3F6 - Software Engineering and Design

Handout 9

User Interface Design

With Markup

Ed Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

The Golden Rule of UI Design

> "A user interface is well-designed when the program behaves exactly how the user thought it would."

Some guiding principles to help achieve this:

- **User familiarity:** Copy well-known features from other programs.
- **Consistency:** Do similar operations in the same way.
- **Choose default values carefully:** Minimise the number of choices novice users have to make.
- **Use metaphors:** To establish the user model.
- **Recoverability:** Ensure UNDO works in every situation.
- **Provide help:** But make it sparing and highly context specific.

Some constraints: Show Slides 1 to 8

- users never read the manual
- users can only read 2 or 3 words at a time
- users can not use a mouse
- users do not like making choices
- users do not like being frightened
- users do not like being needlessly confused
- USERS ALWAYS CLICK OK

The UI Design Process

1. Define a set of representative user types.
2. Identify the most important activities i.e. use cases.
3. Assume some user model based on how you think each user will expect to accomplish each activity?
4. Design the UI based on the current user model
   - use metaphors to illuminate the user model
   - borrow familiar behaviours from other programs esp. controls
   - keep it very simple, users will always assume the simplest model
5. Implement a prototype.
6. Test each use case with each user type and iterate.
   - do not ignore the minority in favour of the majority
   - note areas where people have trouble. These are probably areas where the program model does not match the user model.
7. Refine the user model and iterate from 4

User Types

User types are often application specific but common categories are

- nerds - enjoy figuring out how things work
- teenagers - like nerds but know no danger
- experienced users - think they already know how everything works
- naïve users - often too timid to explore the interface

There is also a dichotomy between

- People raised with computers.
- New users trying to use computers.

For each user type selected as appropriate for testing an application, a profile should be created answering the following questions:

- What are the user's goals?
- What are the user's skills and experience?

The answers to these questions will help design a strategy which allows users to exploit their skills and experience to efficiently satisfy their goals.

A good user interfaces should work well for all common user types. This is not always easy ...

Example:

Pete is a long standing Windows user who suddenly finds that he has to use a Mac program. Pete uses his experience to operate the apparently familiar user interface. He tries to

- drag window edges to resize
- hit space bar to dismiss message alerts
- press `Alt+F4` to close windows
- etc

but none of these things work!

Many people think the Mac interface is easier to use than Windows, but if it was really smart it would notice Pete aimlessly dragging the same window round in circles and wonder what he was really trying to do.

Use Cases

A *use case* describes a typical user activity from start to end.

A good set of use cases motivates a design, and provides a basis for testing.

A use case will consist of

1. Definition of the user's goal
2. List of pre-conditions (e.g. information needed before the task can start)
3. Criteria for successful completion
4. List of main steps in the activity flow
5. Any extensions/alternatives to this use case

This is also called *activity based planning.*

Example: a program to make and send greetings cards.

A simple functional approach would just define the functions that are needed:

- Create a new blank card
- Select a style from a library
- Add/edit text to a card
- Add a graphic to card
- Email card or print it out

A UI to implement this might then consist of a few standard menus

An experienced user would have no problem with this. However, a naïve user faced with a blank screen and just a few generic menu commands such as the above would be lost.

A tool bar with graphic icons might help but real novices need guidance.

Wizards

Wizards guide a user through pre-stored use cases one step at a time.

Example: continuing the greeting card example, the Wizard would lead the user through a sequence of steps:

Step 1 What do you want to do?

> 1. Send a birthday card?
> 2. Send an anniversary card?
> 3. Send a party invitation?
> 4. Start with a blank card?

User chooses (1)

Step 2 Select a background:

> User selects from a palate of birthday card designs

Step 3 Enter desired greeting:

> User types greeting into a text box

Etc.

Show slide 9

User Model

User model is the user's mental model of what is happening.

Note that different users will have different mental models. A good UI design encourages every user to adopt the same mental model.

Eliciting user models:

- usually 5-6 people is enough to gather a concensus
- present each user with a scenario and a task to complete
- use simple prototypes, screen mock-ups or just sketches to describe scenario
- ask user questions to understand how they would set about the task
- hence infer their mental model

Example: Designing a WYSIWYG HTML editor.

Everybody uses Word - but in Word images are embedded in the document. In HTML, images are not embedded, they are stored in separate files. What do you do?

- try to change the user model, if so how?
- modify the UI to hide the storage of images from the user.

User Input

There are five primary styles of issuing commands and data to a computer system:

Direct manipulation e.g. Media Players, Draw programs
: Fast, intuitive and easy to learn, but hard to implement and only suitable when there is an appropriate visual metaphor.

Menu selection e.g. Mobile phone, phone queues
: Avoids user error and needs little typing. But tedious for experienced users and is not suited to many complex options.

Form fill-in e.g. Stock control
: Good for data entry, but takes up screen space.

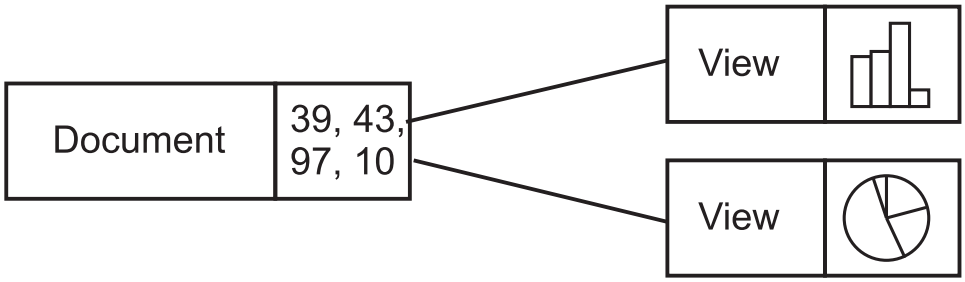
Command language e.g. Travel booking system
: Powerful and flexible, but hard to learn. Has poor error management.

Natural language e.g. Ask Jeeves
: Accessible and easily extended, but requires more typing and is unreliable.

Each has advantages and disadvantages, and styles are often mixed in the same application (e.g. Windows).

Information presentation

All interactive systems need some way of presenting information to the user. It is good design practice to keep the *presentation* separate from the information itself; this is sometimes called the *document-view* architecture.



The representation on the screen can be changed without having to change the underlying architecture.

There is a vast choice of display styles: bar charts, thermometers, dials, as well as just displaying text and numbers. In general, graphical display methods should be used as they can present trends and relative values. A digital display should only be used when precision is required, and even then graphical highlighting could be used to highlight changes.

Colour is very useful for highlighting or increasing clarity. However, don't use more than 4-5 colours, and be consistent with their use (e.g. red for danger.)

Error messages

Error messages should be there to *help* the user.

They should be

- polite
- concise
- consistent
- constructive
- appropriate to user's skill level
- accurate
- informative
- approrriate to setting

See slide 10–13: Good and Bad Error Messages

Metaphors

Using a metaphor can help a user align their mental model with the way that the program actually works.

The classic example is the ubiquitous "desktop" used in nearly all popular operating system interfaces.

A good metaphor is

- extendible - helps user infer how to do new things
- general - applies to whole program, not just one bit of it
- necessary - if the software is obvious to users do not introduce gratuitous metaphors.
- beware of cultural boundaries

See slide 14: Metaphors

UI Testing

Interface evaluation is the process of assessing the usability of an interface and checking that it meets the requirements. A user interface specification may include:

Learnability (time taken to learn)

Speed of operation

Robustness (tolerance to user error)

Recoverability (how well it recovers from errors)

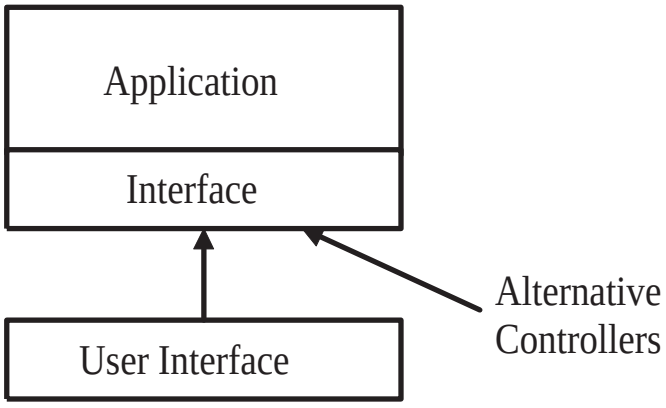
Adaptability (can it be used in different ways?)

One cheap way of evaluating a user interface is with a questionnaire. The questionnaire should be specific but simple, e.g. 'Please rate the quality of the error messages on a scale from 1 (clear) to 5 (incomprehensible).' Questionnaires can be based on a prototype, or even a paper mock-up.

There is no substitute for a prototype, and the best form of evaluation is to observe the user in a 'usability lab'. Ask them to talk you through what they are trying to do. Alternatively, one could observe them through a one-way mirror, or video them for future analysis. show slide 15

UI Implementation

User interfaces should be separated from the underlying application via well-defined interfaces:



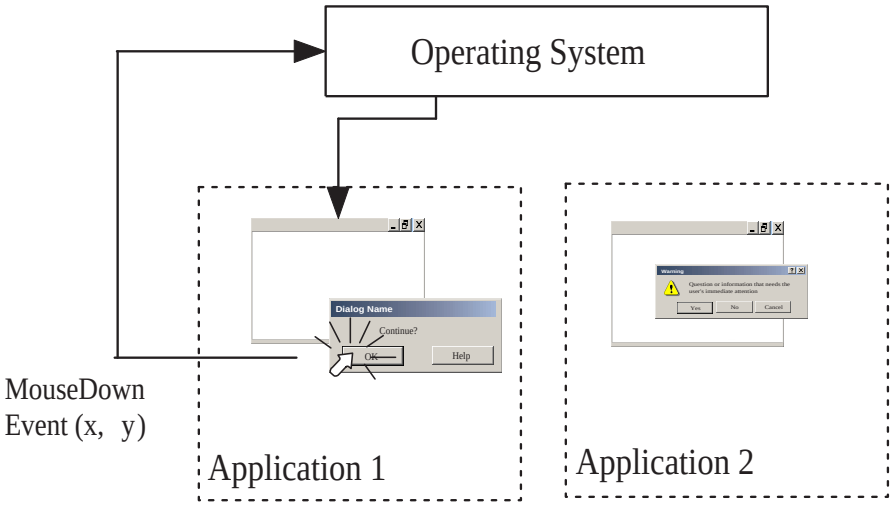
Providing an explicit interface allows alternative ways of controlling the application. Eg

- User-defined scripts to automate common tasks
- Embedded use within other applications (eg opening a browser window inside a word processor)
- Remote access

Note that specifying the interface using a programming language independent Inteface Definition Language (IDL) facilitates remote access and mixed-language working (cf CORBA later in this course).

Graphical User Interfaces

Early GUIs were single-threaded. They achieved the illusion of multi-tasking by using an event model. Modern GUIs often use multiple threads, but the event model remains.



The main program of each application contains an *event loop*

```
Event e;  // e encodes event details
do{
   e = GetEvent();   // wait here till next event available
   switch(e.type){   // for this application
     case MOUSEDOWN:
        HandleMouseDown(e);  break;
     case KEYPRESS:
        HandleKeypress(e); break;
      ...
   }
}until forever;
```

GUI Toolkits

GUIs are often implemented using application framework class libaries:

- pioneered by Apple via "MacApp", then by copied by Microsoft with the MFC (Microsoft Foundation Class) library
- basic functionality is coded in base classes e.g. windows, menus, scrollbars, buttons, etc
- "look and feel" is then dictated by class library so all applications built using the library adopt similar devices and metaphors
- user customises base classes to achieve required functionality e.g.
  - *decorator pattern* used to add widgets to windows
  - *observer pattern* used for synchronising multiple views of an object
- today there are literally hundreds of GUI toolkits available targetted at specific languages, operating systems, cross-platform, etc. Examples are Swing, TK, GTK+, OpenUI, Zinc, Motif, Qt, XVT, FLTK, Wx, ...

Summary

- The key ingredient to a successful UI is to ensure that the user adopts a mental model which is consistent with the underlying program model.
- To achieve this a compromise is needed
  - use metaphors, wizards, etc to guide the user to the right model
  - modify the program model to match user expectations.
- Use cases help design the UI and guide testing.
- Many toolkits are available for rapid GUI construction.