

# Making it easy

Edward Garson, senior consultant for Dunstan Thomas

Consulting, opens our look at user interface testing



*Traditional defect testing involves mastering a range of different types of testing and being assiduous in the pursuit of excellence.* The best testers tend to be exceedingly detail oriented and able consistently to find, exactly reproduce and communicate defects back to stakeholders. It is a distinctly rigorous discipline. There is usually no question as to whether exhibited behaviour constitutes a defect case or not.

On the other hand, usability issues are a totally different beast. Usability transgressions usually go wholly unnoticed by rafts of testers who are understandably focused on the functional behaviour of software.

IT professionals who take a conscious decision to challenge elements of the user interface of a system have the opportunity to excel through the provision of value over and above that which is expected of traditional development roles. Generally speaking, usability is misunderstood and applied by the development community at large. The opportunity to improve systems in a very tangible fashion exists and can feel very rewarding. Furthermore, most development shops do not have a usability expert on hand and generally do not place much emphasis on this aspect of product development.

Armed with some basic principles and the right tools, savvy and self-motivated IT professionals can make a big impact on the quality of the systems they work on by challenging elements of the user interface. But user interface design is as much an art as it is a science: it has the potential to be extremely contentious.

## Usability testing

The amount of time and effort spent testing user interfaces will naturally be a function of the perceived business value of a system. Usability experts recommend allocating 10 per cent of the overall project budget to usability. However this is unrealistic for most organisations.

True usability testing is the domain of usability specialists. Bona fide testing on large projects involves setting up a dedicated usability lab where session participants are videotaped and may be observed through one-way glass. Participants are given a prescribed number of tasks to perform. The manner in which participants go about completing tasks –

such as the navigation choices they make – reveals important information about the usability of various presentation components. Participants are encouraged to verbalise their thought processes to yield further insights during the evaluation.

In his classic tome on usability, *Usability Engineering*, Jakob Nielsen says “There are several methodological pitfalls in usability testing...” He acknowledges that there are problems inherent to “real” usability testing. The factors that contribute to this include disparities in the individual skills of test participants and large margins of error attributed to interpreting test results. As he puts it, “For usability engineering purposes, one often needs to make decisions on the basis of fairly unreliable data...”

The net result is that “real” usability testing should be a verifying and tweaking process when viewed in the context of a holistic approach to software development. This is because instituting changes at this point that are any more significant than tweaking can be a very expensive prospect. Usability testing

should be an integral part of the development process, irrespective of the methodology being used.

Testing user interfaces - either casually or seriously as described - demands knowledge of how to design them in the first instance. Understanding this process is integral to being a good UI designer.

## Use cases: an invaluable foundation

A very close relationship exists between use cases and user interfaces. It should therefore come as no surprise that use cases contribute to the design of user interfaces. A use case is a sequence of actions a system performs that yields an observable result of value to a user. Use cases describe goals of the system in a story-like fashion between the user and the system.

They stipulate the bare essentials in terms of data and interactions necessary to achieve the

functionality described by the use case. In doing so, they facilitate deriving simple user interfaces.

More discrete components within a system may be considered after having settled on the basic user interface. These components are born from requirements elicited during the analysis phase. The requirements are often grouped together into cohesive and meaningful units known as use cases. Use cases provide an invaluable foundation for arriving at the right user interface.

The use case diagram in figure 1 contains two use cases, *Purchase item* and *Browse catalog*. Use case diagrams provide an overview of the system as a whole, while individual use cases describe coarse-grained functionality.

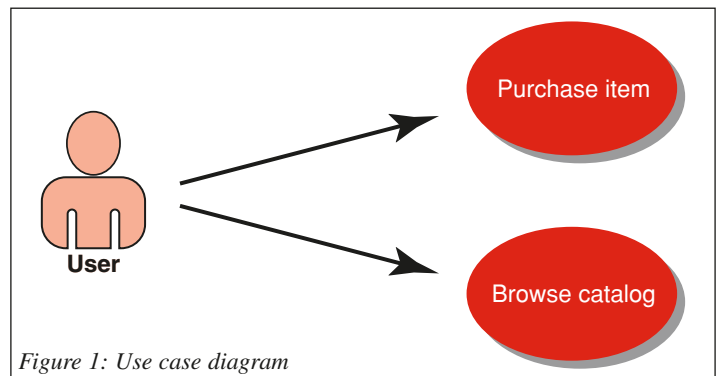


Figure 1: Use case diagram

When designing smart-client user interfaces (ie desktop applications), this broad view of the system is important in considering the foundational, application-level user interface. There exist only a few application-level user interfaces deemed familiar to users on the Windows platform. The vast majority of business-oriented, OLAP-style applications fall into one of these categories. They are:

- 1 Windows Explorer style user interfaces, with a hierarchical tree-view on the left and context-sensitive user interface elements on the right
- 2 Outlook-style user interfaces with a number of different “modes” and a task-centric philosophy
- 3 Multiple document interface (MDI) style applications with a parent window containing one or more child windows (considered a power user’s interface)

#### 4 Microsoft Excel-style user interfaces with separate individual tab sheets

The first step in designing or evaluating a user interface is to first consider the big picture. What foundational user interface style is best suited to meet the requirements of the use cases when considered as a whole? Choose one of the above, unless special circumstances dictate an alternative. Of course this does not apply to the design of a thin-client user interface.

Use cases stipulate the bare essentials in terms of data and interactions necessary to achieve the functionality described by the use case. In doing so, they facilitate deriving simple user interfaces.

#### Use case driven user interface design

Use cases are ideally generic with respect to implementation. There is ideally no indication of how functionality is rendered (eg by a web browser, handheld device or thick client application - or indeed, all three).

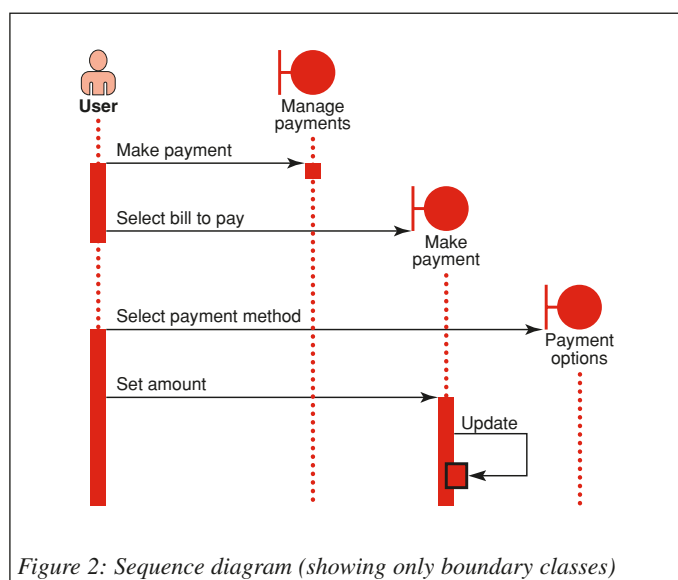


Figure 2: Sequence diagram (showing only boundary classes)

Each use case is supported by a use case document which describes the interactions between the user and the system in an implementation-agnostic manner. The use case should be focused solely on the minimum information required at each step and the workflow, which in the context of user interface design may be viewed as a kind of constraint.

The notion of generic use cases becomes important when testing user interfaces, because the use case is the definitive source of “the minimum of information” required to achieve functionality. Good UIs at a minimum require no more information than that outlined by the use case. Use cases should be completely focused on what is required of the user and the system, not how it will be achieved.

A major component of testing user interfaces involves verifying that how the user interface renders functionality marries up with what is required by the use case. Although this

may seem obvious, a surprising number of flawed user interfaces introduce extraneous steps or implicitly require more data than is required to achieve functionality. These transgressions should be caught first, before concentrating on lower-level details such as the layout of discrete presentation components.

Designing good user interfaces is all about facilitating how the user specifies what is required by a use case in as straightforward and task-oriented manner as possible. The steps required to achieve this will partially depend on the device rendering the user interface. This is where use case realisation comes into play.

#### Use case realisation

Suppose both a web browser and handheld device interface were required of the Purchase item use case. There will of course be significant differences in the user interface between the two implementations. However, the basic use case remains the same in that the same basic data and sequence is still required.

A use case realisation is the implementation of a use case in a specific instance. They exist to support the design process of internal software components, their collaborations and most importantly the user interface. Use case realisations play a pivotal role in the initial design and prototyping of user interfaces.

Use case realisations are developed into one or more sequence diagrams, each representing a scenario through the use

case. Sequence diagrams visually depict the interactions that take place between the user and the system in the course of realising functionality for a discrete scenario.

These diagrams are a great starting point for determining elements of the user interface. Decisions such as how to segregate data, the number of presentation elements required and the order in which they are shown starts to become clear during this activity.

#### Inductive user interfaces

Inductive user interface design has been well received in the usability community. Inductive user interfaces leave users in no doubt as to what they should do at any point in the system. These UIs are focused and have clear and purposeful intent. They are distinctly uncluttered and guide users toward accomplishing tasks. Contrast these user interfaces which expect some knowledge on the part of

the user about how to use the system, with inductive user interfaces, which in spirit do not. Deriving inductive user interface designs can be facilitated by following the design steps previously discussed, especially with respect to navigation.

The prototyping phase should reveal hard data about the decisions we made during the design phase such as how usable the proposed navigational structure truly is. This begs the question of how best to prototype user interfaces in support of this activity.

#### Paper prototyping: more than meets the eye

Paper prototyping is overall the single most effective tool for prototyping user interfaces. Paper prototyping is far more sophisticated than most people realise and should absolutely not be dismissed due to its apparent simplicity.

Benefits of paper prototyping include the fact that it is extremely cheap and easy to learn. It is possible to create and test more user interfaces faster than with any other technique. Paper prototyping also mitigates the apprehension that some users feel during “real” usability testing: some users subconsciously feel that they are in fact being tested, not the user interface. They feel “stupid” if they are unable to complete a task that is attributable to a poor user interface; this can skew test results. Playing computer with mock-up screens on paper reduces these apprehensions and better data is obtained.

Paper prototyping can also be turned on its head. Printouts (or paper mockups) of existing user interface elements can be made in order to test them objectively against proposed improvements or alternative designs. What is learned from these user interface tests can be used to lend credibility to the case for improving them. In this manner, ‘real’ user interfaces get tested against mockups on a level playing field.

It is important to prototype user interfaces ‘early and often’. Several iterations should be performed to get the user interface right. Testing it with a number of different stakeholders is critical to ensure that the user interface meets the requirements of disparate user types.

#### Micro usability

The following fundamental principles outline the basic characteristics of excellent user interfaces. They can be used as a basic guide to actively improve the usability of user interfaces in conjunction with other resources.

#### Simplicity

Simplicity is the foundation of all great user interfaces. Efforts to reduce complexity or ambiguity are always good. Great user interfaces reduce complexity and guide users

toward accomplishing difficult tasks. A good example of this is the Rules Wizard in Microsoft Outlook.

### Platform standards

Rich-client applications (as opposed to thin-client web interfaces) should follow the user interface design guidelines for the platform for which they are intended. For Windows applications, the bible is the *Official Guidelines for User Interface Developers and Designers*. Equivalent documentation exists for the Macintosh platform.

The fundamental principle is that applications should fit seamlessly into the environment in which they live. An application running on the Windows platform should feel like any other standard Windows application.

When testing for platform standards, rigorously apply the guidelines to the user interface. Flag any deviations as usability transgressions and cite the appropriate documentation.

### Consistency

Consistency is a very important element in user interface design. It is better for related things to work in the same way throughout the user interface than to be inconsistent, even if it is slightly flawed. Users learning systems tend to try doing the same things to invoke similar functionality: have them succeed.

### Education

User interfaces sometimes disable widgets that invoke actions that cannot be performed under certain conditions. The problem with this is that it may not be evident to the user how to change the state of the application in order to be able to invoke the desired action. It is sometimes appropriate to enable a widget that invokes an action that is inappropriate in a given context, so long as an informative message appears when the user does so. This improves the 'learnability' of the application.

This problem is hidden by the fact that the people who develop the system are expert in it and so they do not view this as a bug; in fact, they usually don't notice this at all. The ability to put oneself in the mindset of a first-time user is required to improve the UI from this perspective, which is not always easy.

### Informative error messages

Many software professionals are satisfied when an error message is correctly generated when testing error conditions. However, good error messages are the exception rather than the rule.

All error messages should be composed of an informative reason why the error occurred in very simple terms, and it is often helpful to include a suggestion as to how the error may be rectified.

It is interesting to note that error messages are almost invariably authored by the same person who wrote the code that generates them in the first place. Larger software projects should have a single person responsible for writing all error messages, to achieve consistency across the system.

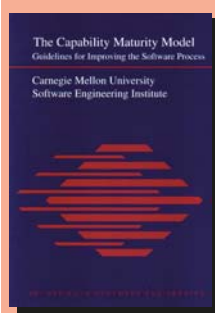
When testing the usability of error messages, play the role of a first-time user. Does the error message make it absolutely clear what went wrong? Would you know precisely what to do to achieve your original intent?

### Conclusion

The best way to learn about user interface design is to study good user interface designs and read up on the subject. There exists a vibrant and active usability community that readily share and disseminate their expertise on the web. It is possible to glean a lot of useful information this way. They put a lot of hard work into discerning what works and what doesn't. This information is shared with the community: it is there for the taking. The rigorous approach described in this article should yield a high degree of confidence that your user interface will be successful. This is because first and foremost it will meet requirements and hopefully do so in as straightforward a manner as possible. PT

# Test library

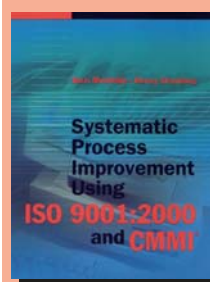
This quarter's new books about testing or of interest to testers



**The Capability Maturity Model**  
Carnegie Mellon University  
Software Engineering Institute  
Addison-Wesley, ISBN 0-201-54665-7

Excellent descriptions of performing assessments, identifying opportunities, progressing improvement and what process maturity means to both customer and supplier make this an essential reference. Published before CMMI so covers only CMM v1.1.

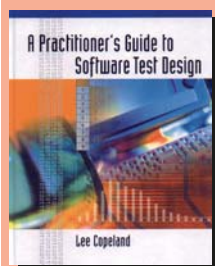
**Relevance to testing:** high, but generic



**Systematic Process Improvement Using ISO 9001:2000 and CMMI**  
Boris Mutafelija and Harvey Stromberg  
Artech House, ISBN 1-58053-487-2

Brings together the various models with very helpful emphasis on the synergy between them, getting roles and responsibilities well defined, and the need for full and continuous management support.

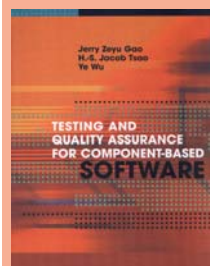
**Relevance to testing:** high, but indirect. No advice specifically for testers



**A Practitioner's Guide to Software Test Design**  
Lee Copeland  
Artech House, ISBN 1-58053-791-X

A very practical book of applicable techniques from one of testing's best-known names. Readable, friendly style, clear layout, a few jokes, and useful practice exercises. Recommended.

**Relevance to testing:** very high



**Testing and Quality Assurance for Component-Based Software**  
Jerry Zeyu Gao et al  
Artech House, ISBN 1-58053-480-5

A general testing textbook, but with examples and explanations drawn from component-reuse setups, eg EJB, COM+, CORBA etc. This doesn't cause much change to the basic testing theory which comprises much of the content.

**Relevance to testing:** very high