User Interface Architectures

- Why study UI Architecture?
- Constructing usable interfaces is difficult
- If whole interactive system has to be substantially rewritten each time a change is made, then hard to develop high quality interfaces.
- Better constructed code -> reduce impact of small changes to interface.

Software architectures

- A method by which software systems are decomposed into components and a specification of how those components interact.
- User interfaces are problematic in this respect.

UI Architectures

- Tools shield designer from underlying complexity.
- Fast development of interfaces-> more iterations -> better designs.
- Can save money with faster development.
- Tools can also be used to address issues like multi-platform support etc.

Separation of concerns

- A major issue is separation between the semantics of the application and the interface provided for the user to make use of that semantics.
- Portability – To allow the same application to be used on different systems.
- Reusability - Separation increases likelihood components can be reused.
- Multiple Interfaces - Several interfaces, same functionality.
- Customisation – by both the designer and the user, without altering underlying application.

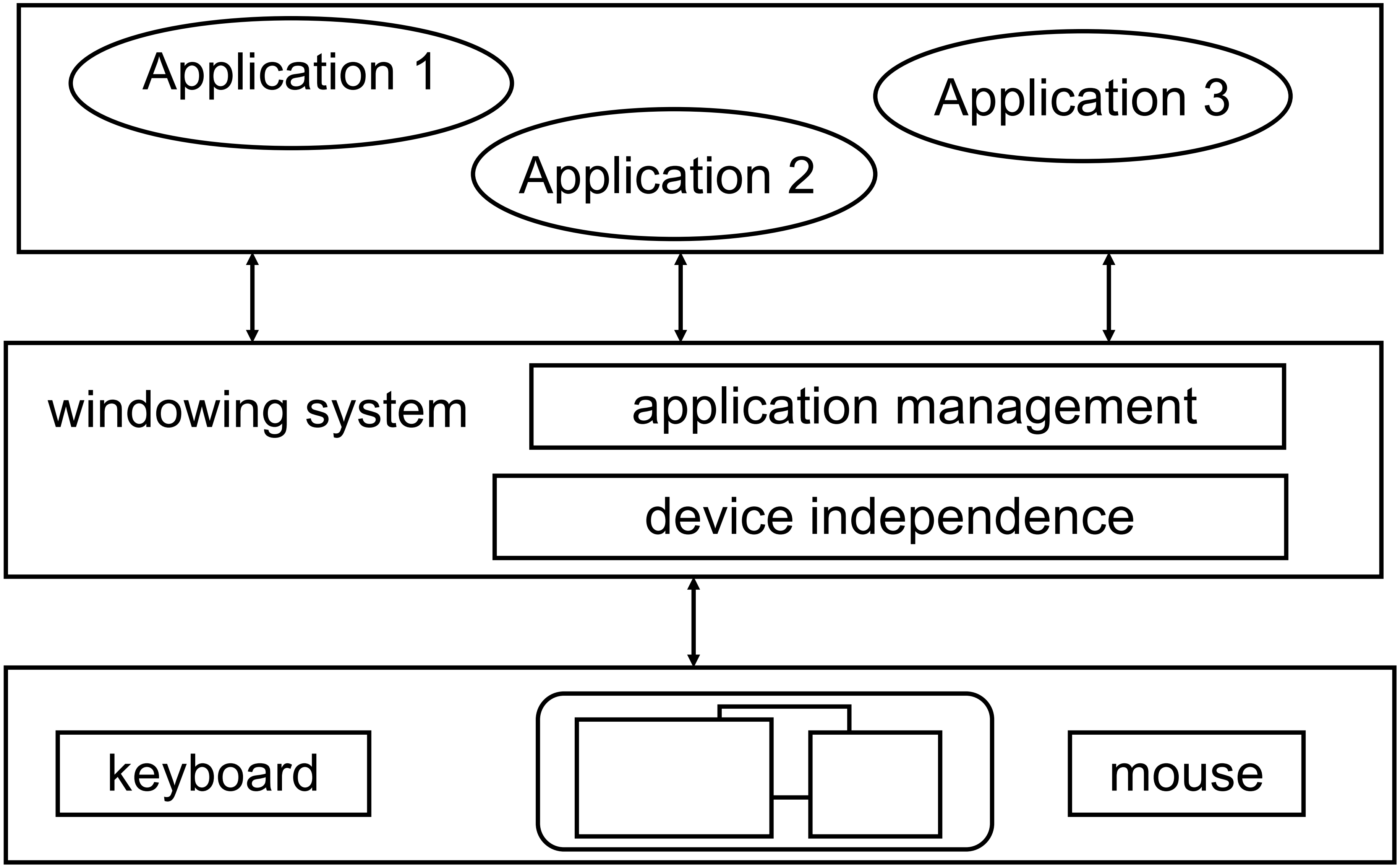
Windowing Systems

- WS provide the tools to build the UI
- Abstract terminal:
  - WS models an abstract terminal
  - Accepts commands
  - Translates them into commands for the specific system in use.
- Portability
  - Use the same window system on another machine.
  - Only need device drivers for new machine and software will run.

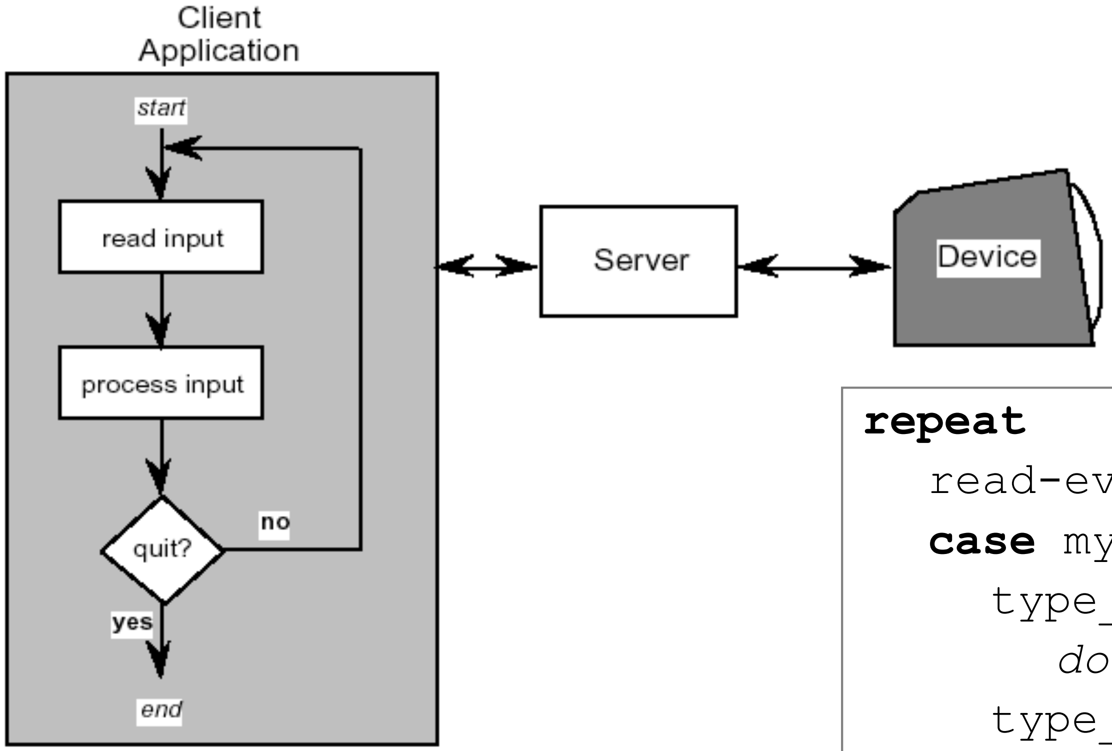
Windowing Systems

- Can run multiple abstract terminals
  - "windows" in the conventional sense
  - WS manages screen/input for each window
  - Simplifies programming.
- Windowing System provides
  - Device independence
  - Management
  - Isolation of individual applications

Windowing System



Read-evaluation loop



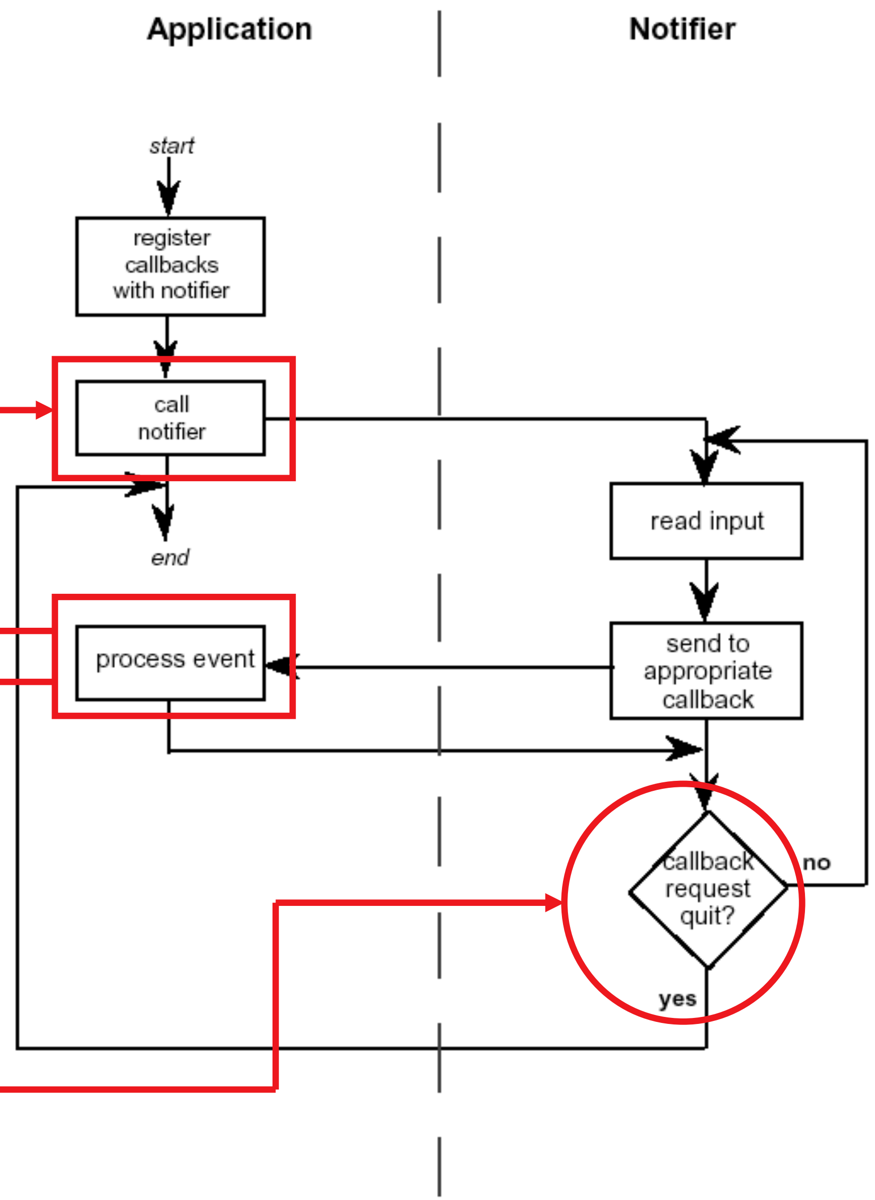
```
repeat
   read-event(myevent)
   case myevent.type
      type_1:
         do type_1 processing
      type_2:
         do type_2 processing
      ...
      type_n:
         do type_n processing
   end case
end repeat
```

Notifiers

```
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
}
```



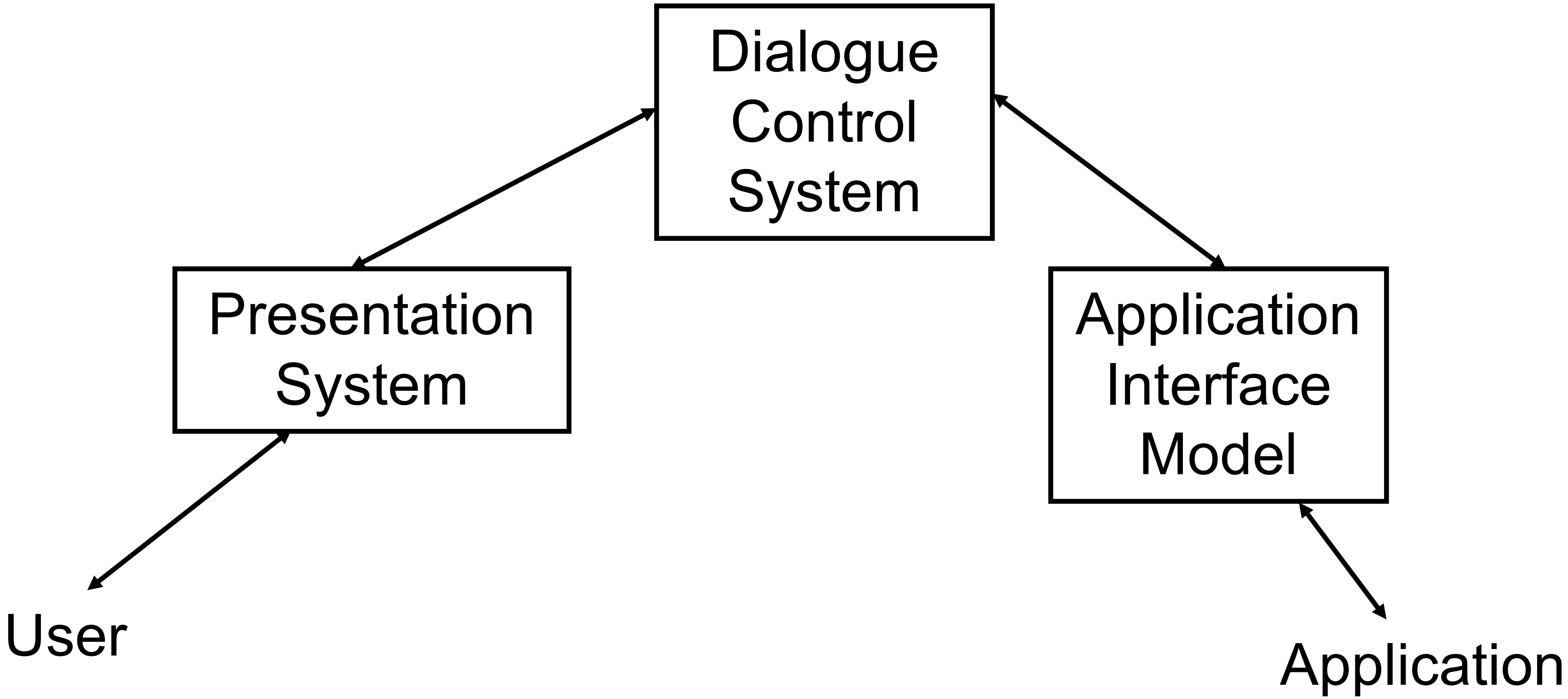
Separating User Interface from Application

- One aim of a UI architecture is to separate the interface from the application.
- How are UI and application connected?
  - Dialogue controller

Seeheim model of UI Architecture

- Classic UI system architecture
- UI system made up of 3 components
  - Presentation system
  - Dialogue control system
  - Application interface model.
- Presentation system
  - Translates between external physical representation and internal logical representation.
  - Generates images on the display
  - Reads data from input devices and converts this raw data into form for dialogue control component.

Seeheim model



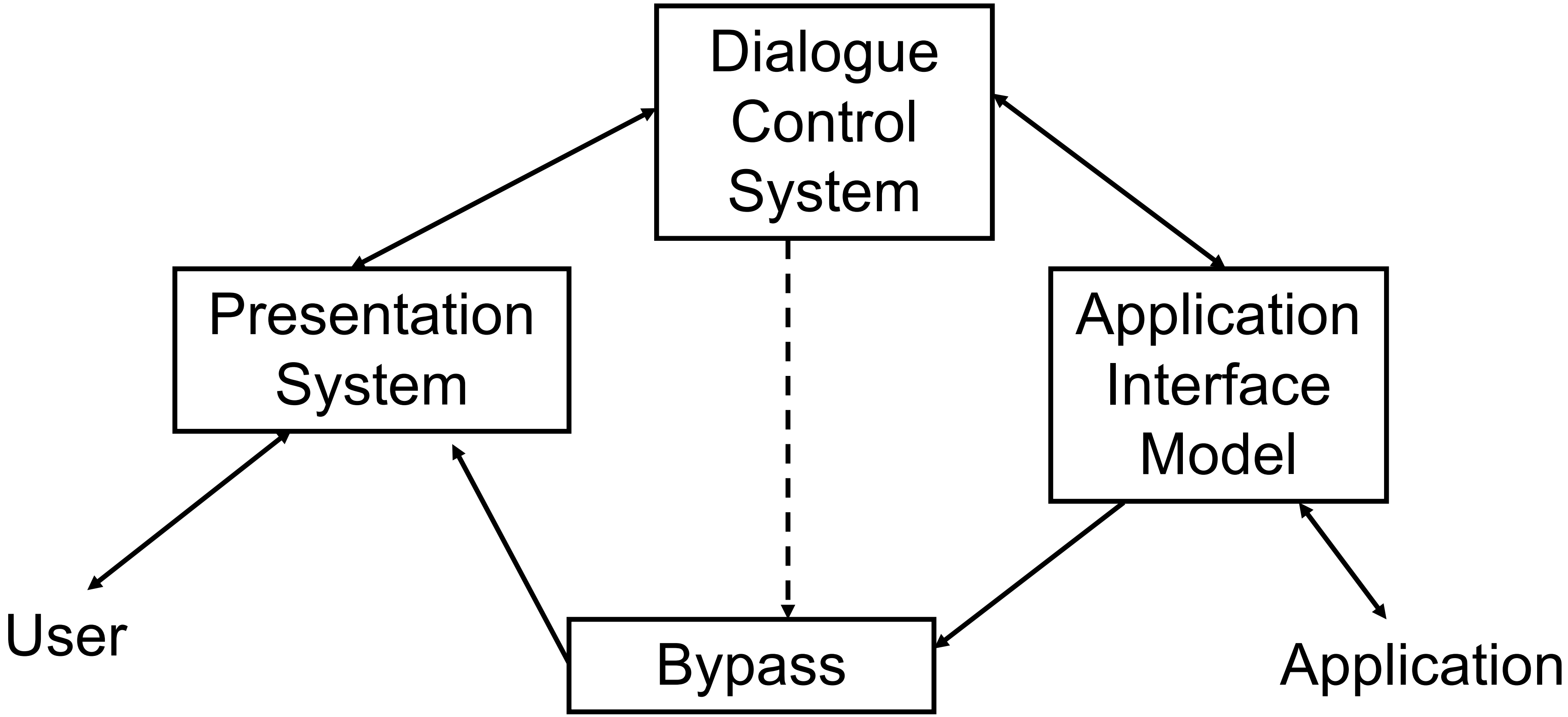
Seeheim components

- Dialogue control system
  - Defines the structure of the dialogue between user and application.
  - Accepts input from presentation system and from application (data to be displayed or data requests) and routes to appropriate destination.
- Application Interface Model - a representation of the application from the user interface’s point of view
  - Specification of application-significant objects
  - Specification of application operations
  - Mapping from objects and operations in interface to actual application data and routines.

Seeheim Architecture - Disadvantages

- Disadvantages
  - Poor for handling multi-threaded interaction (where several separate commands may be active at one time)
  - Poor for handling low-level “semantic” feedback (e.g. dragging a file icon over the wastebasket).
  - Each component (presentation, control and application interface model) is monolithic – more difficult to change.
  - Need to address building large systems from small components.

Seeheim model



Implementation models

- Modern interfaces tend to be collections of relatively independent agents
- The UI of an application is subject to many changes
  - Change of UI for different users
  - Same info can be shown in different windows
  - Changes to underlying data should be reflected quickly everywhere
  - Changes to UI should be easy, even at runtime
  - Different "look and feel" should not affect functional core

Match with object orientation

- Each “object of interest” is separate; e.g. a button
  - produces “button-like” output
  - acts on input in a “button-like” way
  - etc.
- Each object does its tasks based on
  - What it is
  - What its current “state” is
    - Context from prior interaction or application

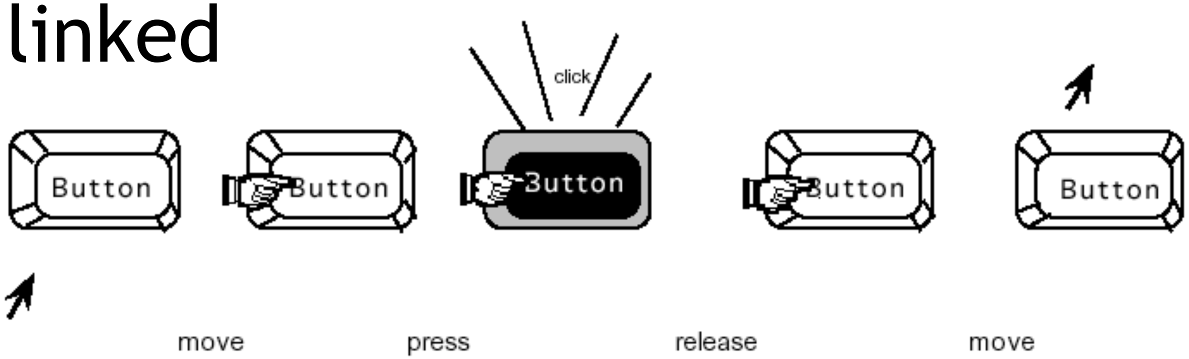
Object based architectures.

- Object-based (interactor) solutions.
- Objects organised hierarchically
  - Normally reflecting spatial containment relationships
  - Get trees of interactors.
- Still want to minimise complexity of individual objects, maximise separation of concerns.
- Options:
  - Containers
  - Inheritance.
  - Aggregation (MVC).

Toolkits

Interaction objects

- input and output intrinsically linked



Toolkits provide this level of abstraction

- programming with interaction objects (widgets, gadgets)
- promote consistency and generalizability through similar look and feel
- amenable to object-oriented programming

Containers

- Put together interaction objects at larger scale than interactors
- Container objects
  - e.g., row and column layout objects
- Containers can also add input & output behavior to things they contain

Interactor level - inheritance

- Inheritance
  - all concerns in one object
  - inherit / override them separately
  - works best with multiple inheritance
  - example: draggable_icon
    - inherit appearance from “icon”
      - output aspects only
    - inherit behavior from “draggable”
      - input aspects only
- Multiple inheritance problematic.

Aggregation

- Different concerns in separate objects
  - Treat collection as “the interactor”
  - Slice up Seeheim
- Changes are easier if we separate input, output and processing.
- Classic architecture: “model-view-controller” (MVC)
  - from Smalltalk 80

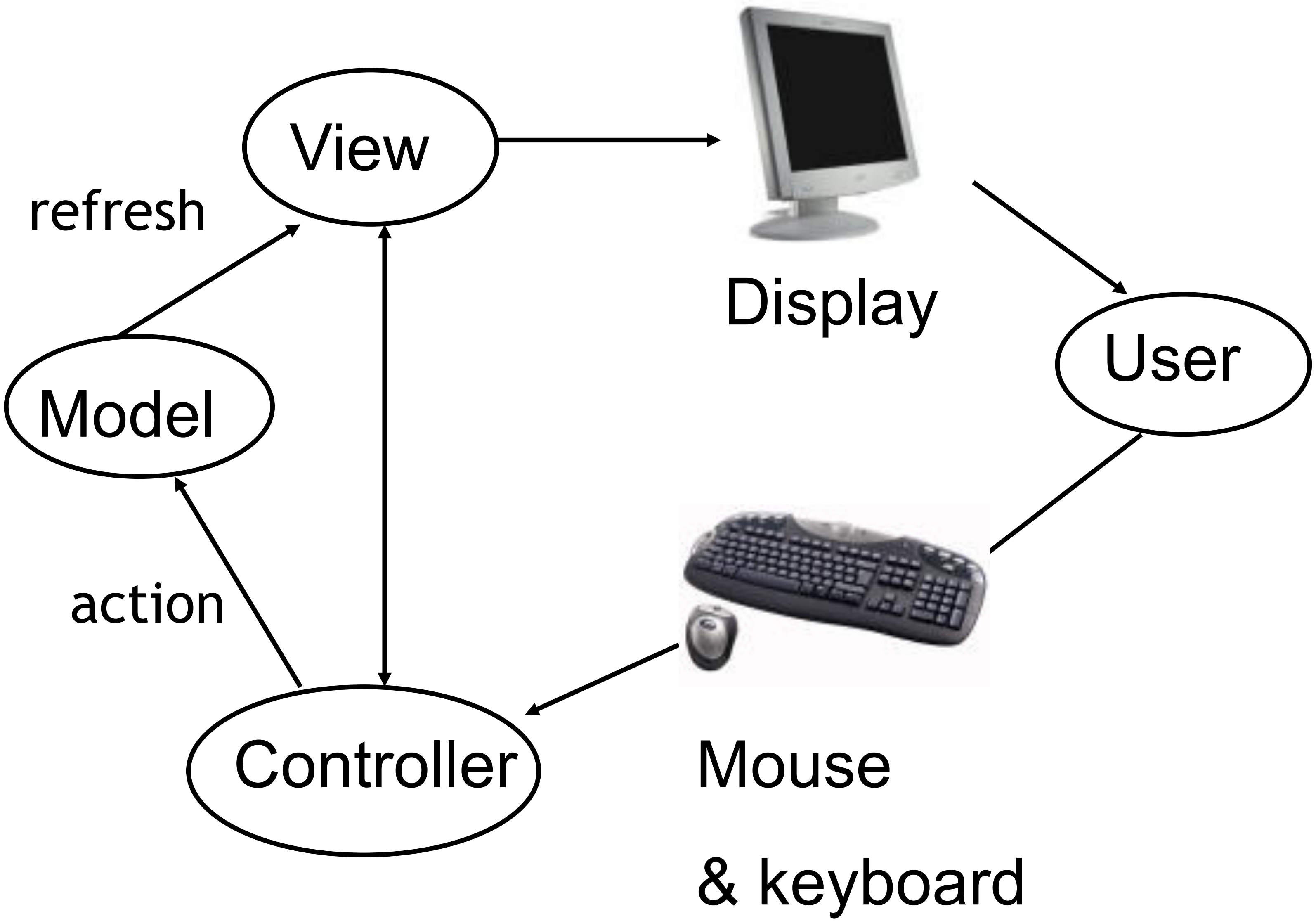
The Model View Controller UI Architecture

- Architectural design pattern that separates the User Interface from the application.
- Interaction model for Smalltalk – one of the earliest successful OO systems.
  - Variants of pattern used in Java, Ruby on Rails, Django and many others.
- MVC organises interactive systems into a collection of interaction objects made up of Model-View-Controller triples.

MVC

- Model-View-Controller triples.
  - The **Model** is any object (application).
  - The **View** is an object which provides a visual representation of a model (output).
  - The **Controller** is an object which handles input actions, sending messages to the view or model, as appropriate (input).
- Views and Controllers comprise UI
- *Change-propagation mechanism* ensures consistency between Model and UI.

MVC architecture



Model

- Encapsulates application-specific data and functionality, providing:
  - methods to update data, which Controller can call
  - methods to access state, which View and Controller can request
- Maintains registry of dependent *Views* and *Controllers* to be notified about data changes
- Examples:
  - text editor: model is text string
  - slider: model is an integer
  - spreadsheet: collection of values related by functional constraints

View

- Mechanism needed to map model data to presentation (view / display)
- When Model changes, View is informed
  - View requests relevant model information
  - View arranges to update screen
- Examples:
  - Slider: rectangular button in rectangular box, line with bead, radial gauge
  - Spreadsheet:
    - Tabular representation
    - Bar chart
    - Histogram

Controller

- Accepts user input events
- Translates events into methods invoked on Model or changes the view.
- Activates/Deactivates UI elements (graying)
- Examples:
  - Textual commands
  - Mouse (point and click) commands

MVC Architecture

- The link between application and presentation built up of MVC units.
- In addition to displaying aspects of its model, a view may contain sub-views.
- Models, views and controllers are part of object hierarchy so can be inherited and modified.
- Single model can be associated with different views and controllers – same app different interface (each V-C pair only associated with one M) – eg. spreadsheet with multiple graphs of same data.

Communication between the M,V,C

- The model, view and controller communicate via dependencies (listeners in Java).
- A listener is set up to listen to changes in the controller, for example
  - When an *event* occurs the listener is notified
  - Only the listeners interested in the event are told.

Advantages of MVC

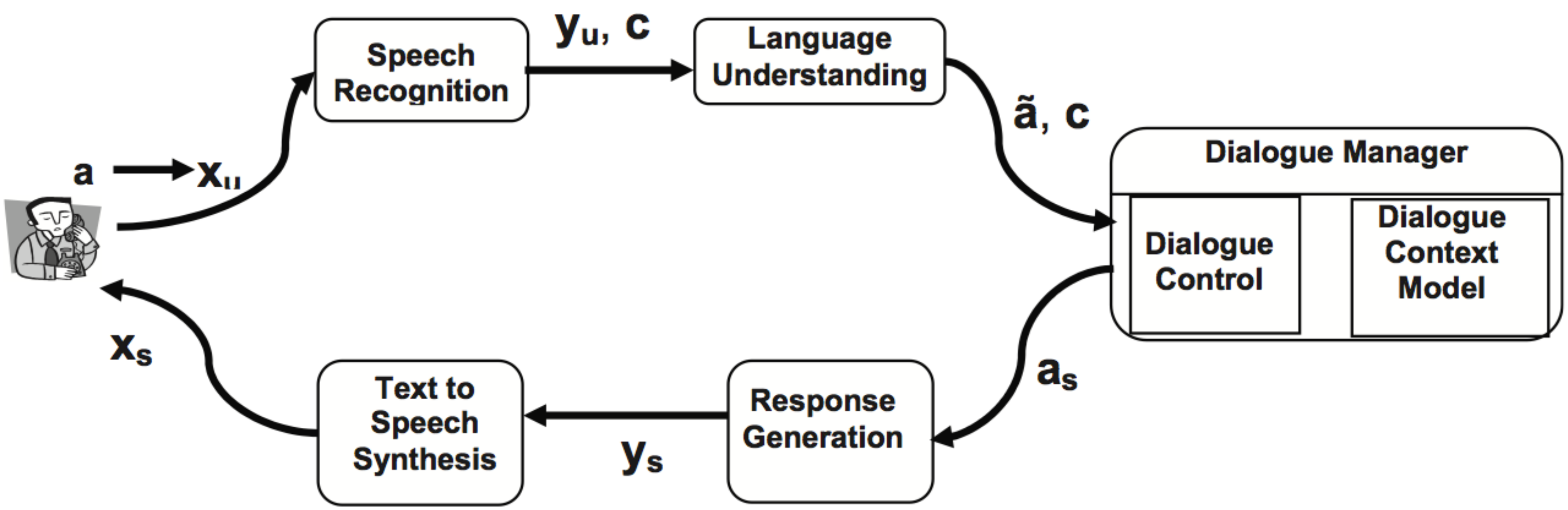
- Seeheim doesn’t deal with building complex interactive systems from small components (S.E.).
  - The MVC approach distributes the handling of presentation, control and application linkage.
- Each MVC triple is organised around a part of the display (the area for which the view has responsibility).
- Model updates its views directly – easier to handle
  - Immediate “semantic” feedback
  - Interleaving of tasks
  - Implementation of ***direct manipulation*** style interfaces
  - Multiple views of same model.

MVC Problems

- Dependency mechanism may lead to a spaghetti network of links
  - Difficult to debug
- Model is not well-developed
  - No notion of interface-application separation; the application consists of one or more model objects;
  - These are linked directly to the interface components (view and controller).
  - Designed for one user
  - Designed for one system
  - Complexity for simple interactors.
  - Potentially excessive updates/messages.

Spoken Dialog System Architecture

(From McTear)



$\mathbf{x_u}$ – user acoustic signal
$\mathbf{y_u}$ – speech recognition hypothesis (words)
a – user dialogue act (intended)
ã – user dialogue act (interpreted)

$\mathbf{a_s}$ – system dialogue act
$\mathbf{y_s}$ – system word string
$\mathbf{x_s}$ – system acoustic signal