University of California, Irvine

Informatics 134

Software User Interfaces
Winter 2022

Mark S. Baldwin
baldwinm@ics.uci.edu

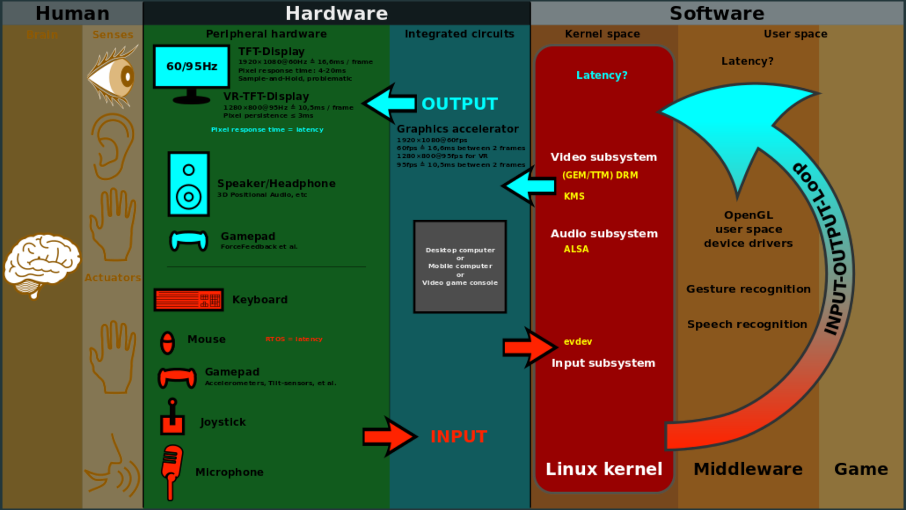
1/13/2022

Agenda

User Interface Architecture

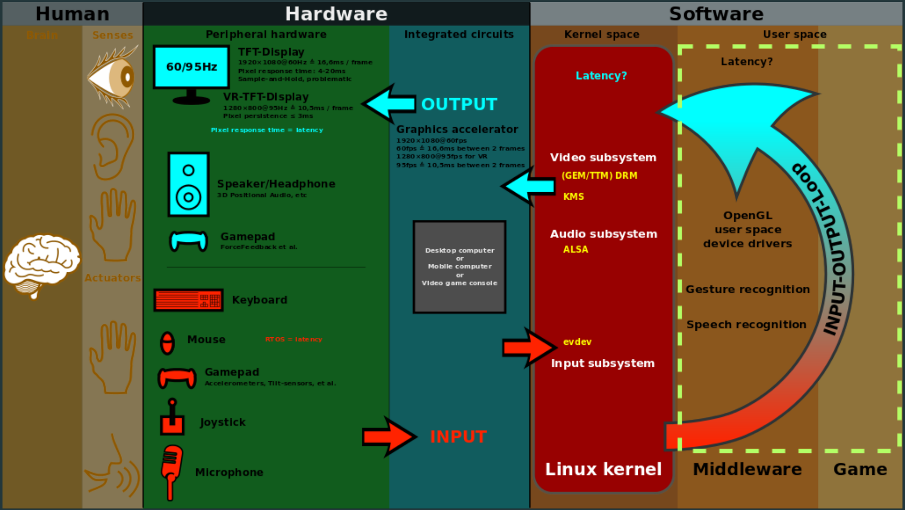
User Interface Architecture

User Interface Architecture



[Wikipedia, 2021a]

User Interface Architecture

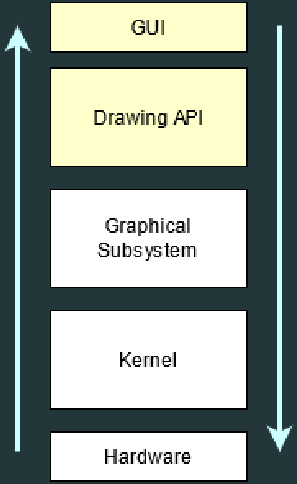


[Wikipedia, 2021a]

User Interface Architecture

User Interfaces from an Architectural Level

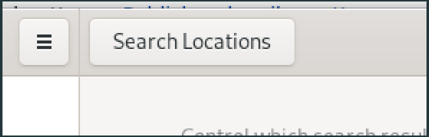
- GUIs rely on many different units of code to function
- Data propagates between these units to represent state and interaction
- Each unit is responsible for making decisions on how to handle a particular operation



User Interface Architecture

The Button Example

What are some observations that we can make about its functionality?



User Interface Architecture

The Button Example

- Clickable
- Can visually change in response to interaction
- Can display data
- Can execute a command

User Interface Architecture

The Button Example

In computer science, these observations are represented by a state chart and implemented through a state machine.

User Interface Architecture

The Button Example

In computer science, these observations are represented by a state chart and implemented through a state machine.

Let’s revisit:

- Clickable
- Can visually change in response to interaction
- Can display data
- Can execute a command

Button State Chart

How would you complete the table?

Current State	Transition	Present State
cs-1	t-1	ps-1
cs-2	t-2	ps-2
cs-3	t-3	ps-3

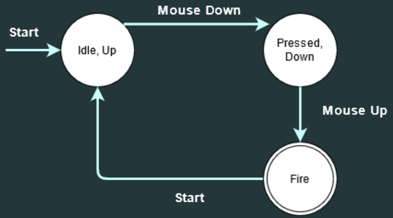
Button State Chart

Current State	Transition	Present State
Idle	Mouse Down	Pressed
Pressed	Mouse Up	Execute
Execute	Mouse Up	Idle

User Interface Architecture

Button State Chart

The simple button example represented using a state chart diagram



User Interface Architecture

The Button Example

Although this simple button example could work, most buttons (and other widgets) are typically far more complex.

What are some other states we might need to support in a fully featured button?

DEMO

User Interface Architecture

Tiny widgets filled with tiny state machines

Let’s revisit our earlier observations...

- Clickable
- Can visually change in response to interaction
- Display data
- Can execute a command

Can you think of any architectures that might bring these widgets together?

User Interface Architecture

Tiny Widgets...MVC

The Model-View-Controller paradigm is the dominant way to represent groups of widgets.

Controller	Clickable
View	Can visually change in response to interaction
Model	Display data
Controller	Can execute a command

User Interface Architecture

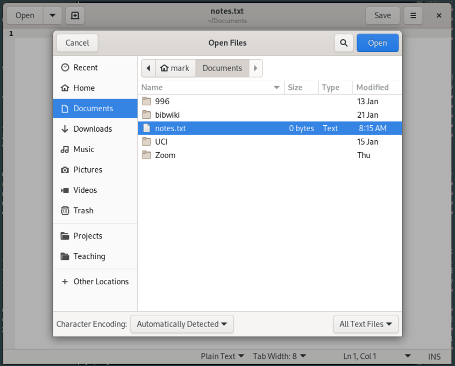
MVC Refresher

Model	View	Controller
Manages data to be presented by the GUI	Visualizes the data stored in the model	Handles user input, model data, and updates

User Interface Architecture

Model…View…Controller

How would we represent the GUI pictured here using an MVC architecture



Name	Size	Type	Modified
996			13 Jan
bibwiki			21 Jan
notes.txt	0 bytes	Text	8:15 AM
UCI			15 Jan
Zoom			Thu

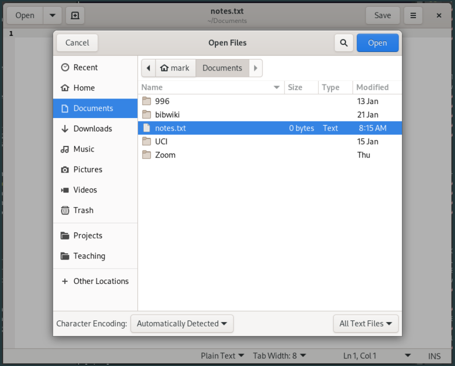
Character Encoding: Automatically Detected

All Text Files

Plain Text Tab Width: 8 Ln 1, Col 1 INS

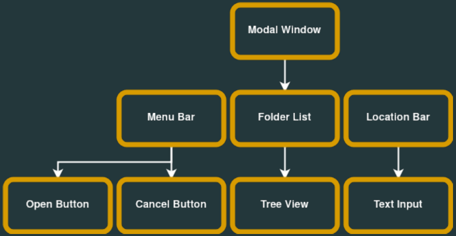
User Interface Architecture

But there's something else interesting about this GUI...



User Interface Architecture

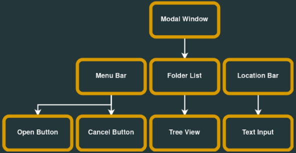
But there’s something else interesting about this GUI...

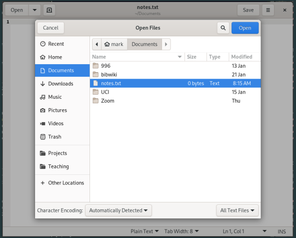


User Interface Architecture

GUIs are structured hierarchically

- Some widgets can contain other widgets
- Container widgets are not always visible
- Hierarchical composition supports layout and communication between widgets





Name	Size	Type	Modified
996			13 Jan
bibwiki			21 Jan
notes.txt	0 bytes	Text	8:15 AM
UCI			15 Jan
Zoom			Thu

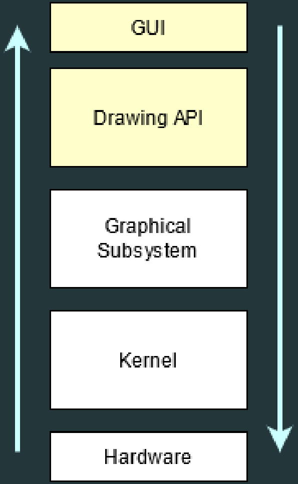
Character Encoding: Automatically Detected

All Text Files

User Interface Architecture

Hierarchical Composition

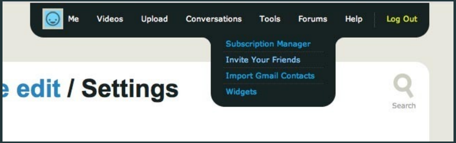
- Layout managers
- Event handling and propagation

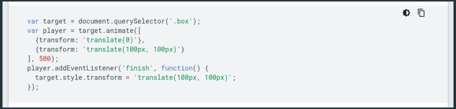


User Interface Architecture

UI's are hard to implement...

- From a design perspective
- From a programming perspective



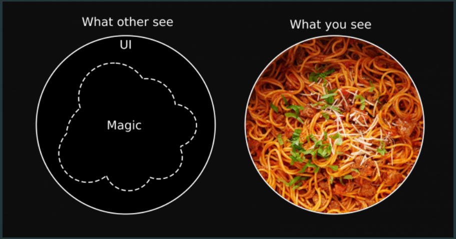


```
var target = document.querySelector('.box');
var player = target.animate([
  {transform: 'translate(0)'},
  {transform: 'translate(100px, 100px)'}
], 500);
player.addEventListener('finish', function() {
  target.style.transform = 'translate(100px, 100px)';
});
```

User Interface Architecture

From a programming perspective

- Reactive, must respond to difficult to predict human behavior
- Event-based, difficult to model **and** modularize
- Dependent on multi-processing (peripherals, displays, local/remote communication)



User Interface Architecture

From a programming perspective

Must be robust enough to handle:

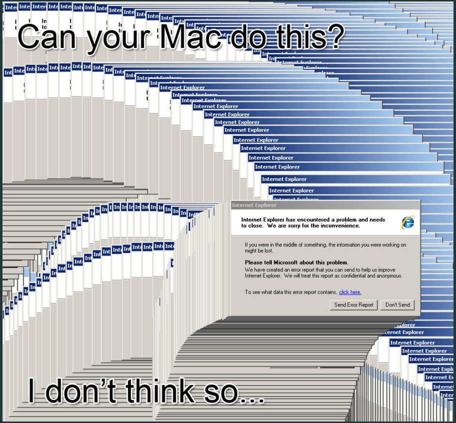
- Device input
- Video and audio
- Background processes

User Interface Architecture

From a programming perspective

Must be robust enough to:

- Avoid crashes
- Support recovery (help, rollback/undo, escape/abort)



Internet Explorer has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to help us improve Internet Explorer. We will treat this report as confidential and anonymous.

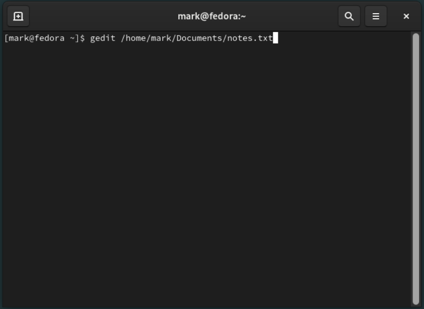
To see what data this error report contains, click here.

What is going on here?

User Interface Architecture

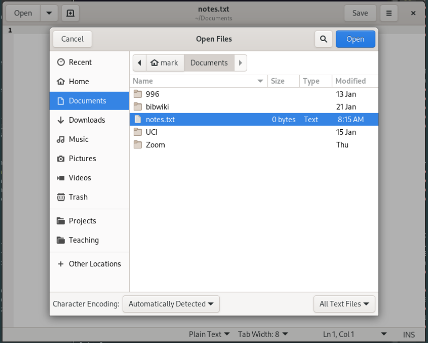
From a programming perspective

Consider the difference between:



```
[mark@fedora ~]$ gedit /home/mark/Documents/notes.txt
```

and:



Name	Size	Type	Modified
996			13 Jan
bibwiki			21 Jan
notes.txt	0 bytes	Text	8:15 AM
UCI			15 Jan
Zoom			Thu

Character Encoding: Automatically Detected

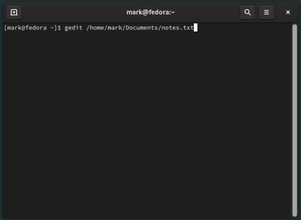
All Text Files

User Interface Architecture

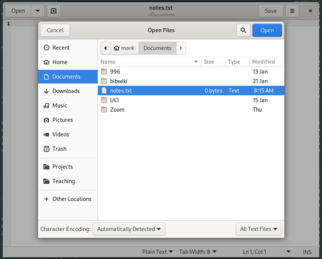
From a programming perspective

Both perform the same action, but the graphical UI must also:

- Support modal
- Cancel (abort/escape)
- Gather and display system resources
- Search
- and many more...



```
[mark@fedora ~]$ gedit /home/mark/Documents/notes.txt
```



Name	Size	Type	Modified
996			13 Jan
bibwiki			21 Jan
notes.txt	0 bytes	Text	8:15 AM
UCI			15 Jan
Zoom			Thu

User Interface Architecture

From a programming perspective

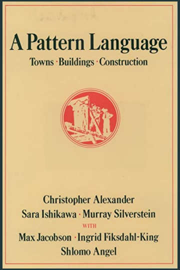
Design patterns, to the rescue?

> Design patterns provide a common language upon which designers and developers can reason about intent and function.

User Interface Architecture

On design patterns

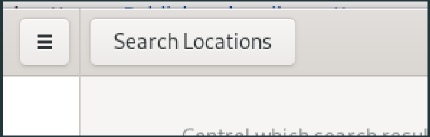
> “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”
>
> ——[Alexander, 1977]


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

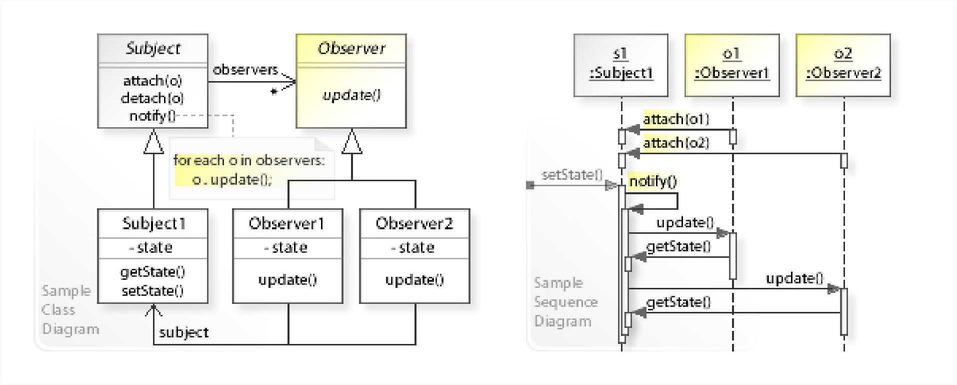
User Interface Architecture

From a programming perspective

UI’s manage complexity through design patterns



The Observer Pattern



[Wikipedia, 2021b]

User Interface Architecture

From a programming perspective

The Observer Pattern

- A standard model for handling event propogation across nearly all UI toolkits

Some examples:

- Microsoft .NET
- TypeScript
- React
- Java

User Interface Architecture

From a programming perspective

When a simple button is filled with so much responsibility...

- Idle state
- Hover state
- Mouse up
- Mouse down
- Pressed
- Released
- Hover up/down?
- Idle down?

We rely on design patterns to manage the complexity

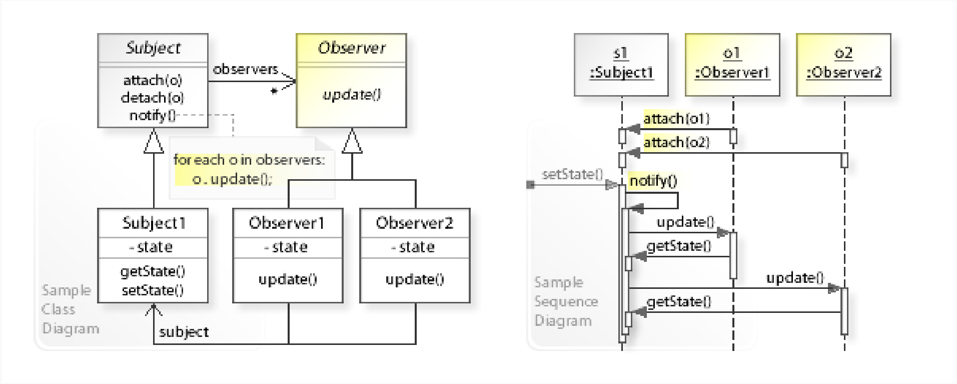
User Interface Architecture

Design patterns are not perfect

As UI complexity grows, design patterns can lead to code that is hard to learn. The observer pattern, for example:

- Promotes side-effects: Since a subject is decoupled from its observer, an event (click, hover) can have *n* observers...
- Difficult to trace control flow and debug

Observer1, Observer2, ObserverN

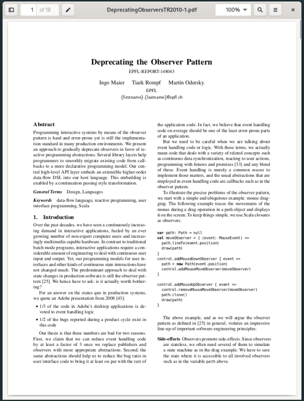


[Wikipedia, 2021b]

User Interface Architecture

Deprecating the Observer Pattern

- Work by Martin Odersky (Scala, Generic Java, many other contributions)
- Via Scala.React system, paradigm shift from observer-based to data-flow based model



Deprecating the Observer Pattern

EPFL-REPORT-148043

Ingo Maier   Tiark Rompf   Martin Odersky
EPFL
{firstname}.{lastname}@epfl.ch

Abstract

Programming interactive systems by means of the observer pattern is hard and error-prone yet is still the implementation standard in many production environments. We present an approach to gradually deprecate observers in favor of reactive programming abstractions. Several library layers help programmers to smoothly migrate existing code from callbacks to a more declarative programming model. Our central high-level API layer embeds an extensible higher-order data-flow DSL into our host language. This embedding is enabled by a continuation passing style transformation.

General Terms Design, Languages

Keywords data-flow language, reactive programming, user interface programming, Scala

1. Introduction

Over the past decades, we have seen a continuously increasing demand in interactive applications, fueled by an ever growing number of non-expert computer users and increasingly multimedia capable hardware. In contrast to traditional batch mode programs, interactive applications require a considerable amount of engineering to deal with continuous user input and output. Yet, our programming models for user interfaces and other kinds of continuous state interactions have not changed much. The predominant approach to deal with state changes in production software is still the observer pattern [25]. We hence have to ask: is it actually worth bothering?

For an answer on the status quo in production systems, we quote an Adobe presentation from 2008 [43]:

- 1/3 of the code in Adobe's desktop applications is devoted to event handling logic
- 1/2 of the bugs reported during a product cycle exist in this code

Our thesis is that these numbers are bad for two reasons. First, we claim that we can reduce event handling code by at least a factor of 3 once we replace publishers and observers with more appropriate abstractions. Second, the same abstractions should help us to reduce the bug ratio in user interface code to bring it at least on par with the rest of the application code. In fact, we believe that event handling code on average should be one of the least error-prone parts of an application.

But we need to be careful when we are talking about event handling code or logic. With these terms, we actually mean code that deals with a variety of related concepts such as continuous data synchronization, reacting to user actions, programming with futures and promises [33] and any blend of these. Event handling is merely a common *means* to implement those matters, and the usual abstractions that are employed in event handling code are callbacks such as in the observer pattern.

To illustrate the precise problems of the observer pattern, we start with a simple and ubiquitous example: mouse dragging. The following example traces the movements of the mouse during a drag operation in a path object and displays it on the screen. To keep things simple, we use Scala closures as observers.

```
var path: Path = null
val moveObserver = { (event: MouseEvent) =>
  path.lineTo(event.position)
  draw(path)
}
control.addMouseDownObserver { event =>
  path = new Path(event.position)
  control.addMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
  control.removeMouseMoveObserver(moveObserver)
  path.close()
  draw(path)
}
```

The above example, and as we will argue the observer pattern as defined in [25] in general, violates an impressive line-up of important software engineering principles:

Side-effects Observers promote side-effects. Since observers are stateless, we often need several of them to simulate a state machine as in the drag example. We have to save the state where it is accessible to all involved observers such as in the variable `path` above.

[Maier et al., 2010]

User Interface Architecture

What can we learn?

- Computational systems are filled with complexity
- We need structure and organization to manage the complexity
- Individual widgets and the graphical interfaces that contain them require patterns and architectures
- Design patterns and architectures can help us communicate and envision how to bring disparate elements together

Next Class

Tuesday

- Lecture and Team Work Time
- Keep working on A2 (DUE 1/24)
- Keep working on T2 (DUE 1/25)

References

References i

Alexander, C. (1977).
A pattern language: towns, buildings, construction.
Oxford university press.

Maier, I., Rompf, T., and Odersky, M. (2010).
Deprecating the observer pattern.
Technical report.

Wikipedia (2021a).
Graphical user interface.

Wikipedia (2021b).
Observer pattern.