural System` and an `Online Repository(3)` has been established. You have begun to specify the user-interface according to `Elastic User-Interface(5)` or you feel that it is more appropriate to specify functionality before the user-interface.

Problem: A culture-sensitive user-interface may contribute to *usability*, but it is still possible that the software does not support the tasks users would like to perform, i.e. lacks utility. These tasks and the context in which they occur can be related to culture.

How do you ensure the software performs functions which are meaningful and useful to people from different cultures?

Forces:

- Software is typically written with specific domains in mind, whether broad (e.g. a spreadsheet) or narrow (e.g. a code inspection tool).
- Domains—whether broad or narrow—are not homogeneous with respect to culture.
- Usability is also influenced by the user's culture, and usability derives from more than just the user-interface. Flexible searching, for instance, cannot be achieved just by applying `Elastic User-Interface(5)`.

Solution: **When you generate a new function, check if it is culture-specific, and if so, refine it to meet the needs of your target cultures.**

Break the requirements phase into several smaller stages. This way, you can progress incrementally, so that planning for the subsequent stage can take into consideration the cultures mentioned in the `Export Schedule(1)`. Whenever you create a new requirement or refine an existing one, consider its impact on the target cultures. Some cues which might suggest culture is an important factor include:

- a requirement depends upon legislation (a taxation rule).

- a requirement implies an organisational role (only certain people are authorised to shut down the assembly line).
- a requirement is based on a philosophical stance (a teacher can annotate text, but students can't [18]).
- a requirement is underpinned by certain ethical values (an employee's actions will be logged).

This process may generate new questions about cultures which the repository should be consulted to solve. If it cannot solve them, seek the most important answers by alternative means (e.g. interviews with domain experts) and update the repository. Once the answers to these questions are known, you will be in a better position to refine the requirements. Your initial idea for a requirement may form the basis of a suitable default, but you may need to extend it to satisfy all target cultures.

Examples: Time-keeping variations imply more than just differences in user-interfaces. Each supported calendar format requires functionality dedicated to handle standard operations (e.g. finding weekday from date, incrementing date). The situation becomes even more complex when differences in other areas, like timezones and work cycles, are considered.

Currency differences can lead to complicated functionality, especially when conversions are required. The introduction of the Euro is a familiar example.

Nielson discussed a French educational product which enables teachers to annotate poems [18]. He noted that in some countries, it would more appropriate to give students the same ability. The decision to include this kind of functionality rests on cultural values such as attitudes to learning and authority.

Resulting Context: Iterate between this pattern and `Elastic User-Interface(5)`. until you are satisfied with functionality and user-interface structure. Establish a `Universal Default(7)` for each function in case the user's culture has not been specifically catered for.

5 Elastic User-Interface

Context: You are producing a `Multicultural System` and an `Online Repository(3)` has been established. You have a solid understanding of general functionality after applying `Flexible Function(4)`, or you feel that it is more appropriate to specify the user-interface before concentrating on functionality.

Problem: The same functionality can be presented to the user in different ways. **How do you create a user-interface which can be used easily and effectively by all targeted cultures?**

Forces:

- Different cultures use different schemes for representing information. Text direction, for example, can be left-to-right, top-to-bottom, right-to-left, and even circular. Relative positioning of graphics and highlighting mechanisms also vary [21].
- Some cultures want more information, or different information, than others. Some cultures are more accustomed than others to inferring missing details [10]. Cultures may also vary in the nature of information required. The right feature cannot just be plugged in later on (e.g. a culture-specific icon or language-specific string).
- Human-human interaction varies considerably across cultures. Some notions of interaction are relevant to the human-computer dialogue, e.g. reactivity versus proactivity exhibited by participants.

Solution: Design the overall structure for the user-interface flexibly, so that GUI elements can subsequently be re-defined and rearranged without massive design changes.

The precise details about what information you are presenting, and how it is arranged will not become clear until the localisation phase. Therefore, develop an elastic user-interface so that the user-interface is not fixed during the internationalisation phase.

As with `Flexible Function(4)`, break the user-interface specification process into several stages, so that you can plan each subsequent stage in the light of the cultures being targeted.

Two general guidelines apply:

- Consult the repository or use other means (e.g. user testing) to discover how elastic the user-interface structure must be. A culture's writing direction can indicate the order in which material of a non-textual nature is also perceived. Also, text size will vary due to variations in alphabet, word lengths, grammar, and conventions. This will affect the space allocated for writing.
- Once the overall structure is present, you will have identified the GUI elements. However, the precise details of GUI elements depend on culture. Therefore, at this stage, identify which GUI elements are culture-specific and leave them abstract, e.g. a button to delete a file could be called *delete-file* and its type simply declared as a button with a bitmap and a label. After this pattern has been applied, an artist performing localisation might create a cross for one culture and a skull-and-crossbones for another, with accompanying labels in the appropriate language.

Rationale: Complicated changes to the user-interface, such as reversing order of elements, can necessitate widespread code changes if they are not designed for. Also, a common problem is the introduction of a language with text too large to fit in its usual location (e.g. a menu bar or dialogue box) [15]. This would be fairly trivial to prevent if it was known that the language was scheduled for introduction.

According to the Slinky meta-model [5], it is possible to split user-interface, human-computer dialogue, and functional core. Architectures based on Slinky should provide adequate support for developers who wish to create `Elastic User-Interfaces(5)`. Coldewey's "User Interface Software" pattern language also covers separation of user-interface and domain [3].

Examples: Some programs enable text to be shown and inserted in right-to-left as well as left-to-right modes. Farsi support within the VIM text editor lets users dynamically switch between orientations, and can simultaneously show one file in two different windows presented in separate orientations.

The web supports various layouts. Horizontal orientation of layout (e.g. whether links are on left or right) is correlated with the orientation of text in the originating country [1]. It is worth noting, though, that an individual site generally does not provide more than one layout.

Resulting Context: Iterate between this pattern and `Flexible Function(4)` until you are satisfied with functionality and user-interface structure. You can design specific features according to `Targeted Element(6)`. Establish a `Universal Default(7)` for the UI structure in case the user's culture has not been specifically catered for.

6 Targeted Element

Context: You have prepared an `Elastic User-Interface(5)`.

Problem: How can you create specific elements to plug into the user-interface framework?

Forces:
- The purpose, state, and workings of a user-interface element should be clear from its appearance.
- A user-interface element's appearance can be misinterpreted if the user's culture was not considered in designing it. An extreme case is when users are presented with foreign languages, but there are more subtle instances. The famous Macintosh trashcan looked like a postal box to Britons, causing great frustration when they tried to e-mail their work [22]!
- It is also possible for a feature to be correctly interpreted, but still be inappropriate. Many applications use hand gestures to represent certain concepts, but some of these are offensive in some countries [8]. Similarly, a given colour can convey different moods, depending on the culture in question [20].

Solution: For each abstract element contained in the `Elastic User-Interface(5)` specification, provide an instantiation targeted to each culture in the `Export Schedule(1)`.

The `Online Repository(3)` can be consulted for guidance. If it proves insufficient, investigate further and remember to feed results back in to the repository for next time.

Many elements will have a common instantiation for an entire group of cultures. This is where the idea of composite cultures discussed in the `Online Repository(3)` can be useful. For instance, a group of cultures called "English-language" can share the same text (of course, this would be a compromise because it is preferable to have "USA-English", "UK-English", etc.).

The `Online Repository(3)` can help to identify features which are misleading or offensive. However, user testing is also essential since people from external cultures can overlook this kind of problem.

Examples: This pattern is exemplified by any software which supports different languages, address formats, units of measurement, currency, etc. Tools and libraries like Java's MessageFormat class support the corresponding development process.

The culture-specific partners of the *Yahoo!* website are another example. They are organised in the same way as their parent, but contain information specialised to their culture, such as weather for certain cities and status of local stockmarkets.

Resulting Context: Adopt a `Language Policy` to help you decide which languages will be supported and to what extent. If your user-interface contains metaphors, ensure they are `Meaningful Metaphors`. Since it can be expensive to target every element to every culture, provide a `Universal Default(7)` in case the user's culture has not been specifically catered for.

7 Universal Default

Context: You have prepared functionality and the user-interface according to `Flexible Function(4)`, `Elastic User-Interface(5)`, and `Targeted Element(6)`.

Problem: Localisation is costly and requires substantial knowledge about target cultures. Providing many features for many cultures can lead to work in the order of n^2. **How do you optimise resources when there are many features (functions, user-interface arrangements, user-interface elements), each of which depends on a wide cross-cultural base?**

Forces:

- Features should generally be tailored, because a single version of a feature is rarely adequate for all cultures. If the single version is targeted to one culture, other cultures may suffer; if the version is intended to be universal and not belong to any particular culture, it may be too general and suit no-one at all.
- Tailoring software features to a culture is an expensive, time-consuming activity. It may involve extensive research and requires local experts.
- It is not always feasible to tailor software for small cultures or to cultures which have unique needs which are difficult to implement.
- An `Export Schedule(1)` cannot always correctly anticipate which cultures will use the software. People from unsupported cultures might move to targeted areas or download software from a website.

Solution: **For each culture-dependent feature, make a default which is universally meaningful.** The aim is to a fallback setting in case nothing has been tailored for some cultures. This will never be as good as a tailored feature, but it is better than producing a default which does not consider the cultural issue at all (e.g. one which is only suitable for the developer's culture). The following guidelines will help.

- Provide text in the most familiar language among supported cultures.
- Endeavour to produce images which are easy to understand and free from cultural bias.
- Since not everyone will know the most familiar language, try to replace text, e.g. button labels, with meaningful images.
- The familiar language may still be non-native for many users. Avoid the use of jargon and concepts which would only be familiar to a small subset of supported cultures.
- Avoid concepts which might offend some cultures.

Rationale: This pattern resembles the naïve approach to culturalisation which claims that we should simply provide universal features. The difference here is that the pattern is seen as a fallback to help reduce development effort, and is certainly not claimed to be ideal from the user's point-of-view.

Examples: Numerous websites provide versions in two languages: one in the native language and one in English. English is being used as the language which will reach the most people.

Some icons and metaphors from popular software paradigms are well-understood in different cultures. Web browsers often use left and right navigation arrows. Word-processors and painting programs use a blank page to represent a new document and scissors for cutting. Help is often represented by a question-mark.

Resulting Context: There is now a default for each user-interface feature, to account for cases when the user's culture has not been specifically catered for.

8 Remaining Patterns: A Summary

The following is a summary of patterns which were not covered above.

Vector Metamodel Determine the dimensions of cultures that interest you, and characterise each `Culture Model(2)` as a vector with a value for each dimension.

Multicultural System Identify culture-dependent features of the functionality and user-interface. Create an appropriate form of each feature for each target culture. Package the various forms together according to `All-In-One`.

Language Policy Form a policy explaining to what extent each culture will be supported, i.e. how much will be translated to the culture's primary and/or secondary languages.

Meaningful Metaphor Since metaphors should relate to everyday experience, tailor metaphors to meet cultural expectations.

All-In-One To avoid stereotyping users and ensure the software is flexible, produce one version with all forms of all features, so the user can tweak settings to their own needs. To accelerate the process, provide a profile of default settings for each target culture.

Citizen ID To decide which cultural profile to use, determine the user's culture on first use and ensure the choice persists until the user changes it.

Simultaneous Feature To support domains or users which deal with more than one culture, present a feature in more than one form at the same time.

9 Concluding Remarks

The Planet language is intended to spark a few thoughts in the minds of developers about what it means to create a truly international application, as opposed to one which is simply multilingual or culturally-neutral.

The language can be expanded in at least two directions. Firstly, Planet helps users but confronts developers with a more complicated development context. We have commented on technical details occasionally, but only to justify the efficacy of some of our patterns. It would be useful to develop a pattern language which encapsulates the complex details of issues such as language translation and time conversions.

A second expansion would be broadening the domain from single-user applications to encompass Computer-Supported Collaborative Work (CSCW). When computer systems are used to mediate interaction among humans from different cultures, a whole new set of challenges open up [4]. At this stage, we have deliberately avoided such complications. The patterns here would still be relevant, but numerous additions and modifications could also be expected.

10 Acknowledgements

The authors would like to thank Todd Coram, whose many insightful comments throughout the shepherding process led to a greatly-improved structure and description.

References

[1] W. Barber and A. Badre. Culturability: The merging of culture and usability, 1998. http://www.research.att.com/conf/hfweb/proceedings/proceedings/barber/. Accessed March 28, 1999.

[2] P. Bourges-Waldegg and S. A. R. Scrivener. Meaning, the central issue in cross-cultural HCI design. *Interacting with Computers*, 9(3):287–309, January 1998.

[3] J. Coldewey. User interface software, 1998. http://jerry.cs.uiuc.edu/plop/plop98/final_submissions/. Accessed March 30, 1999.

[4] J. Connolly. Developing a cultural model. In E. M. del Galdo and J. Nielson, editors, *International User Interfaces*, pages 20–40. John Wiley & Sons, New York, 1996.

[5] J. Coutaz. Architectural design for user interfaces. In J. J. Marciniak, editor, *Encyclopaedia of Software Engineering*, pages 38–49. John Wiley & Sons, New York, 1994.

[6] P. Danton de Rouffignac. *How to Sell to Europe*. Pitman, New York, 1989.

[7] V. Evers and D. Day. The role of culture in interface acceptance. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Human-Computer Interaction: Interact '97*, chapter 44, pages 260–267. Chapman & Hall, London, 1997.

[8] T. Fernandes. *Global Interface Design*. AP Professional, Chesnut Hill, MA, 1995.

[9] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[10] E. T. Hall and M. R. Hall. *Understanding Cultural Differences.* Intercultural Press, Yarmouth, Maine, 1990.

[11] L. Herman. Towards effective usability evaluation in Asia. In J. Grundy and M. Apperly, editors, *Proceedings of OZCHI 96*, pages 135–136. IEEE Computer Society, Los Alamitos, CA, 1996.

[12] L. M. Hynson, Jr. *Doing Business with South Korea: A handbook for Executives in the Public and Private Sector*. Quorum, New York, 1990.

[13] M. Ito and K. Nakakoji. Impact of culture on user interface design. In E. M. del Galdo and J. Nielson, editors, *International User Interfaces*, chapter 6, pages 105–126. John Wiley & Sons, New York, 1996.

[14] S. C. Jain. *Marketing Planning and Strategy*. South-Western Publishing, Cincinanati, OH, 1993.

[15] T.V. Luong, J.S.H. Lok, D. Taylor, and K. Driscoll. *Internationalization: Developing Software for Global Markets.* John Wiley & Sons, USA, 1995.

[16] K. Maney. Technology is “demolishing” time, distance. USA Today, September 2 1997. http://www.usatoday.com. Accessed January 20, 1998.

[17] M. K. Mooij. *Global Marketing and Advertising: Understanding Cultural Paradoxes.* Sage, Thousand Oaks, CA, 1998.

[18] J. Nielson. Usability testing of international interfaces. In J. Nielson, editor, *Designing User Interfaces for International Use*, pages 39–44. Elsevier Science Publishers, Amsterdam, 1990.

[19] F. Rafii and S. Perkins. Internationalizing software with concurrent engineering. *IEEE Software*, 12(5):39–46, September 1995.

[20] P. Russo and S. Boor. How fluent is your interface? Designing for international users. In S. Ashlund, A. Henderson, E. Hollnagel, K. Mullet, and T. White, editors, *InterCHI 1993*, pages 342–347. IOS Press, Amsterdam, 1993.

[21] P. Sukaviriya and L. Moran. User interface for Asia. In J. Nielson, editor, *Designing User Interfaces for International Use*, pages 189–218. Elsevier, Amsterdam, 1990.

[22] D. Taylor. *Global software: Developing Applications for the International Market.* Springer-Verlag, New York, 1992.

[23] E. Uren, R. Howard, and T Perinotti. *Software Internationalization and Localization*. Van Nostrand Reinhold, New York, 1993.