HCI252 Implementation support - extra notes

Alan Dix, 16/4/2010

The slides for this part of the course were taken largely from chapter 8 of the Human–Computer Interaction textbook, but with a few extra bits and sometimes in a slightly different order. These notes fill in any gaps and point to the relevant parts of chapter 8.

Introduction

(see also section 8.1)

The core question dealt with here is: how does HCI affect of the programmer? Obviously HCI affects the design and therefore *what* is produced; however, the nature of user interfaces can make certain aspects of programming the application more difficult. That is, HCI also impacts *how* the design is put together.

One example is the separation between applications in a typical desktop environment. From a software engineering point of view it is important to keep them separate so that if the word processor crashes it does not bring down the web browser and vice versa. However, when the user looks at the screen it is all apparently available – just as in real life you may use your mug to hold open the page of a book, you don't think "mugs are functionally different from pages" you just do it. However, on the desktop even cut-and-paste or drag-and-drop between applications is effectively breaking the functional separation. Within a single application the same issue arises and users expect to be able to freely move between aspects of the user interface even if they belong to different modules of the underlying application.

One issue is that the user-interface developer does not normally want to think about the details of the electronics of the trackpad on a laptop computer or the optical sensors on a mouse. That is there is a need for levels of abstraction that lift the programming from the specific details of hardware to interaction techniques.

This greater level of abstraction is provided by a development tools and techniques:

windowing systems – to provide basic independence from low-level devices and manage inter-application conflicts and

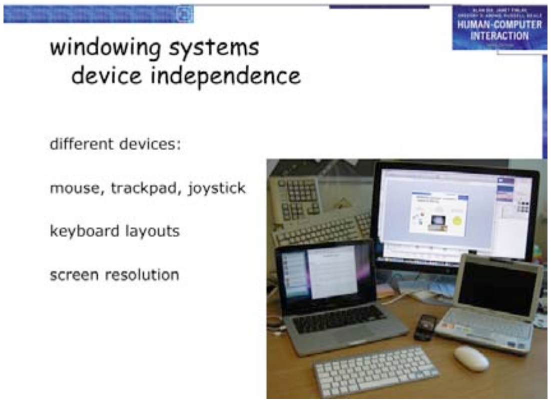
interaction toolkits – to make it easier to use higher-level widgets such as menues and buttons

architecture styles and frameworks – which help to structure the way one thinks about and constructs complex user interfaces

windowing systems

device independence

A computer may use many different kinds of pointing device (see also chapter 2): mouse, trackpad, joystick; it may even have a touchscreen such as an iPhone or iPad. Similarly there are various different kinds of keyboards from traditional QUERTY keyboards, to multi-tap phone keypads, and software-keyboards on touchscreens. Even 'standard' keyboards come in slightly different layouts in different countries and have more or less special keys such as cursor arrows, or function keys. Furthermore screens come in different resolutions from a few hundred pixels across on a phone to many thousands in a desktop 'cinema' display.



windowing systems
device independence

different devices:

mouse, trackpad, joystick

keyboard layouts

screen resolution

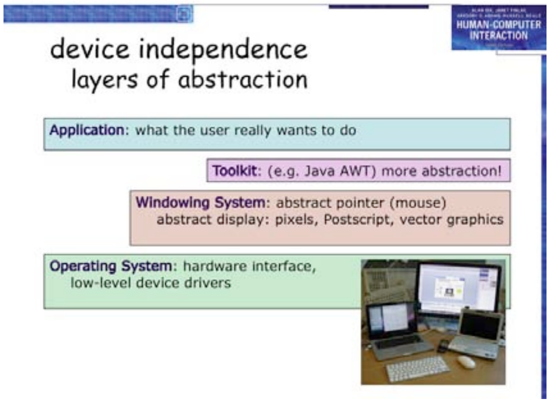
As an application developer you often want to ignore these details as much as is sensible for the nature of the interaction. This abstraction is provided by a number of layers of software. These differ slightly in different platforms, but the typical layers are:

operating system – This provides the basic hardware interface, and low-level device drivers.

windowing system – This has a number of functions, but one is to provide abstractions such as an abstract idea of a pointer/mouse and an abstract screen/display model. The display model is often based on pixels, but there are alternatives such as the use of Postscript or vector graphics (see box page 291).

toolkit (e.g. Java AWT/Swing) – These provide higher level abstractions such as components for menus, tabbed displays. Sometimes toolkits themselves come in several levels each adding more abstraction over the layer below.

The application will deal most with the highest level of toolkit, but typically can access the raw window manager or operating system when the toolkit does not provide everything that is needed. For example, the cut-and-paste support in Java AWT is limited, so for specialised applications you need to create small modules of native code to access the underlying window system clipboard. However, the more applications access underlying windowing systems or operating systems, the more code needs to be re-written when porting between platforms.



device independence
layers of abstraction

Application: what the user really wants to do

Toolkit: (e.g. Java AWT) more abstraction!

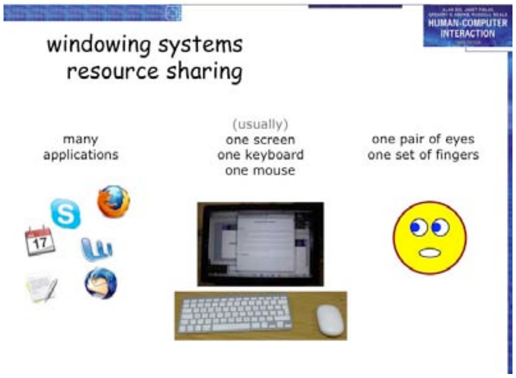
Windowing System: abstract pointer (mouse)
abstract display: pixels, Postscript, vector graphics

Operating System: hardware interface,
low-level device drivers

Fig 1. the event-paint cycle

resource sharing

Often you have many applications on a computer, but typically one screen, one keyboard and one mouse. Furthermore, the user has a single pair of eyes and fingers, so that even if you had a screen huge enough to show every possible application the user would not be able to loom at them all at once! One job of the windowing system is to share these fixed interaction resources between applications. See section 8.2, p 291, for the notion of a virtual/abstract terminal so that each application for many purposes can think it has the computer with all its resources to itself.

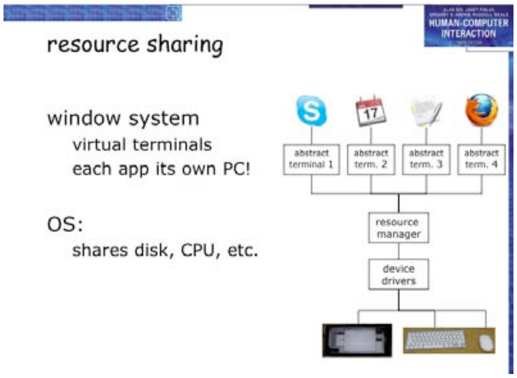


windowing systems
resource sharing

many
applications

(usually)
one screen
one keyboard
one mouse

one pair of eyes
one set of fingers



resource sharing

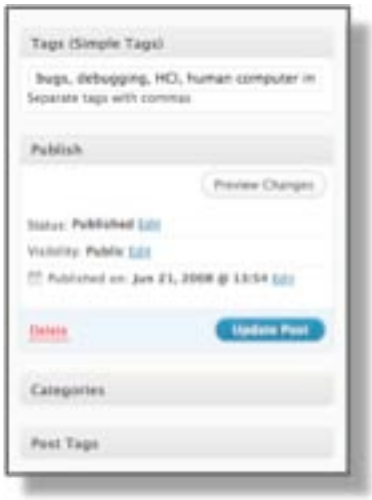
window system
virtual terminals
each app its own PC!

OS:
shares disk, CPU, etc.

The input devices are usually shared using the idea of input focus. By clicking on a window, or sometimes tabbing between them, the user can choose to use the keyboard to type into different applications, or the mouse to select or point in different windows. Note this is effectively *time-based sharing* as at any point in tome a single application 'owns' the keyboard and mouse.

For the screen there are several possibilities for window layout:

tiled (space shared) – Here each window/application is given a dedicated portion of the screen and can do what it likes there. This is often found in web sidebars, where widgets are stacked one above another. Note that this is a form of *space-based sharing* as each application has a part of the screen space. Of course, this has a natural limit when the screen is full. For web sidebars this is partly managed by the fact that the screen can scroll, however also each widget may be able to be hidden or expanded by clicking its title bar, thus giving the user more control over what is seen at any single moment.



bugs, debugging, HCI, human computer in

Separate tags with commas

Status: Published Edit

Visibility: Public Edit

Published on: Jun 21, 2008 @ 13:54 Edit

Fig 1. tiled sidebar with expandable widgets
in Wordpress admin screen

single app (time shared) – Some systems do the opposite and dedicate the whole screen to a single application or window, swopping which application gets the screen at any point in time. This was found in early versions of Windows, but is now more common in mobile devices such as phones as the screen is so small anyway that splitting it further would be silly. Note that this is a form of *time-based sharing* of the resource as at any moment precisely one application 'owns' the screen. The window system needs only have some means to swop between applications. For example, on the iPhone this is achieved by clicking the big button at the bottom and selecting an icon representing the intended application.

overlapping (time and space) – For desktop and laptop PCs, the most common layout is nether of the above, but instead the use of overlapping windows. In this case we have something that has elements of both time and space based sharing as some part of the screen have overlaps and are therefore time-shared (depending on which is on top) and other parts, where smaller windows sit side-by-side, are space shared.

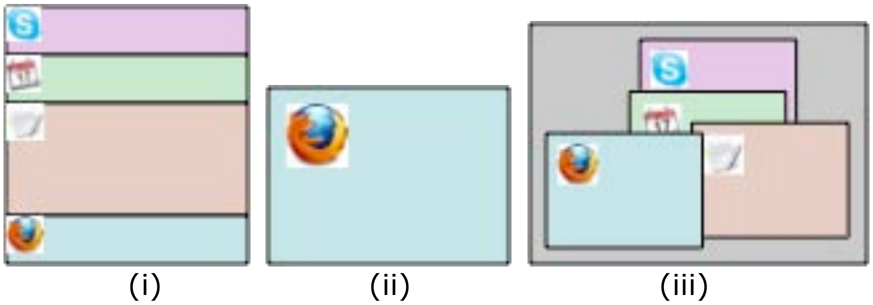


Fig 2. window layout (i) tiled (ii) single app and (iii) overlapping

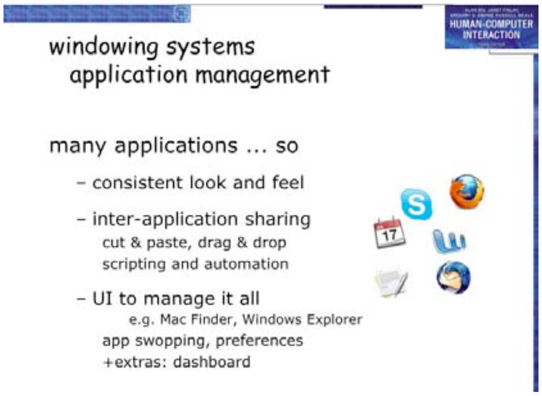
Of course it is not just the screen and keyboard that users care about; other aspects of the device are also shared, especially when thinking about a mobile device. There is one battery, so that power management is crucial. Some phone-based operating systems work very like desktop-based ones with applications running all the time and consequentially running down the battery! The iPhone is often criticised for being *single-threaded*, but this is almost certainly one of the reasons for relatively long battery life. The network is also a shared resource and if one application hogs it may slow down others ... furthermore it may cost the user in data charges!

application management

There are many applications! While they typically have control of what happens inside their windows, the window manager provides a consistent look-and-feel to the window 'decoration', the borders, resizing controls, title bar. As an application programmer you simply create a window and occasionally get events such as 'resized' or 'close', but otherwise the windowing system worries about what they look liek and how they behave.

The window system also takes responsibility for many aspects of inter-application sharing such as cut & paste and drag & drop. As an application developer there are typically calls to the windowing systems to say "here is some data of this type for the clipboard", and the windowing system will provide events such as "data of this type has been dropped here". The window system also manages a degree of negotiation between the application providing data (where it was cut/copied or dragged from) and the application using it (paste or drop location). Some windowing systems also provide a level of scripting or automation between applications, for example the Mac Automator.

Finally, the windowing system has to provide some form of user interface to mange things like swopping between applications, closing windows, selecting the keyboard focus, or arranging overlapping windows and setting various global user preferences (e.g. Mac Finder, Windows Explorer). As well as 'big applications', even desktop interfaces now often have additional micro-applications such as the Mac Dashboard, which also need means for activating, etc.



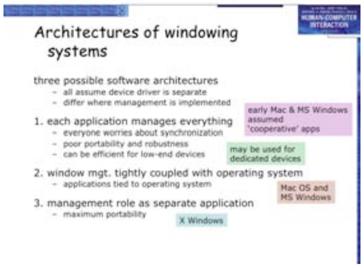
windowing systems
application management

many applications … so

- consistent look and feel
- inter-application sharing
  cut & paste, drag & drop
  scripting and automation
- UI to manage it all
  e.g. Mac Finder, Windows Explorer
  app swopping, preferences
  +extras: dashboard

architecture of window systems

(see section 8.2.1)



Architectures of windowing systems

three possible software architectures
- all assume device driver is separate
- differ where management is implemented

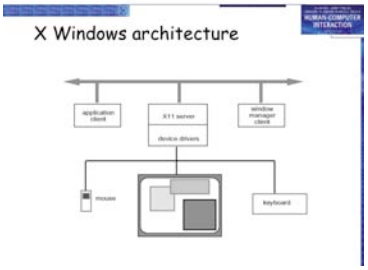
1. each application manages everything
   - everyone worries about synchronization
   - poor portability and robustness
   - can be efficient for low-end devices
2. window mgt. tightly coupled with operating system
   - applications tied to operating system
3. management role as separate application
   - maximum portability

early Mac & MS Windows assumed 'cooperative' apps

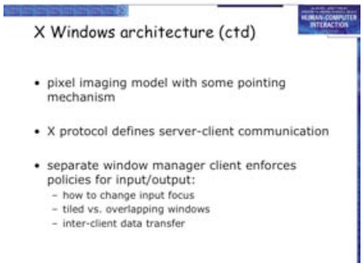
may be used for dedicated devices

Mac OS and MS Windows

X Windows



X Windows architecture



X Windows architecture (ctd)

- pixel imaging model with some pointing mechanism
- X protocol defines server-client communication
- separate window manager client enforces policies for input/output:
  - how to change input focus
  - tiled vs. overlapping windows
  - inter-client data transfer

not just the PC

Issues of application management and architecture are not just issues for PCs, but any platform where there are multiple applications:

phone – Faces similar issues to the PC sharing screen, keyboard, etc.. As noted the choice is usually to go for ‘full screen’ apps, not overlapping windows, however Vodafone 360 has semi-open apps, which take up several 'tile' locations in the screens showing application icons.

web – In web applications the browser in many ways takes to role of window manager. The browser may make use of he window system's ability to have several browser windows open, but within a window space is usually managed using tabs, which are effectively a form of space-based sharing.

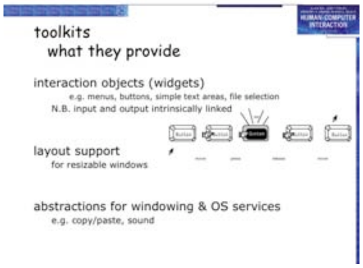
web micro-apps – Various web platforms allow the user to add micro-applications such as Facebook apps and Google widgets. The platform may offer these access to shared data (e.g. Facebook friend's birthdays) and have to manage screen space, often using a combination of space-shared columns and time-shared tabs.

dedicated devices (e.g. microwave control panel) – These are mostly coded direct to hardware as they have very bespoke input and output. However, there are appliance-oriented variants of Java and Windows providing some higher-level abstractions.

programming the application

toolkits

(see section 8.4)

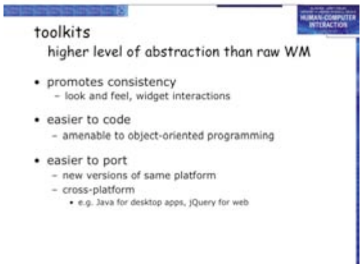


toolkits
what they provide

interaction objects (widgets)
e.g. menus, buttons, simple text areas, file selection
N.B. input and output intrinsically linked

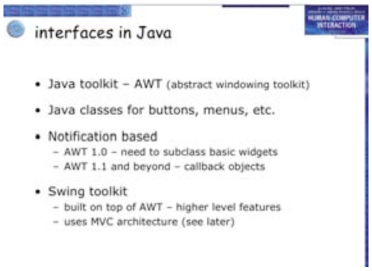
layout support
for resizable windows

abstractions for windowing & OS services
e.g. copy/paste, sound



toolkits
higher level of abstraction than raw WM

- promotes consistency
  - look and feel, widget interactions
- easier to code
  - amenable to object-oriented programming
- easier to port
  - new versions of same platform
  - cross-platform
    - e.g. Java for desktop apps, jQuery for web

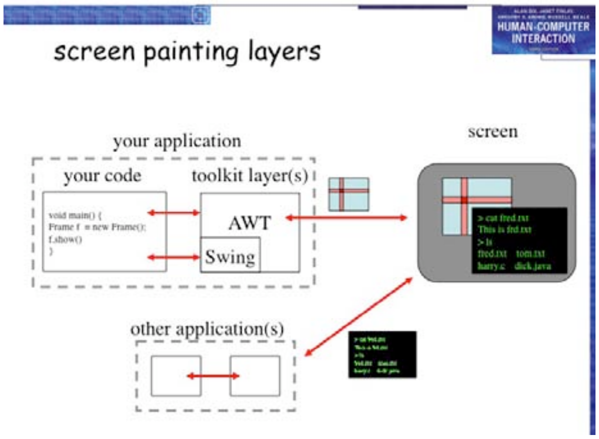


interfaces in Java

- Java toolkit – AWT (abstract windowing toolkit)
- Java classes for buttons, menus, etc.
- Notification based
  - AWT 1.0 – need to subclass basic widgets
  - AWT 1.1 and beyond – callback objects
- Swing toolkit
  - built on top of AWT – higher level features
  - uses MVC architecture (see later)

paint models

When you ant to put something on the screen it goes through all the layers referred to earlier. Your code interacts with a toolkit (say Jave AWT/Swing), which then passes this on to the window manager, which then manages things like overlaps with other application windows before interacting through the operating system and device drivers to actually paint pixels on the screen.



screen painting layers

```
void main() {
Frame f = new Frame();
f.show()
}
```

```
> cat fred.txt
This is frd.txt
> ls
fred.txt    tom.txt
harry.c    dick.java
```

Systems and toolkits differ in what you actually draw to:

direct to the screen – The simplest is when the application are given direct access to screen pixels. This is clearly most efficient for high-throughput graphics, such as vide replay, but has problems if the application misbehaves and starts to draw to areas of the screen that it shouldn't such as other applications' space or the windowing system's own UI elements.

direct to screen via viewport – The windowing system may exert a little more control by only allowing access through a 'viewport'. This means that when the application asks to draw pixels the output may be clipped if it is outside the allowed region, or if part of the window is currently covered by another window. This might also include coordinate transformations o that the application can effectively draw in a window with x and y coordinates 0-199, but have the window really positioned in the middle of the actual screen. For example, Java AWT works in this manner.

separate buffer – Sometimes instead of writing instantly to the screen the applications drawing operations are applied to a buffer in memory and only when this is finished is the whole buffer written to the screen. This may happen at the level of toolkit or and/or underlying window system and is normally called *double buffering* in a toolkit or retained bitmap in the windowing system. The latter we deal with later. The reason for double buffering at the toolkit level (whether or not the windowing system has a buffer), is to reduce flicker. Without double buffering it may be that the user sees a screen half-drawn, whereas double buffering means that the entire window instantly swops from the old to the new buffer. This is an option in Java Swing.

display list – Instead of working wit the screen as an array of pixels, some systems store a list of operations recording, say, that there should be a line, image ort text at a particular location on the screen. The toolkit then worries about showing these on the screen. This means that the application can simply change the display list when screen elements need to be updated, rather than redraw the whole screen. Also it allows the toolkit or window system to perform optimisations and hence is used in some high-performance graphical toolkits including OpenGL.

Buffering may also be used by the window system to store parts of the application window in order to more quickly update the screen when the user is swopping between overlapping windows. For example Mac OSX offers applications the choice of three levels of buffering which differ largely in how they cope with overlapping or translucent windows.:

nonretained – This is the simplest options, the window manager remembers nothing and whenever a part of the application window that was hidden is later exposed the application is asked to redraw the previously hidden portion. This works best if the contents of the application window are changing very rapidly as any hidden parts will need to have fresh contents anyway.

retained – Where the window manager buffers just the hidden parts of overlapping windows. This means that when the window is later exposed the hidden part can be instantly drawn. Note that 'hidden' here includes being covered by a translucent overlay which may later move.

buffered – Here the window manager keeps a copy of the entire window, both hidden parts and non-hidden parts. This takes most memory, but gives the maximum responsiveness if, for example, the window is itself translucent and is dragged over other windows.

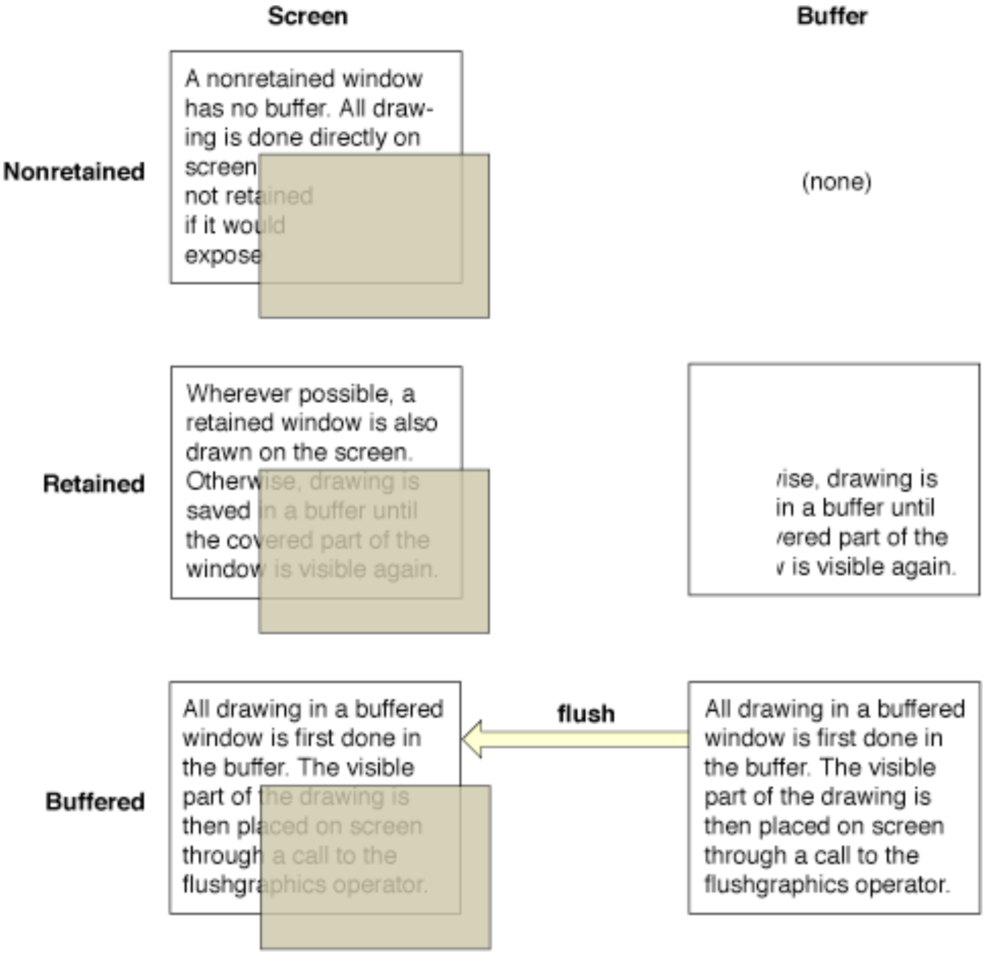

Screen
Buffer
Nonretained
A nonretained window has no buffer. All drawing is done directly on screen
not retained
if it would
expose
(none)
Retained
Wherever possible, a retained window is also drawn on the screen. Otherwise, drawing is saved in a buffer until the covered part of the window is visible again.
vise, drawing is
in a buffer until
vered part of the
v is visible again.
Buffered
All drawing in a buffered window is first done in the buffer. The visible part of the drawing is then placed on screen through a call to the flushgraphics operator.
flush
All drawing in a buffered window is first done in the buffer. The visible part of the drawing is then placed on screen through a call to the flushgraphics operator.

Fig 3. Buffering options in Mac OSX. from http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CoreAppArchitecture/CoreAppArchitecture.html

There are different reasons why the screen needs to be redrawn:

internal events – Sometimes it is an event inside the application which leads to the needs to update the screen. For example, in a clock the digits need to change, or if downloading a large data file the progress indicator may need to update. In the case of internal events the application 'knows' that the screen has changed, but may need to tell the toolkit and ultimately the window manager.

external events – Alternatively the event may be due to something the user did to the application. For example the user might have clicked the 'bold' icon and the currently selected word needs to be emboldened. In this case it is the window manager that first 'knows' that the user ahs clicked the mouse, then passes this to the toolkit, which may sometimes respond directly (e.g. during navigation of a menu) or pass it on to the application.

However, just because the screen needs to be updated does not mean the update happens at that moment. For example, if there were many updates within a few tens of milliseconds, it would not be worth updating the screen several times as this would all be within a single display frame.

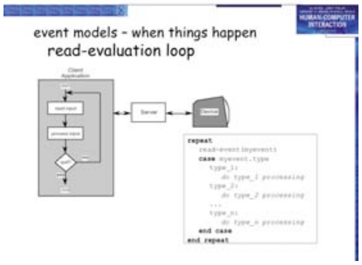
internal control – This is probably the easiest option to understand. When the application wants the screen changed it simply tells the toolkit/window manager that something needs to be updated. This has the problem noted above of potentially wasted updates, or taking time redrawing the screen when maybe user input is queued up needing to be processed. However this method works fine if there is some sort of intermediate representation such as the display list or a retained bitmap as then the actual display is only updated once per frame.

external control – In this case the toolkit / window manager decides when it wants a part of the screen to be updated (for example, when a hidden part is exposed) and asks the application to draw it. In Java this is what happens in a 'paint()' method. However, while this works easily for externally generated events when the window system 'knows' that a change is required, there is of course a problem for internally generated change. This is the purpose of the 'repaint()' method in Java; the application is saying to the toolkit "please repaint my window sometime when you are ready". At some later point, often when the input event queue is empty, the toolkit calls the applications 'paint()' method and screen is updated.

dra*w once per frame* – This is a variant of external control used principally in video-game engines where much of the screen updates in each frame (e.g. first person shooters, or car racing). Once per frame the application is asked to redraw the screen. If event happen between these calls the application usually just updates some internal state but does not update the screen itself. Then when it is asked to redraw itself, the application takes the current state, maybe polls the state of joystick buttons, and generates the new screen.

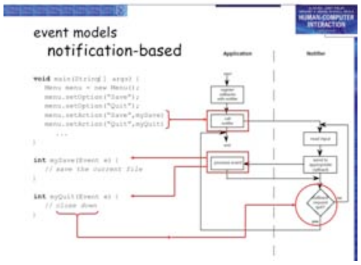
event models

(see section 8.3 and notes "Events in Java: what happens when you click?")



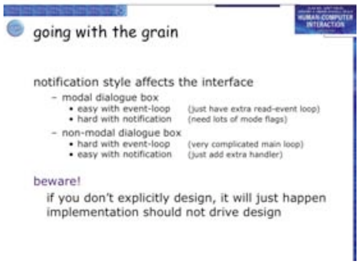
event models - when things happen
read-evaluation loop

```
repeat
   read-event(myevent)
   case myevent.type
      type_1:
         do type_1 processing
      type_2:
         do type_2 processing
      ...
      type_n:
         do type_n processing
   end case
end repeat
```



event models
notification-based

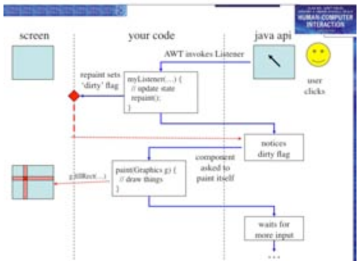
```
void main(String[] args) {
   Menu menu = new Menu();
   menu.setOption("Save");
   menu.setOption("Quit");
   menu.setAction("Save",mySave)
   menu.setAction("Quit",myQuit)
      ...
}

int mySave(Event e) {
   // save the current file
}

int myQuit(Event e) {
   // close down
}
```

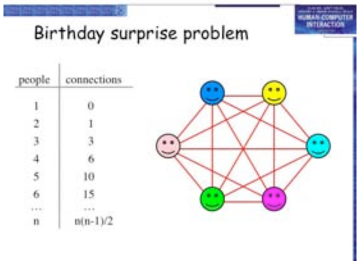


going with the grain

notification style affects the interface
- modal dialogue box
  - easy with event-loop (just have extra read-event loop)
  - hard with notification (need lots of mode flags)
- non-modal dialogue box
  - hard with event-loop (very complicated main loop)
  - easy with notification (just add extra handler)

beware!
if you don't explicitly design, it will just happen
implementation should not drive design





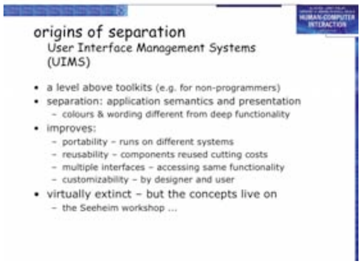
Birthday surprise problem

people	connections
1	0
2	1
3	3
4	6
5	10
6	15
...	...
n	n(n-1)/2

architecture and frameworks

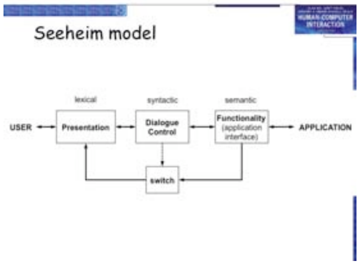
separation and presentation abstraction: Seeheim

(see section 8.5, page 307–309)

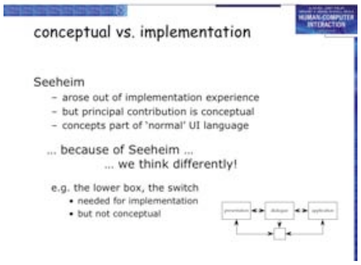


origins of separation
User Interface Management Systems (UIMS)

- a level above toolkits (e.g. for non-programmers)
- separation: application semantics and presentation
  - colours & wording different from deep functionality
- improves:
  - portability – runs on different systems
  - reusability – components reused cutting costs
  - multiple interfaces – accessing same functionality
  - customizability – by designer and user
- virtually extinct – but the concepts live on
  - the Seeheim workshop ...



Seeheim model

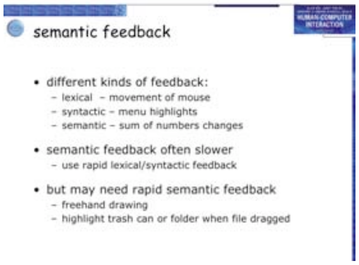


conceptual vs. implementation

Seeheim
- arose out of implementation experience
- but principal contribution is conceptual
- concepts part of 'normal' UI language

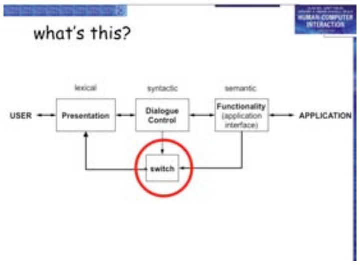
... because of Seeheim ...
... we think differently!

e.g. the lower box, the switch
- needed for implementation
- but not conceptual

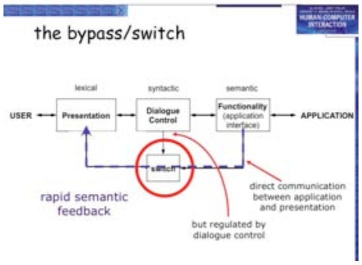


semantic feedback

- different kinds of feedback:
  - lexical – movement of mouse
  - syntactic – menu highlights
  - semantic – sum of numbers changes
- semantic feedback often slower
  - use rapid lexical/syntactic feedback
- but may need rapid semantic feedback
  - freehand drawing
  - highlight trash can or folder when file dragged



what's this?



the bypass/switch

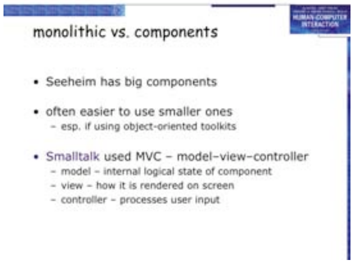
rapid semantic feedback

direct communication between application and presentation

but regulated by dialogue control

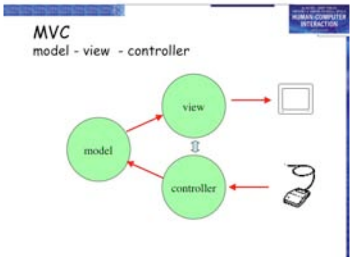
component models: MVC and PAC

(see section 8.5, page 308)

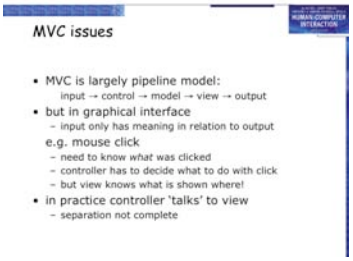


monolithic vs. components

- Seeheim has big components
- often easier to use smaller ones
  - esp. if using object-oriented toolkits
- Smalltalk used MVC – model–view–controller
  - model – internal logical state of component
  - view – how it is rendered on screen
  - controller – processes user input



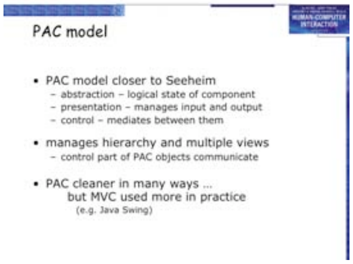
MVC
model - view - controller



MVC issues

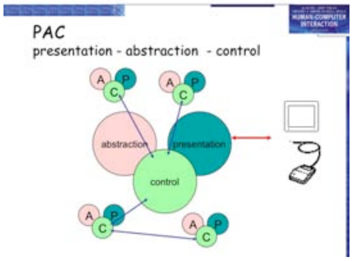
- MVC is largely pipeline model:
  input → control → model → view → output
- but in graphical interface
  - input only has meaning in relation to output

  e.g. mouse click
  - need to know *what* was clicked
  - controller has to decide what to do with click
  - but view knows what is shown where!
- in practice controller 'talks' to view
  - separation not complete



PAC model

- PAC model closer to Seeheim
  - abstraction – logical state of component
  - presentation – manages input and output
  - control – mediates between them
- manages hierarchy and multiple views
  - control part of PAC objects communicate
- PAC cleaner in many ways …
  but MVC used more in practice
  (e.g. Java Swing)

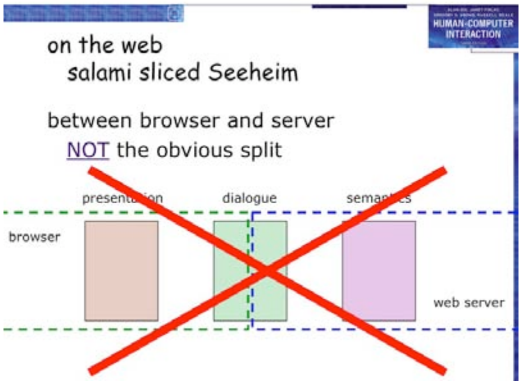


PAC
presentation - abstraction - control

on the web – salami sliced Seeheim

The Seeheim. MVC and PAC architectures were all developed in the light of desktop GUI applications. In web applications we can see similar facets but often arranged differently, and in particular the components do not necessarily all sit together in one place.

Given the browser is the bit close to the user it is tempting to think that Seehem presentation layer would live entirely in the browser, with the deeper functionality/semantics at the server end and a separation somewhere in the dialogue component.



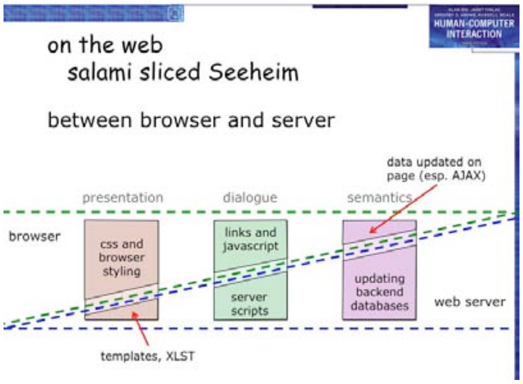
on the web
salami sliced Seeheim

between browser and server
NOT the obvious split

In fact, it is not this simple. Instead, we typically see bit of each layer spread between browser and server. At the presentation layer, the actual layout is clearly performed in the browser as is the application of CS style sheets, managing layout of screen areas depending on the window size etc. However, the actual HTML that is delivered to the web page is usually generated in the server either with raw code, or with the help of template engines such as Smarty, or in XML-based web-stacks XSLT to transform XML to HTML.

The dialogue component is similarly split. Some things happen at the browser end. In an old HTML site this is limited to when the user selects links or interacts with web forms; however in a Javascript-rich or Flash-based site the level of interaction can be quite high. However, some of the dialogue is happening at the server side. When the user selects a link to a generated page or presses a form 'submit' button, the back-end server script needs to work out what to do next.

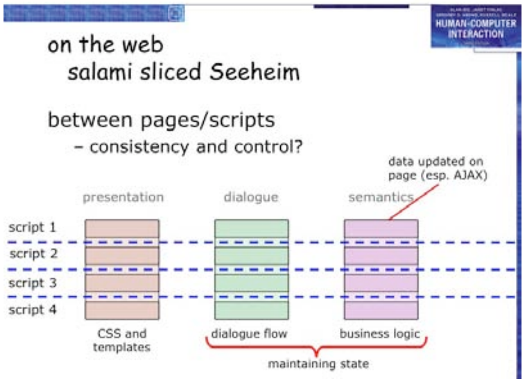
Finally the semantics is largely situated in back-end databases and business logic. However, in rich web applications including those based on AJAX and Web2.0 technology, data may be updated on the page. For example, Google docs spreadsheet is downloaded completely into the web browser and edited there with updates periodically sent to the backend server to keep the two in synchronisation.



on the web
salami sliced Seeheim

between browser and server

Furthermore if one looks at the scripts generating individual application pages, we will typically see bits of presentation, dialogue and semantics in each. The problem then is that the interaction state becomes fragmented and often developers find it hard to keep track of state spread between server session variables, URL parameters, hidden fields in forms and cookies. There are some MVC-like frameworks for web development, which try to untangle the mess, but the picture is far from solved.



on the web
salami sliced Seeheim

between pages/scripts
– consistency and control?