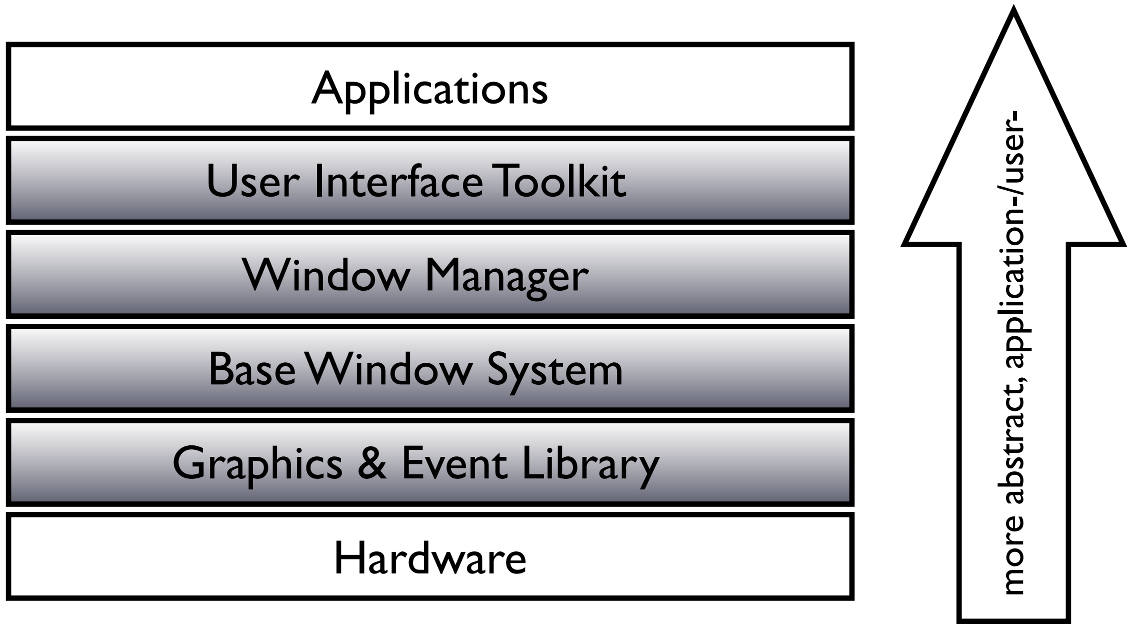
Designing Interactive Systems II

Computer Science Graduate Programme SS 2010

Prof. Dr. Jan Borchers
Media Computing Group
RWTH Aachen University

http://hci.rwth-aachen.de/dis2

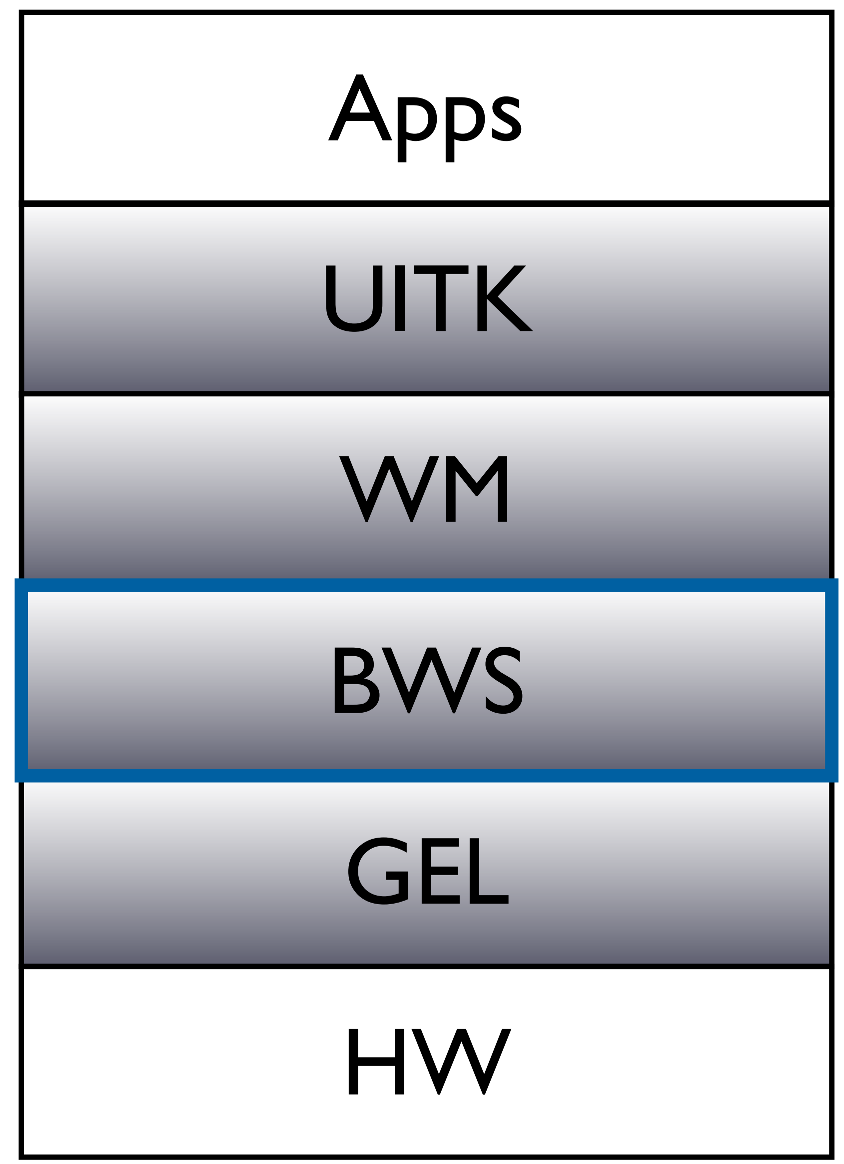


Review

- 4-Layer Model
- Graphics and Event Library
  - Hides hardware and OS aspects
  - Drawing operations

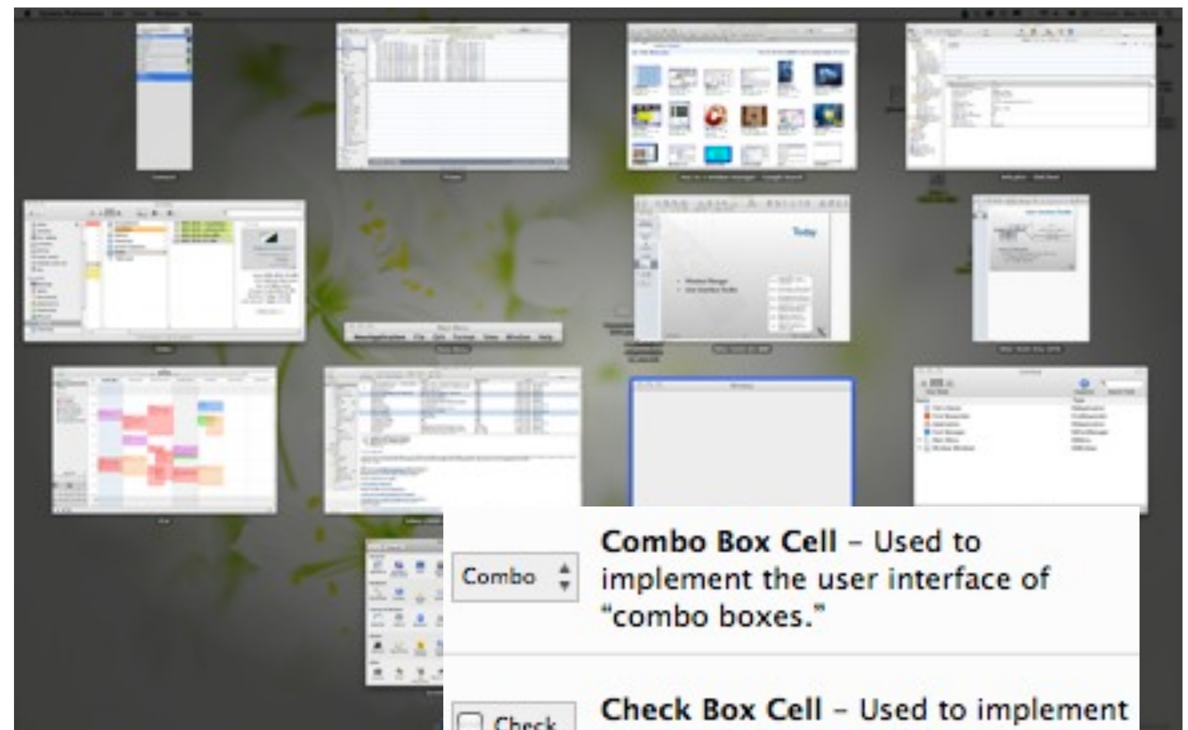
Review

- Base Window System
  - Map n applications with virtual resources to 1 hardware
  - Offer shared resources, synchronize access
  - Windows & canvas, graphics contexts, color tables, events
  - Event multiplexing and demultiplexing
  - Window hierarchies

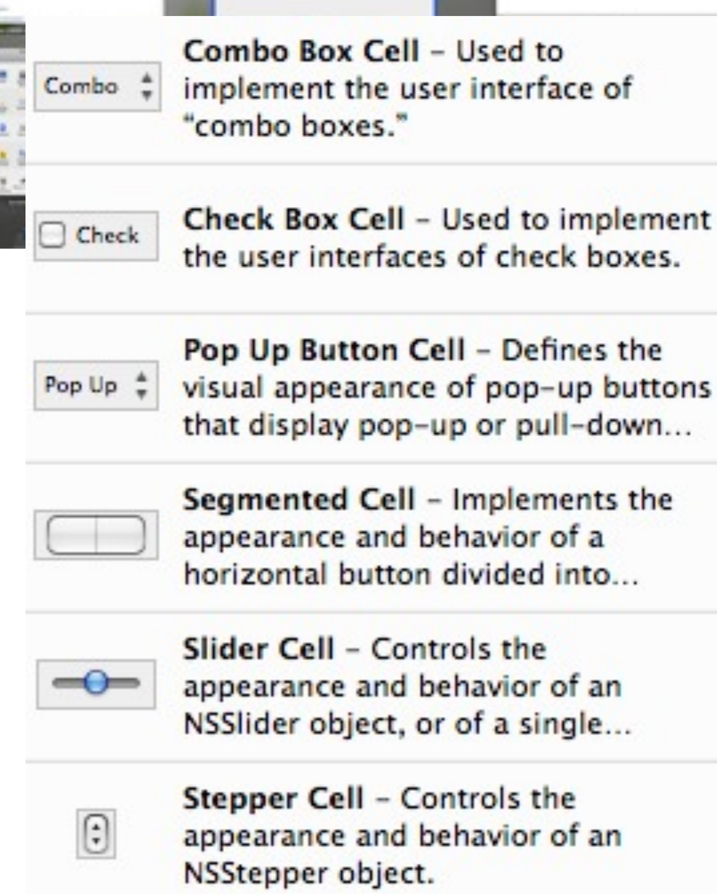


Today

- Window Manager

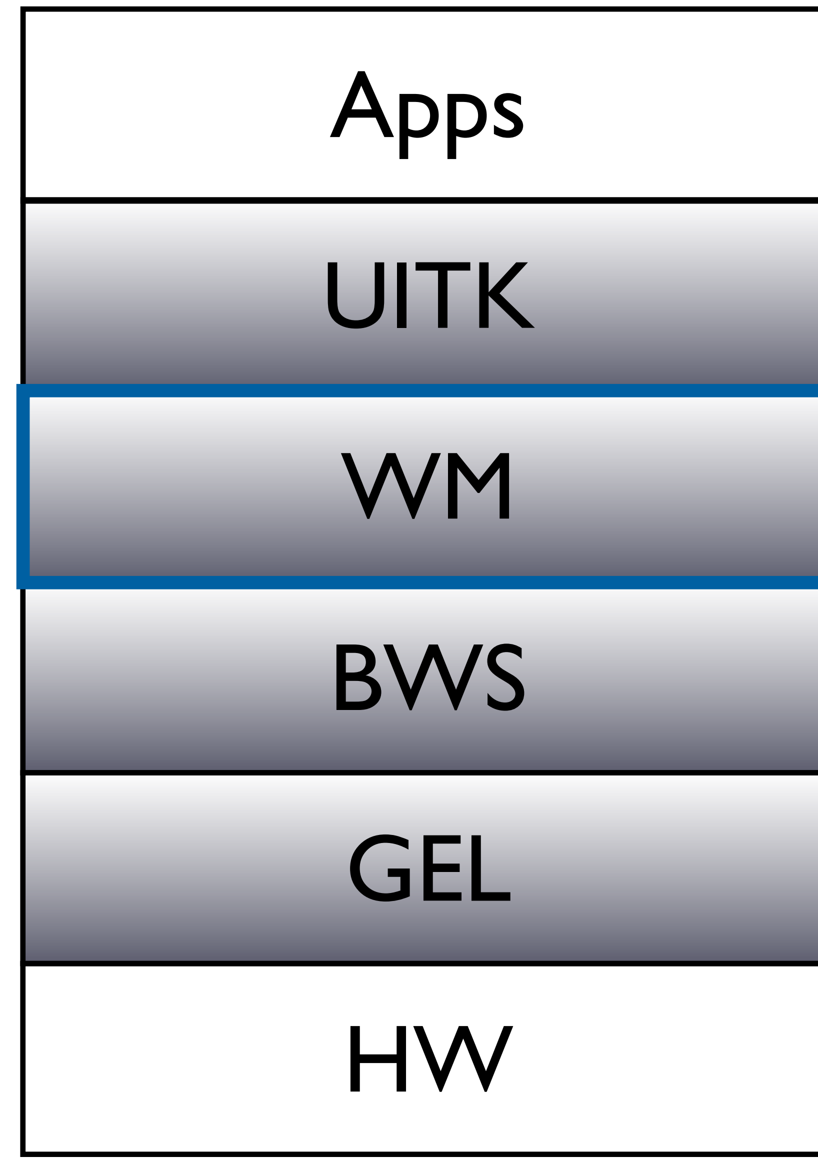


- User Interface Toolkit



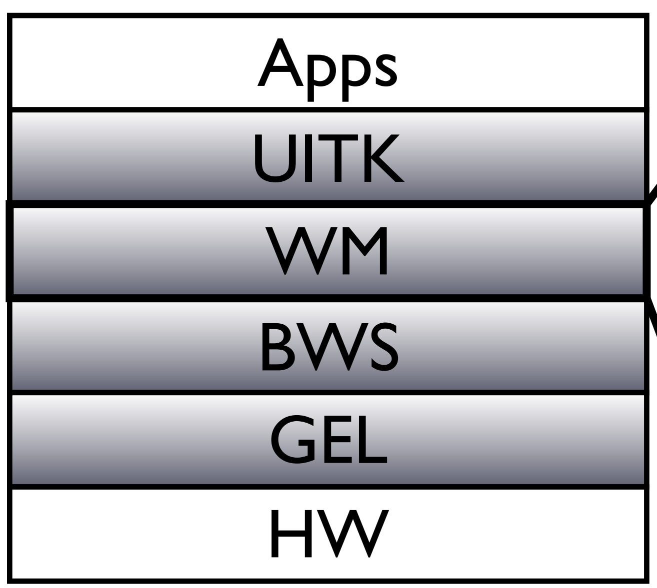
Window Manager: Motivation

- Position and decorate windows
- Provide Look&Feel for interaction with WS
- So far: applications can output to windows
  - User control defined by application
  - May result in inhomogeneous user experience
- Now: let user control windows
  - Independent of applications
  - User-centered system view
- BWS provides mechanism vs. WM implements policy



Window Manager: Structure

Application-independent user interface



Appearance (“Look”)	Behavior (“Feel”)	Look & Feel
Tiling, Overlapping,...	Pop-up menu at click	Techniques
Request position change,...	Fetch events	Communicate with BWS

Screen Management

- What is rendered where on screen? (layout question)
- Where empty space? What apps iconified? (practical q's)
- Example: Negotiating window position
    - Application requests window at (x,y) on screen;
      ignores position afterwards by using window coordinate system
    - BWS needs to know window position at any time to handle coordinate transformation, event routing, etc. (manages w)
    - User wishes to move window to different position
    - Or: Requested position is taken by another window
- Three competing instances (same for color tables,...)
- Solution: *Priorities,* for example:
    - Prior (app) < Prior (WM) < Prior (user)
    - WM as advising instance, user has last word

Session Management

- Certain tasks are needed for all apps in consistent way
  - Move window, start app, iconify window
- Techniques WM uses for these tasks
  - Menu techniques
    - Fixed bar+pull-down (Mac), pop-up+cascades (Motif),...
  - Window borders
    - Created by WM, visible/hidden menus, buttons to iconify/maximize, title bar

Session Management

- WM techniques continued
  - Direct manipulation
    - Manipulate onscreen object with real time feedback
    - Drag & drop,...
    - Early systems included file (desktop) manager in window manager; today separate "standard" application (Finder,...)
  - Icon technique: (de)iconifying app windows
  - Layout policy: tiling, overlapping
    - Studies showed tiling WM policy leads to more time users spend rearranging windows

Session Management

- WM techniques continued
  - Input focus: Various modes possible
    - Implicit (focus follows pointer): mouse/kbd/... input goes to window under specific cursor (usually mouse)
    - Explicit (click to type): clicking into window activates it (predominant mode today)
  - Virtual screens
    - Space for windows larger than visible screen
    - Mapping of screen into space discrete or continuous

Session Management

- WM techniques continued
  - Look & Feel evolves hand-in-hand with technology
    - Audio, video I/O
    - Gesture recognition
    - 2.5-D windows (implemented by WM, BWS doesn't know)
    - Transparency
  - To consider:
    - Performance hit?
    - Just beautified, or functionally improved?

Late Refinement

- WM accompanies session, allows user to change window positions, etc. (changing app appearance)
- For this, application must support late refinement
  - App developer provides defaults that can be changed by user
  - Attributes must be publicized as configurable, with possible values
  - App can configure itself using startup files (may be inconsistent), or WM can provide those values when starting app
  - With several competing instances: priorities (static/dynamic!...)

Levels of Late Refinement

- Per session, for all users
    - System-wide information (table, config file,...) read by WM
- Per application, for all users
    - Description for each application, in system-wide area
- Per application, per user
    - Description file for each application, stored in home directory
- Per application, per launch
    - Using startup parameters (options) or by specifying specific other description file

Implementing Late Refinement

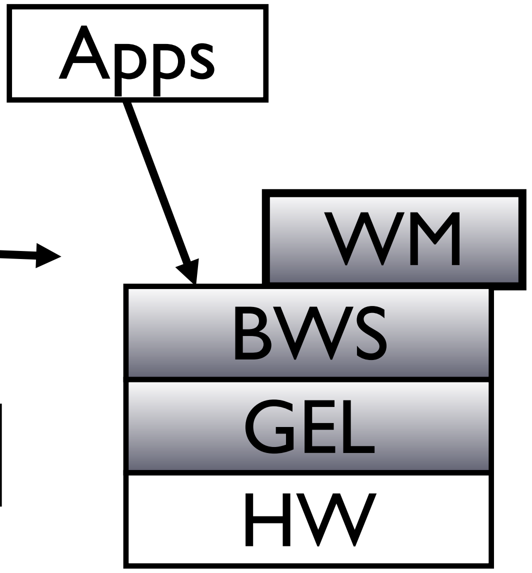
- Table files
  - Key-value pairs, with priority rule for competing entries
  - Usually clear text (good idea), user versions usually editable
  - Modern versions: XML-based
- WM-internal database
  - Access only via special editor programs
  - Allows for syntax check before accepting changes, but less transparent; needs updating when users are deleted,.....
  - *Random Rant: Why Non-Clear-Text Config Files Are Evil*
- Delta technique
  - Starting state + incremental changes; undo possible

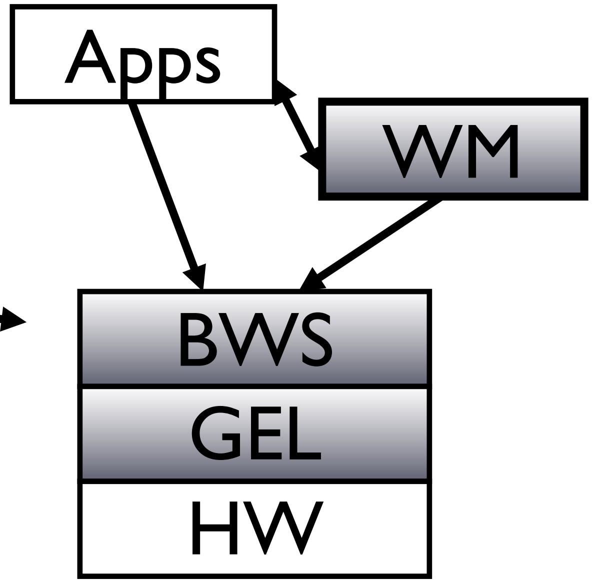
Example: plist for login window application (Mac OS X)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://
  www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>PicturePathLW</key>
  <string>/Library/User Pictures/Flowers/Sunflower.tif</string>
  <key>RetriesUntilHint</key>
  <integer>3</integer>
  <key>lastUserName</key>
  <string>borchers</string>
  <key>lightWeightLogin</key>
  <false/></dict>
</plist>
```

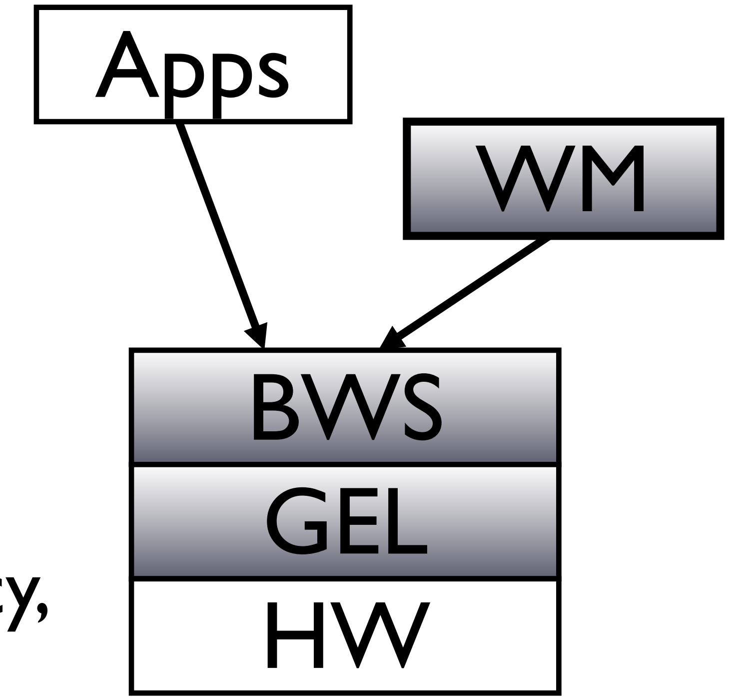
Window Manager: Location

- WM=client of BWS, using its access functions
- WM=server of apps, can change their appearance
- Several possible architectures
  - WM as upper part of BWS
    - Saves comms overhead
    - But overview suffers
  - WM as separate server
    - More comms
    - But exchangeable WM





Window Manager: Location



- Separate user process
  - Uses mechanism of shared resources
  - E.g., requests window position from BWS, checks its conformance with its layout policy, and requests position change if necessary
  - More comms, but same protocol as between apps & BWS; no direct connection app—WM

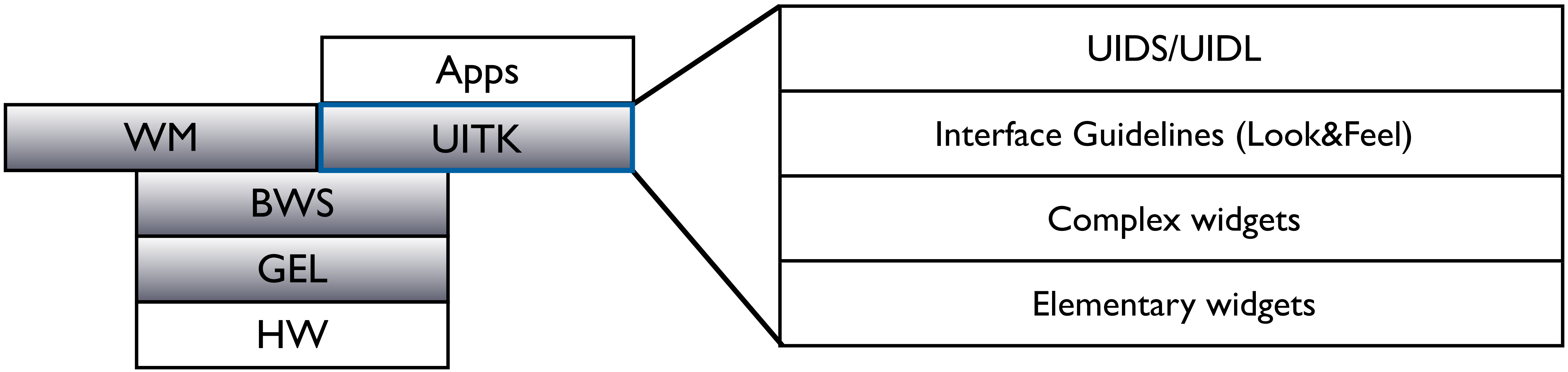
Window Manager: Conventions

- Visual consistency
  - For coding graphical information across apps
  - Reduce learning effort
- Behavioral consistency
  - Central actions tied to the same mouse/kbd actions (right-click for context menu, Cmd-Q to quit) - predictability
- Description consistency
  - Syntax & semantics of configuration files / databases consistent across all levels of late refinement
  - Usually requires defining special language

Window Manager: Conclusions

- WM leads from system- to user-centered view of WS
- Accompanies user during session
- Potentially exchangeable
  - Allows for implementation of new variants of desktop metaphor without having to change entire system
  - E.g., still much room for user modeling (see, e.g., *IUI 2002*)
- WM requires UI Toolkit to implement same Look&Feel across applications

User Interface Toolkit



- Motivation: Deliver API
  - problem/user-oriented instead of hardware/BWS-specific
  - 50–70% of SW development go into UI
    - UITK should increase productivity

UITK: Concept

- Two parts
  - Widget set (closely connected to WS)
  - UIDS (User Interface Design System to support UI design task)
- Assumptions
  - UIs decomposable into sequence of dialogs (time) using widgets arranged on screen (space)
  - All widgets are suitable for on-screen display (no post-desktop user interfaces)
  - Note: decomposition not unique

UITK: Structure

- Constraints
  - User works on several tasks in parallel → parallel apps
  - Widgets need to be composable, and communicate with other widgets
  - Apps using widget set (or defining new widgets) should be reusable
- Structure of procedural/functional UITKs
  - Matched procedural languages and FSM-based, linear description of app behavior
  - But: Apps not very reusable

UITK: Structure

- OO Toolkits
  - Widget handles certain UI actions in its methods, without involving app
  - Only user input not defined for widget is passed on to app asynchronously (as seen from the app developer)
    - Matches parallel view of external control, objects have their own "life"
  - Advantage: Subclass new widgets from existing ones
  - Disadvantage:
    - Requires OO language (or difficult bridging, see Motif)
    - Debugging apps difficult

UITK: Control Flow

- Procedural model:
  - App needs to call UITK routines with parameters
  - Control then remains in UITK until it returns it to app
- OO model:
  - App instantiates widgets
  - UITK then takes over, passing events to widgets in its own event loop
  - App-specific functionality executed asynchronously in *callbacks* (registered with widgets upon instantiation)
  - Control flow also needed between widgets

Defining Widgets

- Widget :

$$(W = (w_1 \ldots w_k), G = (g_1 \ldots g_l), A = (a_1 \ldots a_m), i = (i_1 \ldots i_n))$$

  - Output side: windows W, graphical attributes G
  - Input side: actions A that react to user inputs I
  - Mapping inputs to actions is part of the specification, can change even at runtime
  - Actions can be defined by widget or in callback
- Each widget type satisfied a certain UI need
  - Input number, select item from list,...

Simple Widgets

- Elementary widgets
  - Universal, app-independent, for basic UI needs
  - E.g., button (trigger action by clicking), label (display text), menu (select 1 of n commands), scrollbar (continuous display and change of value), radio button (select 1 of n attributes)

In-Class Exercise: Button

- What are the typical components (W, G, A, I) of a button?
- Sample solution:
  - W=(text window, shadow window)
  - G=(size, color, font, shadow,...)
  - A=(enter callback, leave callback, clicked callback)
  - I=(triggered with mouse, triggered with key, enter, leave)

Simple Widgets

- Container widgets
  - Layout and coordinate other widgets
  - Specification includes list C of child widgets they manage
  - Several types depending on layout strategy
- Elementary & Container widgets are enough to create applications and ensure look&feel on a fundamental level

Complex Widgets

- Applications will only use subset of simple widgets
- But also have recurring need for certain widget combinations depending on app class (text editing, CAD,...)
  - Examples: file browser, text editing window
- Two ways to create complex widgets
  - Composition (combining simple widgets)
  - Refinement (subclassing and extending simple widgets)
  - Analogy in IC design: component groups vs. specialized ICs

Widget Composition

- Creating dynamic widget hierarchy by hierarchically organizing widgets into the UI of an application
  - Some will not be visible in the UI
- Starting at root of dynamic widget tree, add container and other widgets to build entire tree
  - Active widgets usually leaves
  - Dynamic because it is created at runtime
  - Can even change at runtime through user action (menus,...)

Widgets and Windows

- The dynamic widget tree usually matches geographical *contains* relation of associated BWS windows
- But: Each widget usually consists of several BWS windows

→ Each widget corresponds to a subtree of the BWS window tree!

→ Actions A of a widget apply to is entire geometric range except where covered by child widgets

→ Graphical characteristics G of a widget are handled using priorities between it, its children, siblings, and parent

Refinement of Widgets

- Create new widget type by refining existing type
- Refined widget has mostly the same API as base widget, but additional or changed features, and fulfills Style Guide
- Not offered by all toolkits, but most OO ones
- Refinement creates the Static Hierarchy of widget subclasses
- Example: Refining text widget to support styled text (changes mostly G), or hypertext (also affects I & A)

Late Refinement of Widgets

- App developer can compose widgets
- Widget developer can refine widgets
- → User needs way to change widgets
- → Should be implemented inside toolkit
- Solution: Late Refinement (see WM for discussion)
- Late refinement cannot add or change type of widget characteristics or the dynamic hierarchy
- But can change values of widget characteristics

Style Guidelines

- How support consistent Look&Feel?
  - Document guidelines, rely on developer discipline
    - E.g., Macintosh Human Interface Guidelines (but included commercial pressure from Apple & later user community)
  - Limiting refinement and composition possible
    - Containers control all aspects of Look&Feel
    - Sacrifices flexibility
  - UIDS
    - Tools to specify the dialog explicitly with computer support

Types of UIDS

- Language-oriented
  - Special language (UIL) specifies composition of widgets
  - Compiler/interpreter implements style guidelines by checking constructs
- Interactive
  - Complex drawing programs to define look of UI
  - Specifying UI feel much more difficult graphically
    - Usually via lines/graphs connecting user input (I) to actions (A), as far as allowed by style guide
- Automatic
  - Create UI automatically from spec of app logic (research)
- *Examples in upcoming lectures*