

# Informatics 134

Software User Interfaces  
Spring 2024

---

Mark S. Baldwin

*baldwinm@ics.uci.edu*

4/04/2024

# Agenda

1. User Interface Architecture
2. Assignment 1 & 2: Roll Your Own Widget(s)
3. Next Class
4. References

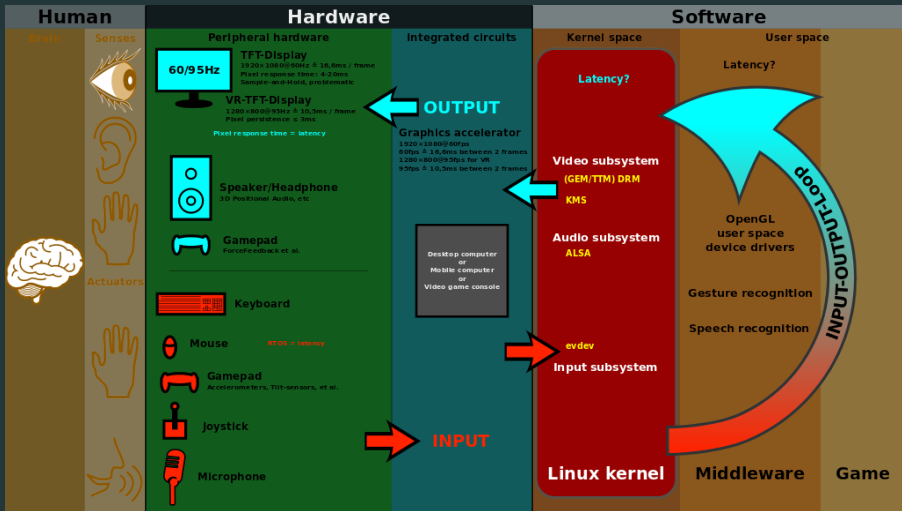
# User Interface Architecture

---

# User Interface Architecture

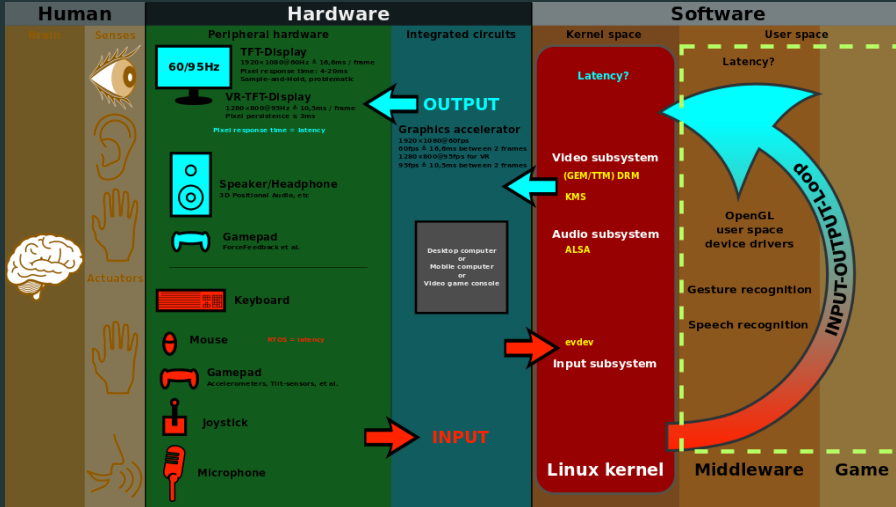


# User Interface Architecture



[Wikipedia, 2021a]

# User Interface Architecture



[Wikipedia, 2021a]

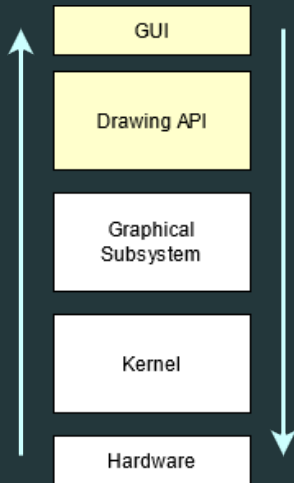
# User Interface Architecture

## User Interfaces from an Architectural Level

GUIs rely on many different units of code to function

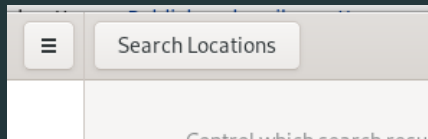
Data propagates between these units to represent state and interaction

Each unit is responsible for making decisions on how to handle a particular operation



## The Button Example

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





## The Button Example

Clickable

Can visually change in response to interaction

Can display data

Can execute a command

## The Button Example

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

## 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?

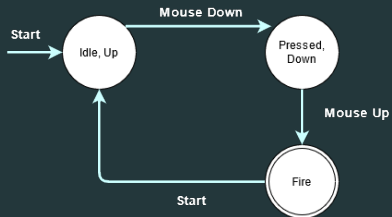
Current State	Transition	Present State
cs-1	t-1	ps-1
cs-2	t-2	ps-2
cs-3	t-3	ps-3

## Button State Chart

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

## Button State Chart

The simple button example represented using a state chart diagram



## 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?

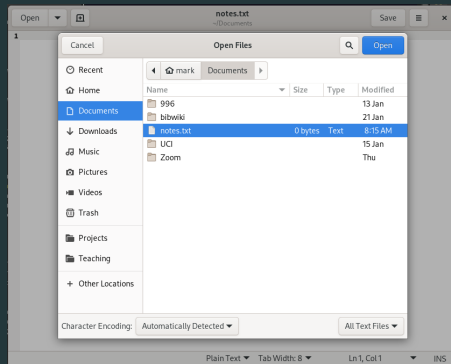
**DEMO**



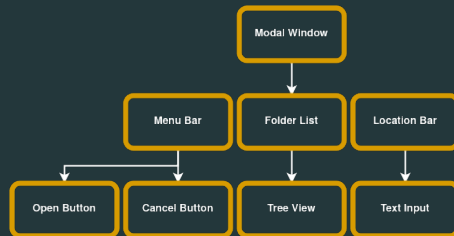
# User Interface Architecture

Let's consider a slightly more complex example

What are some observations we can make about the various widgets in this user interface?



## Hierarchy



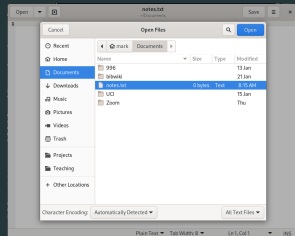
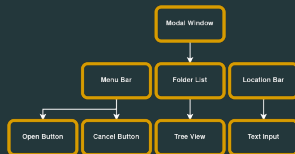
# User Interface Architecture

## GUIs are structured hierarchically

Some widgets can contain other widgets

Container widgets are not always visible

Hierarchical composition supports layout and communication between widgets

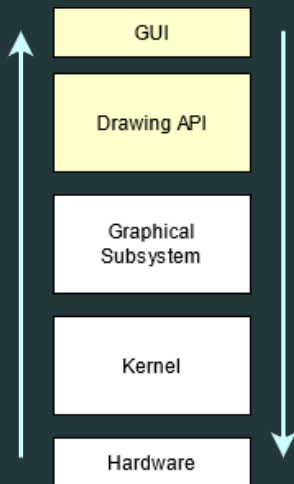


# User Interface Architecture

## Hierarchical Composition

Layout managers

Event handling and propagation

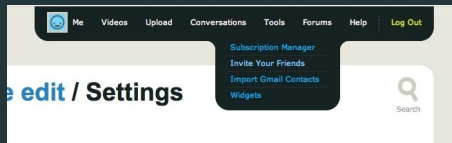


# User Interface Architecture

## UI's are hard to implement...

From a design perspective (more on that later!)

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)';
});
```

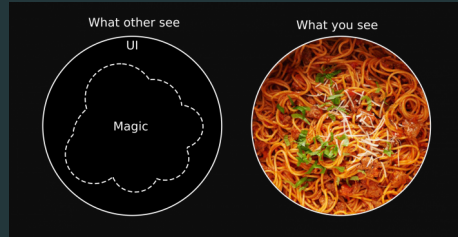
# User Interface Architecture

## 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)



I don't think so...

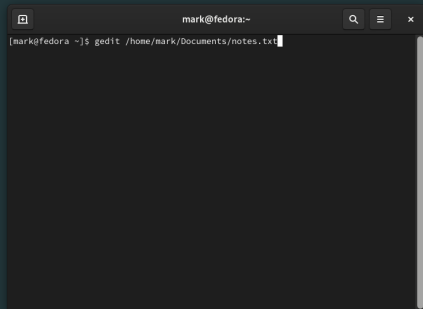
What is going on here?



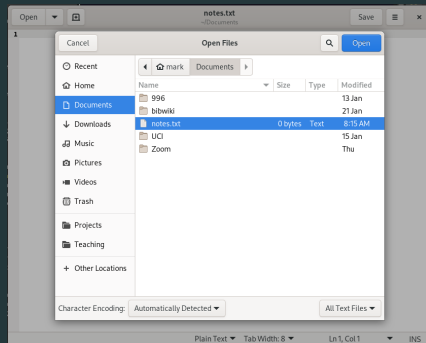
# User Interface Architecture

From a programming perspective

Consider the difference between:



and:



# User Interface Architecture

## From a programming perspective

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

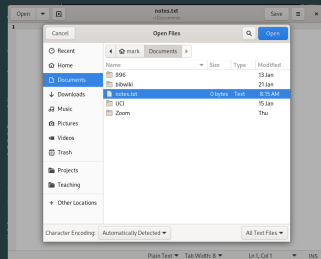
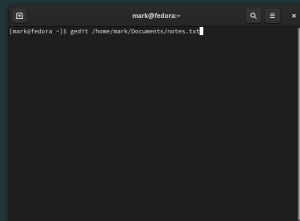
- Support modal

- Cancel (abort/escape)

- Gather and display system resources

- Search

- and many more...



## 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.

## 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.”

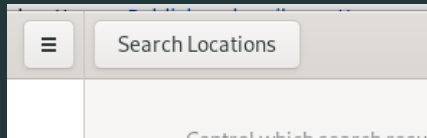
——[Alexander, 1977]



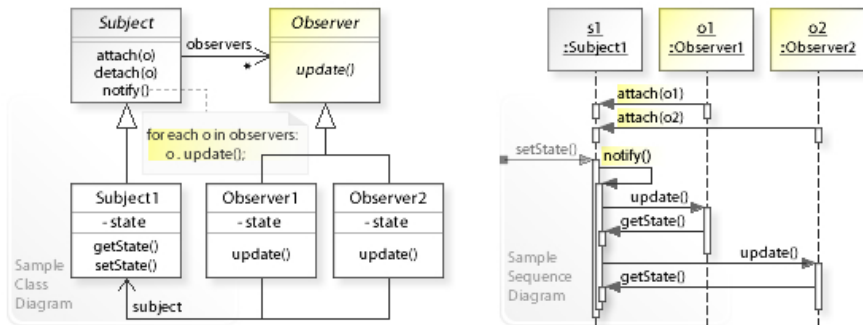
# User Interface Architecture

## From a programming perspective

UI's manage complexity through design patterns



## The Observer Pattern



[Wikipedia, 2021b]

## From a programming perspective

### The Observer Pattern

A standard model for handling event propagation across nearly all UI toolkits

### Some examples:

Microsoft .NET

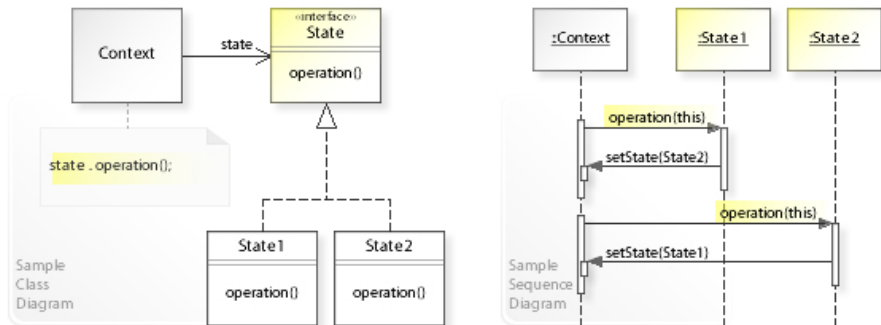
TypeScript

Angular

React

Java

## The State Pattern



[Wikipedia, 2021c]



## From a programming perspective

### The State Pattern

A standard model for managing object behavior when it's internal state changes

### Some examples:

Microsoft .NET

TypeScript

Angular

React

Java

## From a programming perspective

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

- Idle state
- Hover state
- Mouse up/down
- Pressed/released
- Hover/idle down?

We can rely on design patterns to manage the complexity. State and Observer are frequently paired together to abstract much of this complexity into patterns that are easier to reason about...but...

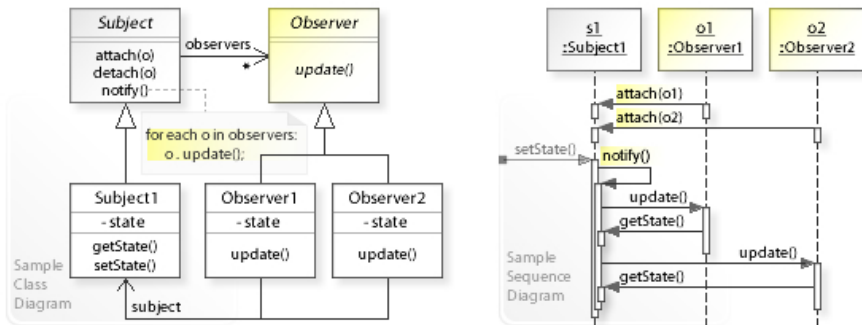
## Design patterns are not perfect

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]

## 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



[Maier et al., 2010]

# Today we see a combination of both in many modern web frameworks

## Data-flow in Angular

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-hello',
5   template: `
6     <h1>{{ message }}</h1>
7     <button (click)="updateMessage()">
8       Update Message
9     </button>
10 `
11 })
12 export class HelloComponent {
13   message: string = 'Hello World!';
14
15   updateMessage() {
16     this.message = 'New message!';
17   }
18 }
19
```

## Observer in Angular

```
1 import { Component } from '@angular/core';
2 import { MyService } from './my.service';
3
4 @Component({
5   selector: 'my-component',
6   template: `<div>{{ message }}</div>`
7 })
8 export class MyComponent {
9   message: string;
10
11   constructor(private myService: MyService) {
12     this.myService.observableMessage$
13       .subscribe((message) => {
14         this.message = message;
15       });
16   }
17 }
18
```

## Declarative vs. Imperative

### Data-flow

Less error prone without event handling/callbacks

Declarative, so easier to reason about and learn

Known to be more scalable and responsive

### Observer

Loose coupling between objects

Support for modularity

## 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



# Assignment 1 & 2: Roll Your Own Widget(s)

---

## We will be using SVG

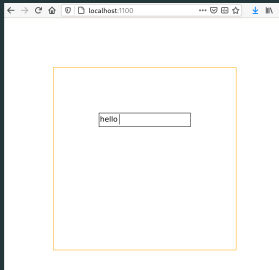
SVG.js (<https://svgjs.dev/>)

No dependencies—fast

Choose your own editor or IDE

## Hierarchy in SVG

Let's start with a look at hierarchy in raw SVG using SVG.js



```
1 import {SVG} from './svg.min.js';
2
3 SVG.on(document, 'DOMContentLoaded', function(){
4     var draw = SVG().addTo('body').size('1000px', '1000px');
5     var window = draw.group();
6     window.rect(400,400).stroke("orange").fill("white");
7
8     var group = draw.group();
9     var rect = group.rect(200, 30).fill("white").stroke("black");
10    var text = group.text("hello").move(2,4);
11    var caret = group.line(45, 2.5, 45, 25).
12        stroke({ width: 1, color: "black" });
13
14    group.move(100,100);
15
16    window.add(group);
17    window.move(100,100);
18 });
```

## SVG and the DOM

Which language?

Javascript or Typescript?

Typescript is built on strong types and object oriented principles

Javascript does not require transpiling

## SVG and the DOM

### Object Oriented Model

- Already popular with GUI toolkits

- Conceptually similar to primitives and aggregates

- Easier (IMO) to reason about how a hierarchy should be structured.

## Object Oriented Hierarchy in Typescript

Abstraction

Class == graphical object

Instances

Inheritance

---

```
1 interface WidgetState{
2     ...
3 }
4
5 class Window implements WidgetState{
6     ...
7 }
```

---

## Object Oriented Hierarchy in Typescript

Abstraction

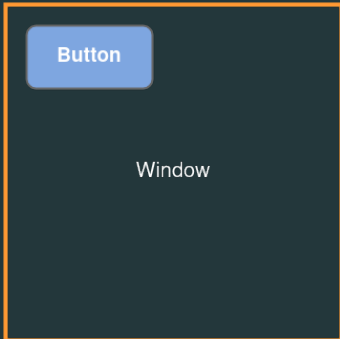
Class == graphical object

Instances

Inheritance

```
1 abstract class Component implements IAccessibility {  
2  
3 }  
4  
5 class Widget extends Component {  
6     constructor() {  
7         super(100, 50);  
8         ...  
9     }  
10 }
```





---

```
1 let w = new Window(500,500);  
2 let btn = new Button(w);  
3 btn.backColor = "blue";
```

---

## Object Oriented Model

Already, you can see how decisions need to be made.

- Design and optimize classes and interfaces to avoid duplicate code (costly space/performance)

- Encapsulate when possible (the Button class should not need to worry about bgcolor change)

- Sensible names and parameters

# A1-A2: Custom Graphical Toolkit

**You will create the following widgets (and more!):**

- Button (use starter code, customize)
- Check Box
- Radio Button
- Scroll Bar
- Progress Bar
- Custom (your choice)

## A1-A2: Custom Graphical Toolkit

**You will be responsible for all of the following:**

- Apply a custom theme across all of your widgets
- Implement features
- Create a state chart for each widget
- Create a small GUI program that makes use of all of your widgets

# A1-A2: Custom Graphical Toolkit

## Getting started:

- Full assignment is upon the course website
- Start looking at the SVG.js documentation
- Start working on your button
- We will be covering more over the next few weeks

**Let's Dive In!**

**Next Class**

---






- Structured Graphics
- Deeper Overview of SVG and the custom toolkit



## 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.**
-  Wikipedia (2021c).  
**State pattern.**