Reusing Knowledge about Users

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

Abstract

Requirements engineers should work towards specifications of systems which address both utility (relevance to user needs) and usability (e.g. learnability, efficiency). Our limitations in predicting human behaviour prevent us from preparing a user-centered specification from first principles. We argue that developers can improve on the situation by reusing the discovered knowledge itself, as well as reusing the processes of obtaining and using that information. The focus of this paper is internationalisation. We describe the design of a repository of cultural knowledge, and a pattern language which captures the process of acquiring that knowledge. Finally, we demonstrate our vision for these tools with an hypothesised scenario.

1 Introduction

A system is born when a general need or opportunity arises. For example, a company might realise it needs to improve its inventory-tracking process, or a shrink-wrap manufacturer may decide that there is a market need to provide a browser for first-time web users. Software requirements for such a system must be based on an understanding of who the users are and of the tasks they wish to perform, or there can be no guarantee of a usable product. Information about users is required to refine the basic description of the system into a detailed specification.

It is not always easy to obtain accurate user information. Users are often busy people, especially when they have a high profile in their organisation. Devoting time to interviews may be regarded as time wasted, or higher priority issues may take precedence. Even when users are consulted, human nature is so complex that it can be difficult to predict how people will react to the new system when it is delivered.

A tension therefore exists between the desire to produce user-centered requirements and the principle of specification before implementation. Typically, this is resolved by iterative lifecycle models, but this form of development can cause the program to degrade into a complex, unmaintainable mass of code if change requests are continually accommodated. Change is inevitable, and it is important to build flexible systems which do not constrain designers unnecessarily.

To avoid unnecessary changes, we need to improve our ability to predict user responses and attitudes towards the system. This paper suggests that we should reuse existing knowledge to achieve this goal. User experiences with implemented applications can facilitate future work. The "future work" might be a later version of the application, a separate program with the same users, or a separate program with users who are different but have similar characteristics. All three cases are plausible. Specifically, we are arguing the following:

- Developers should establish organisation-wide repositories of knowledge about their user base—"User Profile Repositories".
- Processes are required to guide the repository's use; a repository alone is insufficient.
- Pattern languages represent an effective way to describe the process associated with the repository, as well as the repository itself.

2 What can Requirements Engineers Reuse?

2.1 Requirements Reuse: Previous Work

There are already some usability-oriented approaches to reuse, such as Sutcliffe and Carroll's notion of classifying and reusing claims [13]. An example of a claim is *Rare event monitor*, which describes the use of warnings to inform the user of infrequent, dangerous events. The upside and downside of such claims can be explored and claims can be accumulated into a library. Pattern languages such as Tidwell's [14] interaction design patterns also facilitate reuse of usability-related concepts. The `Interaction History` pattern, for instance, suggests that the sequence of user interactions should be captured in certain situations.

The user profile repository discussed here is orthogonal to claims and usability patterns, though those techniques can support how the repository is used. The intention of a repository is to capture details about the characteristics of the users a developer caters for. This information can then be used in subsequent re-design and can also feed into other projects with similar user bases. Consider a developer who produces a portfolio of financial products for home users. After a market forecasting tool is released, the help desk receives numerous calls from users who find the language in the online help too technical. Access to this kind of information can be helpful not only for future versions of the forecasting tool, but also for other products aimed at the same market.

2.2 Illustrative Example: Capturing user characteristics relevant for internationalised software

In a previous paper, we described how the cultural variable impacts on both functional and non-functional requirements, and proposed a repository as a means of supporting the internationalisation process [9]. Such a repository is now under development, and is described in this section.

2.2.1 Database structured according to Cultures and Factors

In [9], we described how a culture can be classified into overt and covert factors, following a similar classification by Yeo [15]. Overt factors are tangible, obvious features of a culture such as

units of measurement. Covert factors are complex, ill-defined characteristics like mental disposition. We refined overt factors into six factors: time, language, writing, measures, formatting, and external systems. Covert factors was broken into mental disposition, perception, social interaction, context of use.

The repository is structured according to these factors, with a many-to-many relationship between cultures and factors. This makes it easy to support several standard operations:

- Search for keywords or use a more sophisticated query mechanism.
- Select a culture and see how it varies according to each factor. This would be useful for someone tailoring an application for a new market.
- Select a factor and see how each culture varies according to that factor. Sometimes, adding a new feature requires someone to consider how it will affect different cultures. An artist creating an icon for a new feature needs to know how every supported culture will interpret it.
- Select a combination of cultures and factors.

Since the information is stored in a database, it is relatively easy to support a new retrieval mechanism if the need arises.

2.2.2 Web-based interface

The repository is web-based, which has the following advantages over a paper-based equivalent:

- The repository can easily be shared across the entire organisation, even if staff are geographically scattered.
- No special software is required, just a regular browser.
- The interface is familiar to virtually all users.
- Hypertext can document associations between related entities.
- The user can explore the information space in different ways.

2.2.3 A user-modified, dynamic website

In a keynote speech at APCHI '98, Gerhard Fischer challenged the HCI community to design technology which extend users' horizons beyond the status of "couch potatoes"; in other words, to facilitate active design, rather than passive consumption [5]. An example is the Dynasites framework [12], which helps to create sites which users can directly edit. The distinction between users and contributors becomes blurred.

The internationalisation repository supports the notion of community ownership by providing a discussion forum for each piece of information. A single *statement* is present for each culture (e.g. Australia), each factor (e.g. language), and each culture-factor pair (e.g. Australian language). The statement effectively summarises the viewpoint of interested user-contributors. The corresponding discussion forum is a series of *comments* which can clarify the statement or

suggest new points. At present, it is planned that the site-wide administrator will feed comments back into the statement. Future versions should enable trusted users to directly alter the statement, and the site administrator will simply monitor changes.

The version under development already distinguishes between contributors according to whether or not they have created an account and whether their email address is known. The level of trust determines whether or not their comments can be placed immediately on the site. Also, it forms an indication for users as to the validity of a discussion point. Other information provided alongside each point is: the date it was entered, the statement on that date (which may have undergone revision), and the user name and email address if known. Also, it will be possible for user-contributors to rate the statement's validity as an extra guide.

2.3 Extending the Repository Concept

The repository described above would help developers who deliver software internationally. However, the general approach to capturing user needs is more general. Consider once again the developer of financial software. Although it was stated that its market is "home users", there are in fact many kinds of home users: some people do tax returns once a year, others trade shares on a daily basis. Marketers refer to such groups as segments and use them to guide design as well as other activities, e.g. targeted advertising campaigns [2]. User profiles might also relate to roles such as "receptionist" or "purchase planner", rather than a set of characteristics. Software methodologies sometimes refer to such models as "actor" [7]. Other methodologies mix characteristics with roles, as in Constantine and Lockwood's "user roles models" [3].

In general, a user profile repository can be structured according to a classification of users. Developers should make repositories common to all projects where users are similar in nature, rather than forming project-specific repositories. This approach maximises potential for information-sharing and reuse. In reality, people do not fit neatly into categories. However, modelling users in this way can be a helpful starting point. A user who seems to belong to two different categories still benefits from both categories having been considered. The software should still be made flexible enough to accommodate individual needs. This is one of the topics addressed in the following section.

3 Supporting Knowledge Reuse with Patterns

3.1 A Process to Complement the Repository

A user profile repository acts as a central point for storing knowledge. If it is well-organised, it can prevent a requirements engineer having to reinvent the wheel in order to discover some piece of information about users. It can answer questions such as:

- What do users know about topic T?
- What topics are related to topic T?
- Where was this information discovered? What is the supporting evidence?

However, many issues remain open. For example:

- How can the information be used to formulate or refine requirements?
- If new information is required, how can it be obtained?
- What can be done to avoid stereotyping users into fixed classifications?

Essentially, the missing information is techniques and processes by which the repository can be used. According to Cybulski, there are three phases in the requirements reuse cycle [4]:

Analysis Finding components which can be reused and determining how they can be generalised.

Organisation Storing and retrieving components along with associated classification and searching mechanisms.

Artefact Synthesis Selecting a component by choosing among candidates, adapting the result, and integrating it into the rest of the artefact.

A repository on its own helps with organisation, but does not tell people how to use it. There are different solutions to this dilemma. One might be to produce supplementary guidelines for usage. However, this still does not help with overall process. A quality assurance or management plan might be the appropriate way to treat this issue, but it will not deal with guidelines.

A more integrative approach is offered by the pattern language paradigm. This technique considers recurring design problems, the forces which designers must resolve, and solutions which have previously proven successful. Patterns originally arose in the field of building architecture. An example is `A Place to Wait` [1]. The *context*, i.e. situation when this pattern applies, is a place where people have to wait. One *force* is that people waiting cannot leave the area, while a *conflicting* force is the fact that the timing of the event is uncertain. When people are waiting together for a doctor or administrator, these forces combine to form an anxious situation for participants. A *solution*, i.e. a way to resolve these forces, is to draw people in who are not waiting. An *example* is a hospital which built a neighbourhood playground doubling as a children's waiting area. (Italics indicates standard attributes of design patterns.)

The solution of a pattern may suggest more, lower-level, problems to solve. This leads to more patterns. `A Place to Wait` leads to `Street Cafe`, for instance. Combining patterns in this way leads to a *pattern language*, which will provide explicit guidance from high-level to low-level issues. Pattern languages have been applied to software architecture [6], and more recently to higher-level issues such as software usability (e.g. Tidwell's patterns mentioned earlier [14]). To illustrate how patterns can help in the context of user profile repositories, we now describe a pattern language which supports the internationalisation repository.

3.2 Illustrative Example: A Pattern Language guiding use of the Internationalisation Repository

Figure 1 shows the relationships among patterns in the *Planet* pattern language for developing internationalised software [10]. Arrows in the figure indicate dependency. Thus, the arrow from `Targeted Element` to `Universal Default` means that `Targeted Element` solves one problem, and opens up a new problem which is solved by applying `Universal Default`.

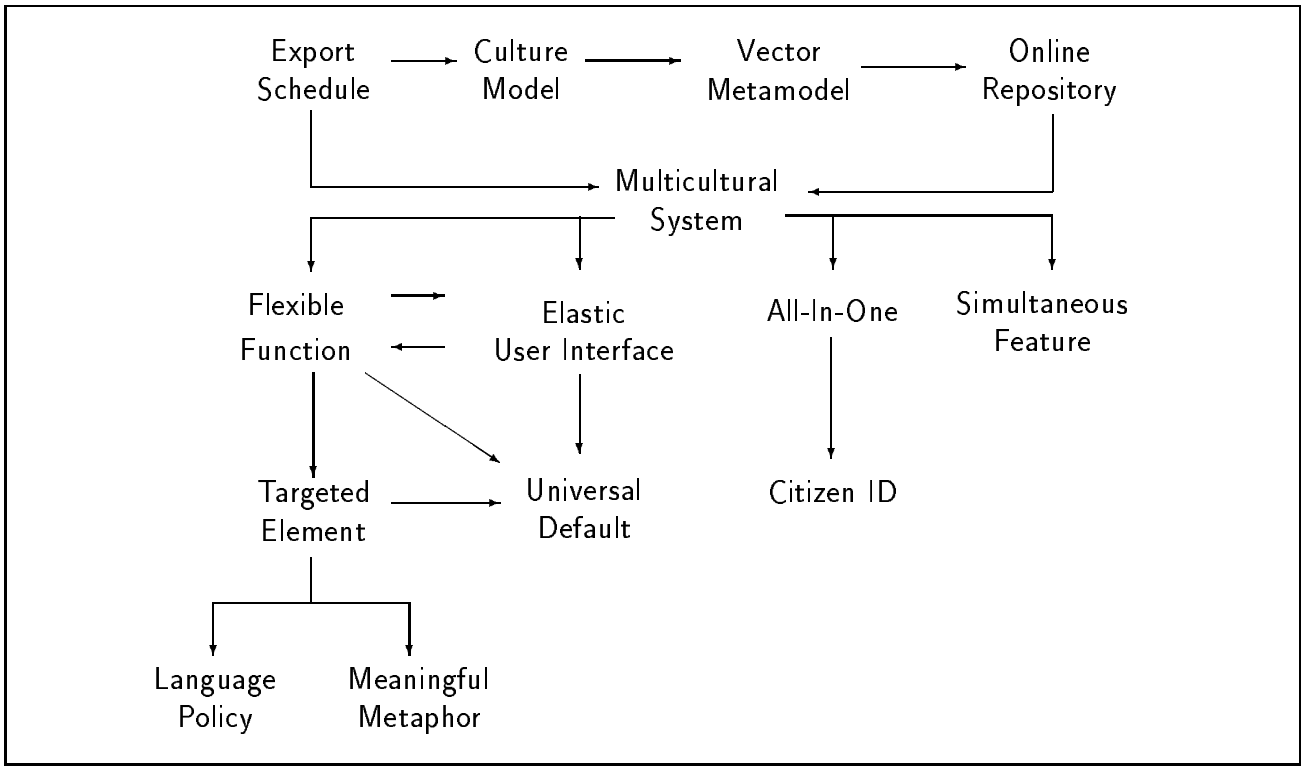

Figure 1: A map of the Planet pattern language for Software Internationalisation.

It is beyond the scope of this paper to explain the patterns in full detail. What follows is a summary of the intention of each pattern, with references to other patterns being shown in `Typewriter` font.

1. Export Schedule: Produce a schedule which indicates *when* each targeted culture will be supported, and *how important* it is to support each targeted culture.

2. Culture Model: Construct models of cultures which are relevant to your projects. When you discover new information about a culture, add it to the culture model.

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

4. Online Repository: Create an online repository, accessible to the entire organisation. Compose it of `Culture Models` all based on the same `Vector Metamodel`.

5. 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`.

6. Flexible Function: 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.

7. Elastic User-Interface: Design the overall structure for the user-interface flexibly, so that GUI elements can subsequently be re-defined and rearranged without massive code/design changes.

8. Targeted Element: For each abstract element contained in the `Elastic User-Interface` specification, provide an instantiation targeted to each culture in the `Export Schedule`.

9. Universal Element: For each culture-dependent feature, make a default which is universally meaningful.

10. 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.

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

12. All-In-One: To keep the software as flexible as possible and avoid stereotyping users, 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.

13. 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.

14. 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.

To illustrate the form of a pattern, a representative pattern—`Flexible Function`—is shown below.

Pattern Name `Flexible Function`

Context: You are producing a `Multicultural System` and an `Online Repository` has been established. You have begun to specify the user-interface according to `Elastic User-Interface` 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`.

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`. 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 cannot [11]).
- 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 [11]. 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` until you are satisfied with the functionality and user-interface structure. Establish a `Universal Default` for each function in case the user's culture has not been specifically catered for.

3.3 Pattern Languages for Other Repositories

The Planet language demonstrates how patterns can be used to deliver practical guidance to developers wishing to reuse knowledge about their user base. The language guides the overall lifecycle because it contains clear starting points, and individual patterns advise developers where to look after they have been applied. The pattern solutions provide cues to help developers determine what they should look for in the repository, and how the information might be obtained if it is not present.

Since patterns are combinations of artefacts and process, they are also a convenient way to describe the repository itself. Planet achieves this with `Online Repository`, `Vector Metamodel`, and `Culture Model`. Patterns, by their nature, facilitate a reusable process. In this application of patterns, reuse is even stronger because the process happens to be based around a repository which itself facilitates reuse.

Just as the internationalisation repository could be adapted to consider characteristics other than culture, so can many of the ideas represented by Planet be generalised. One general feature of Planet is that it is based on continually feeding back new information about users into the repository as it is learned. This is based on the view that real-life organisations cannot justify employing full-time staff to work on reusable components; instead, the library of components should be enhanced during everyday developments [8]. This principle is quite general, and could apply to a financial software company learning about its typical users. Thus, it is an example of the general nature of Planet in guiding repository usage.

Having said this, it should be noted that the Planet patterns can be used independently of the repository concept, even though we have chosen to explain the repository within Planet. A different developer trying to capture users characteristics could modify the patterns to reflect a different lifecycle or other concerns. In fact, there is no requirement to support the repository with patterns. We argue, however, that patterns provide an elegant way to do so.

4 A Scenario for Requirements Reuse

Following is an envisionment scenario to show how the repository and the pattern language work together.

David is a requirements engineer at Factory Expressware, an Australian company providing off-the-shelf productivity tools to manufacturers. The company has focused only on local customers to date, and has asked David to investigate the possibility of entering the Asian market with the existing Staff-Register program for performance monitoring and payroll management. David decides to create an internationalisation repository and apply the Planet language.

The starting point is to establish an `Export Schedule`. He explains to the marketing department and senior managers that the software development will progress more smoothly if they can provide any guidance about future plans. After some deliberation, the marketing department gives him a tentative schedule: full Japanese and Chinese versions within twelve months, with other countries flagged as possibilities, at least for partial versions.

David looks at Staff-Register and, following the `Vector Metamodel` pattern, begins to write down dimensions of culture that he will be interested in. Some are the general overt and covert factors such as units of measurement and mental disposition. Other culture-varying factors are specific to the product, e.g. How are organisations structured? What variables are used to monitor staff?

After management sets up local agencies in the target market, David flies out and interviews the new staff to help him build up `Culture Models`. He also has access to some potential customers. In each country, he gathers information according to dimensions he is interested in. For instance, he visits a factory and looks at hard-copies of performance reviews to see what variables are considered. The culture models evolve and are placed into the repository.

When David returns, he considers how the requirements must be changed to reflect what he has learned. First, he considers what new functionality is required (`Flexible Function`). Since taxation laws change the nature of employee payment, he refines the existing specification

to provide more flexible payment arrangements. After looking at functionality, he moves on to the user-interface (`Elastic User-Interface`). He found that in some countries, managers were evaluating the performance of workers whom they did not know very well. Usability will be enhanced by including a photograph of the worker, he reasons, and therefore specifies a user-interface which is "elastic" enough to show a photograph if desired.

Work continues over several months, with remaining patterns being applied and earlier patterns being revisited. The repository is continually updated as new information comes in. Eventually, the product is released, and management is happy with the result. At this point, the overall approach has helped David to structure his work.

However, it is a decision by management which brings out the full potential of the approach. Happy with the initial work on Staff-Register, several other products are scheduled for export, including an accounting package and an inventory monitor. Much of the material about users and their domain has already been discovered. General issues like currency are known, as are details specific to the manufacturing industry. Some of this will not be directly applicable, such as which variables are used in performance reviews. Others will save a lot of work, e.g. information on taxation laws. The fact that David will not have to re-obtain such details is only part of the benefit. A potentially more important advantage is that the Staff-Register program has already been implemented and tested in a real situation. If David hears complaints, he can update the repository accordingly. This allows undeveloped products to benefit directly from results of full-product testing. Thus, the problem of not being able to anticipate human needs becomes much less critical.

As for future versions of Staff-Register, these too will benefit. In expanding from English-only text to text in Chinese and Japanese, David could have required fonts only in those languages. However, the wiser choice of Unicode allows adaptation easily for other countries. If David used the `Export Schedule` effectively, he would have set in place flexible functionality and user-interface structures capable of adapting to the new cultures.

5 Conclusion

We have explained how user profiles can be maintained in a dynamic repository. This approach ensures that information gained during one project can be reused in the next project. Currently, this may be done via the memory of engineers or occasional cues in earlier documentation. However, a more formal mechanism will maximise reuse potential and avoid over-reliance on key personnel. Using the repository in conjunction with a pattern language provides a strong degree of support for the overall process.

The repository and pattern language discussed throughout this paper are specifically focused on cultural issues. However, they also act as templates, and developers could easily adapt them to consider other aspects of users. The internationalisation repository will be publicly-accessible, but a company could set up its own private repository. Aside from protecting its intellectual property, the benefit would be that information is specific to the company's user base. This means the attributes within the repository will be specific, as in the detail about performance reviews in the Section 4 scenario. The disadvantage, of course, is that the approach denies organisations the chance to learn from others' experiences. A hybrid approach might eventually be possible.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] H. E. Brown, R. Shivashankar, and R. W. Brucker. Requirements driven market segmentation. *Industrial Marketing Management*, 18(2):105–112, May 1989.

[3] L. L. Constantine and A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of User-Centered Design*. Addison-Wesley, 1999.

[4] J. L. Cybulski. Reusing informal requirements: Review of methods and techniques. In G. Shanks and P.A. Swatman, editors, *1st Australian Requirements Engineering Workshop*, pages 2.1–2.17. Monash University, Melbourne, 1996. http://www.dis.unimelb.edu.au/staff/jacob/publications/-awre-96/1AusRE-Fmt.pdf. Accessed June 15, 1999.

[5] G. Fischer. Beyond "couch potatoes": From consumers to designers. In J. Tanaka, editor, *Asia-Pacific Computer-Human Interaction (APCHI) '98 Proceedings*, pages 2–9. IEEE Computer Society, Los Alamitos, CA, 1998.

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

[7] I. Jacobson. The use-case construct in object-oriented software engineering. In J. M. Carroll, editor, *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, 1995.

[8] M. Laitkorpi and A. Jaaksi. Extending the object-oriented software process with component-oriented design. http://www.cs.uta.fi/~aj/omtpp/extobjco.ps. Accessed March 1, 1999.

[9] M. J. Mahemoff and L. J. Johnston. Software internationalisation: Implications for requirements engineering. In D. Fowler and L. Dawson, editors, *Australian Conference on Requirements Engineering (ACRE) '98*, pages 83–90. Deakin University, Geelong, Australia, 1998.

[10] M. J. Mahemoff and L. J. Johnston. The Planet pattern language for software internationalisation, 1999. To be published in *Pattern Languages of Program Design '99 Online Proceedings*.

[11] 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.

[12] J. Ostwald. Dynasites, 1997. http://www.cs.colorado.edu/~ostwald. Accessed March 1, 1999.

[13] A. Sutcliffe and J. Carroll. Generalizing claims and reuse of hci knowledge. In H. Johnson, L. Nigay, and C. Roast, editors, *People and Computers XIII: Proceedings of HCI '98*, pages 159–176. Springer, London, 1998.

[14] J. Tidwell. Interaction patterns, 1998. http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/. Accessed March 30, 1999.

[15] A. Yeo. World-wide CHI: Cultural user interfaces, a silver lining in cultural diversity. *SIGCHI*, 28(3):4–7, July 1996.