Informatics 134

Project in User Interaction Software

Software User Interfaces

Here it is again.

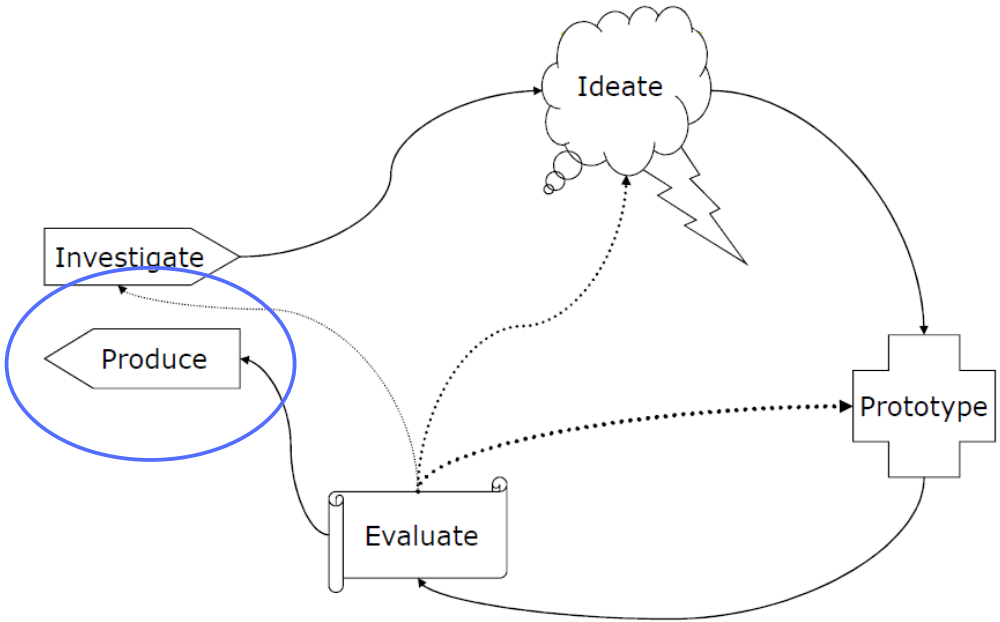
So you're heading into production...

Let's talk about your experiences so far.

- Good?

- Bad?

- Pro/con?



Software User Interfaces

While a proven approach, the iterative prototyping process is time consuming and can be costly…

- Financial cost

- Workload cost

- Physical cost

- Cognitive cost

Software User Interfaces

We apply iterative prototyping and human-centered design principles to user interface design because it works.

And...

User interfaces are hard to implement...

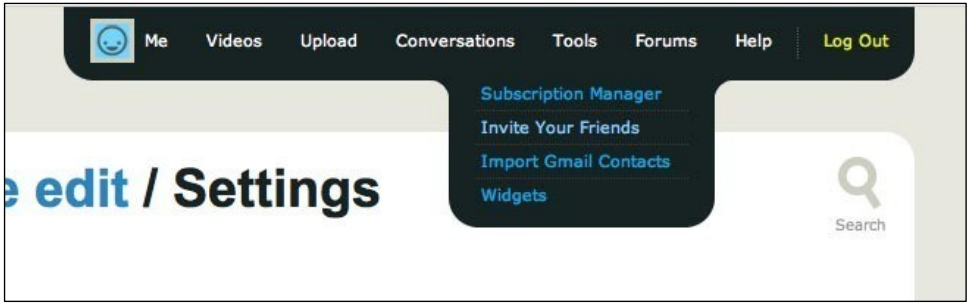
Software User Interfaces

User interfaces are hard to implement…

From a Design Perspective

And

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)';
});
```

Software User Interfaces

User interfaces are hard to implement…

From a Programming Perspective

Software User Interfaces

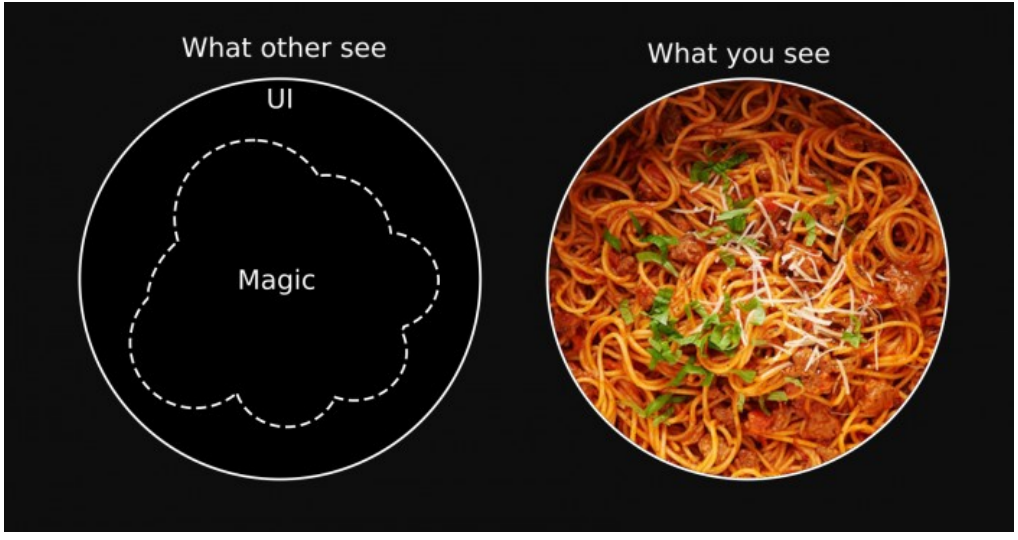
User interfaces are hard to implement…

- Reactive, must respond to hard-to-predict human behavior

- Event-based, difficult to model AND modularize

- Dependent on multi-processing

- Peripherals
- Displays
- Local and remote communication channels



Software User Interfaces

User interfaces are hard to implement…

- Must be robust enough to handle:

- Device input
- Video and audio
- Background processes

Software User Interfaces

User interfaces are hard to implement…

- Must be robust enough to handle:

- Avoid crashes
- Support recovery:
  - Helpful messages
  - Rollback/Undo
  - Escape/Abort

Software User Interfaces

User interfaces are hard to implement…

- Traditionally difficult to test

- Some automation w/ additional programming
- Gap between user experience and programmatic correctness

- Paradigms in constant flux

- Not necessarily a bad thing
- But always learning…

For example, web UI's:

- Static HTML

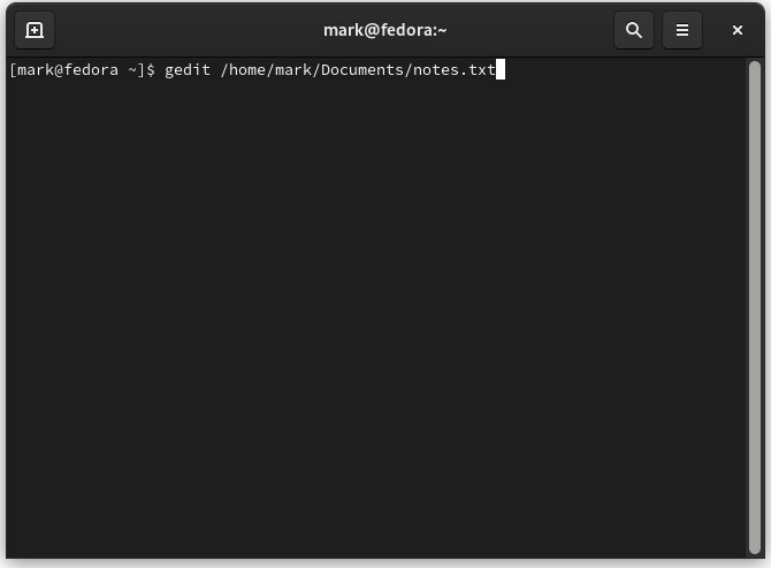
- Dynamic HTML + CSS

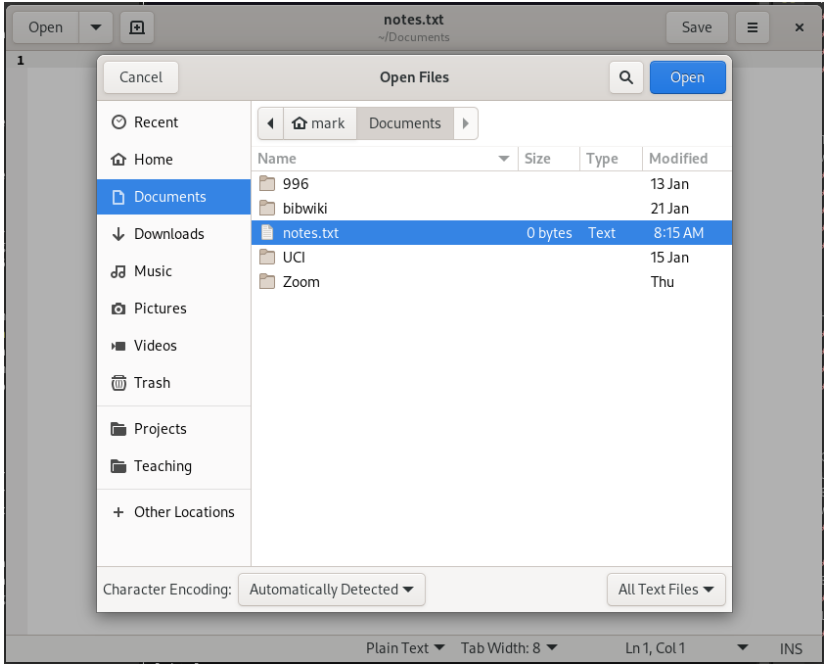
- Increasingly declarative (what not how) paradigms

- Web frameworks
- Transpiling
- CSS 3
- Templates/Components

Software User Interfaces

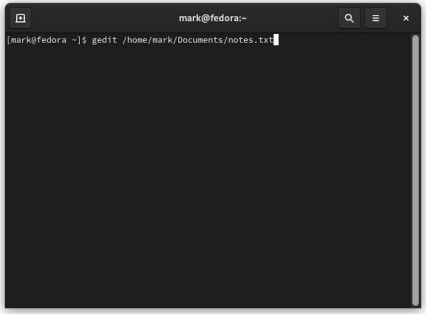
Consider the difference between:





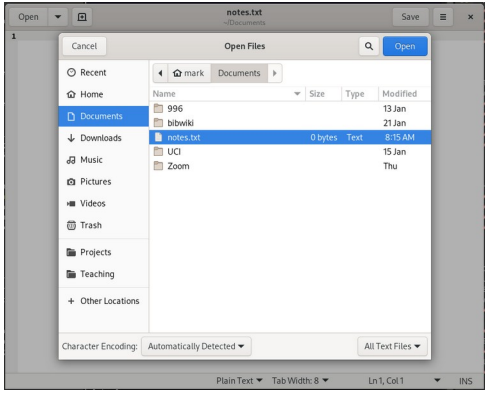
Software User Interfaces

Consider the difference between:



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

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

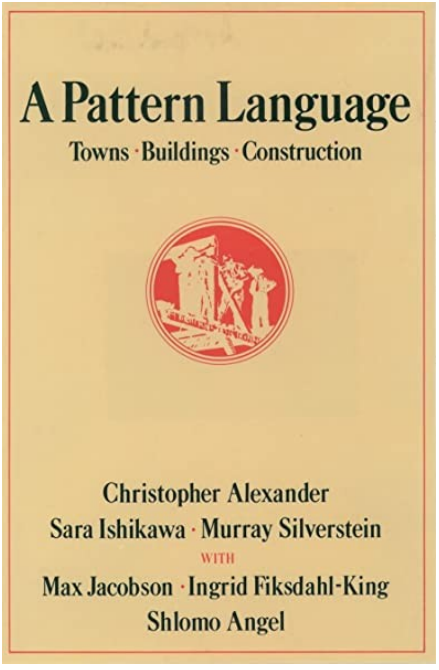


Software User Interfaces

Design patterns can help…

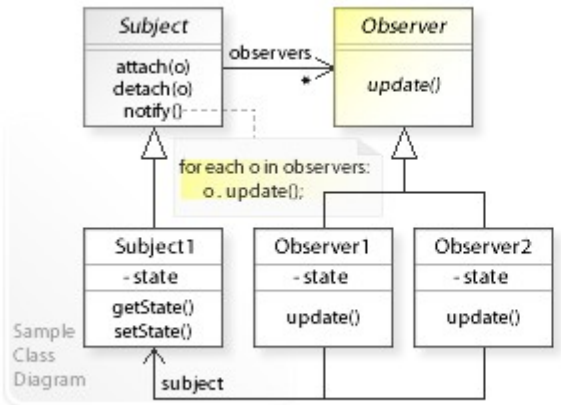
Provide a common language upon which designers and developers can reason about intent and function.

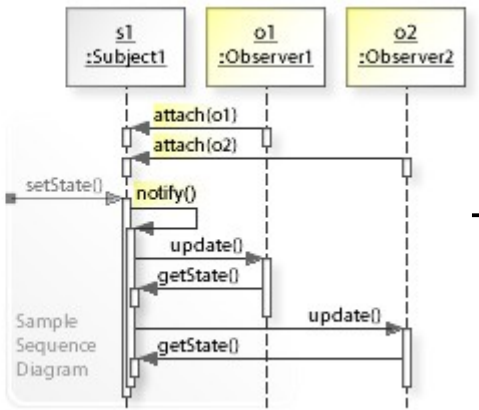
“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”



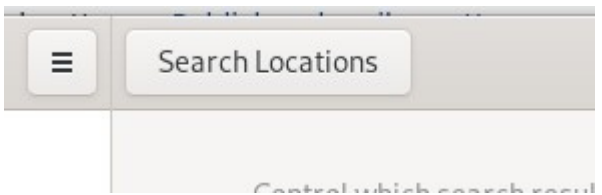
Software User Interfaces

UI's manage complexity through design patterns:





The Observer Pattern



or listener, or pub/sub

https://en.wikipedia.org/wiki/Observer_pattern

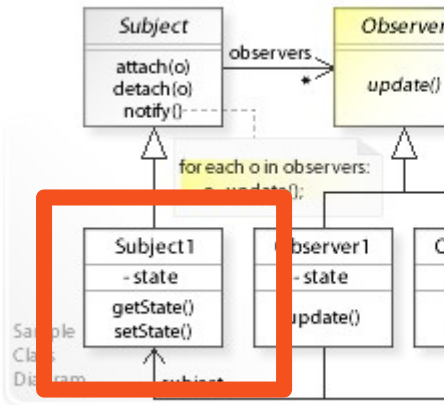
Software User Interfaces

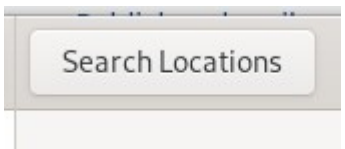
The Observer Pattern

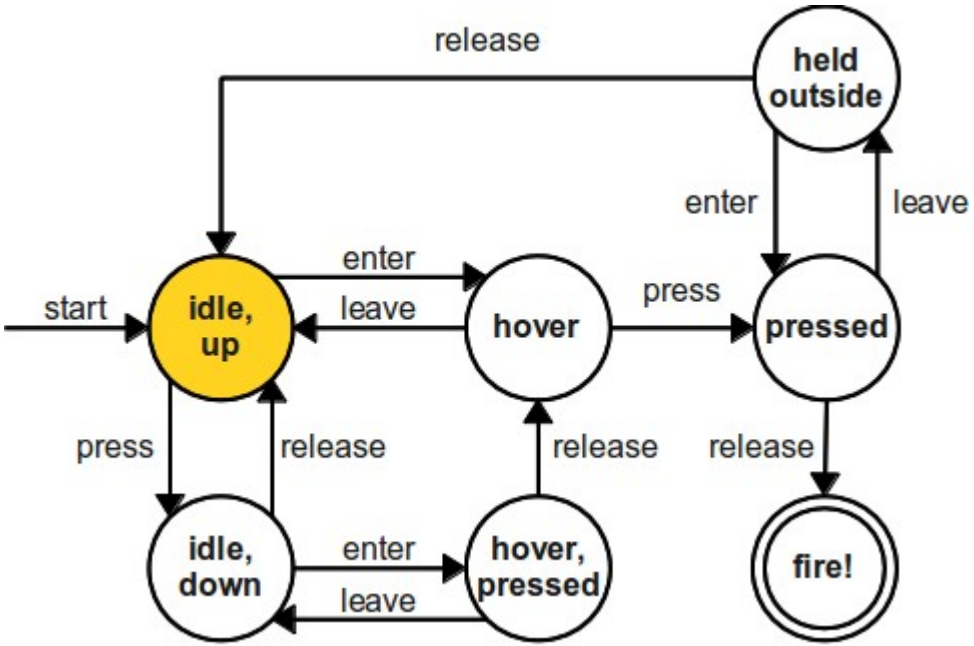
A standard model for handling event propagation across nearly all UI toolkits...

Software User Interfaces

But even a simple button is filled with complexity.







https://web.stanford.edu/class/archive/cs/cs103/cs103.1142/button-fsm/

Software User Interfaces

As UI complexity grows, design patterns lead to code that is hard to learn.

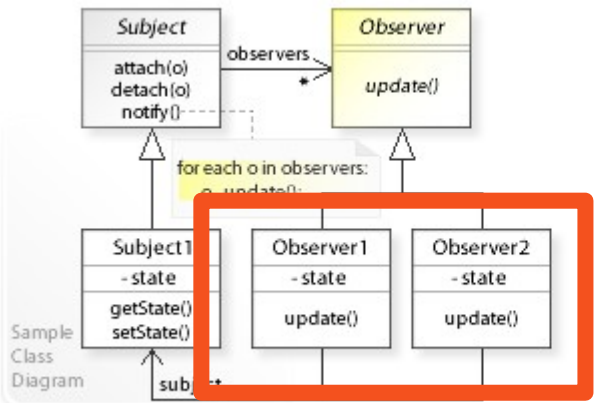
The observer pattern, for example:

- promotes side-effects

- Since a subject is decoupled from its observers, a button event (click, hover) can have *n* observers…

- Difficult to trace control flow/debug.

- And many more…



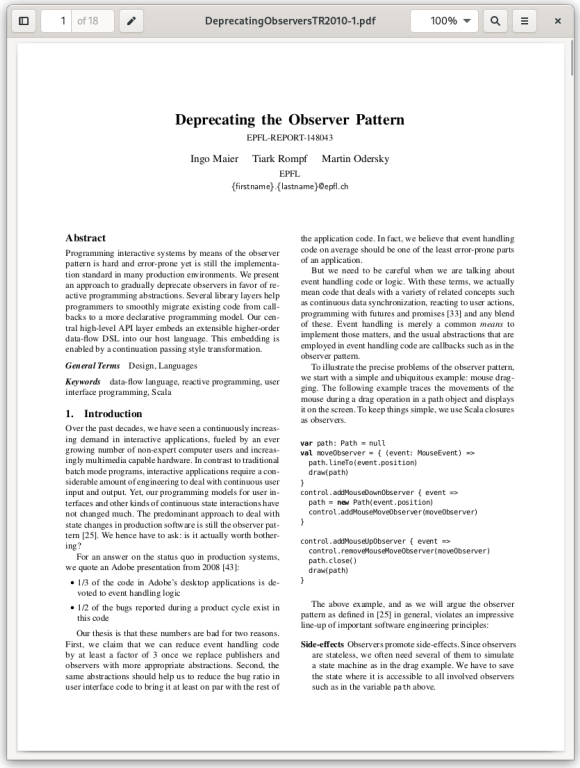
Software User Interfaces

Deprecating the Observer Pattern

- Work by Martin Odersky (Scala, Generic Java, other contributions)

- Via Scala.React system, paradigm shift from observer-based to data-flow based model.

https://infoscience.epfl.ch/record/148043/files/DeprecatingObserversTR2010.pdf



1 of 18 DeprecatingObserversTR2010-1.pdf 100%

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.

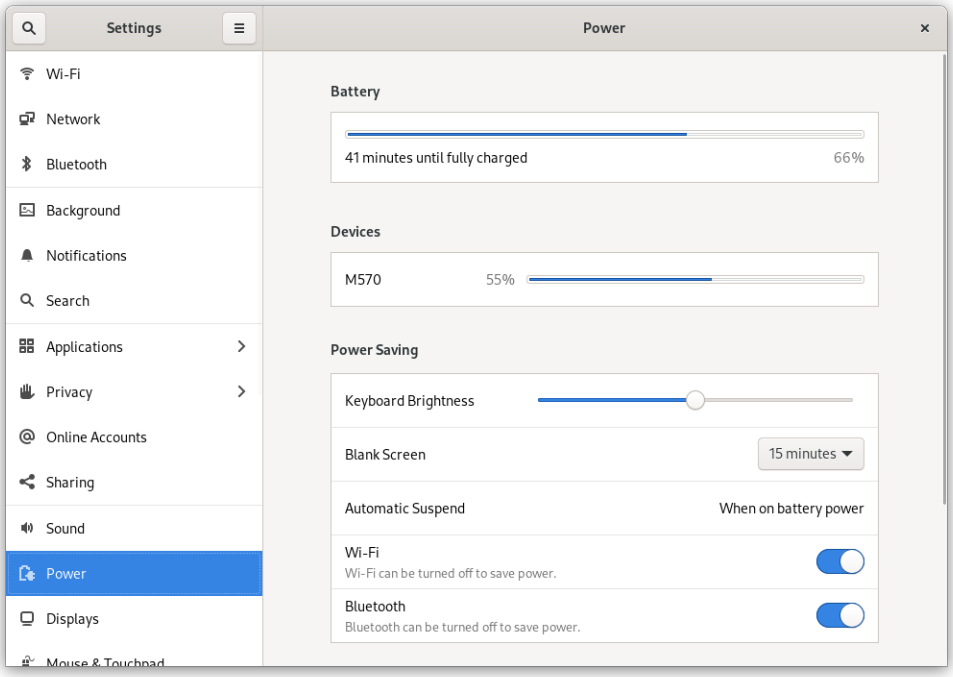
```
var path: Path = null
val moveObserver = { (event: MouseEvent) =>
  path.lineTo(event.position)
  draw(path)
}
control.addMouseDownObserver { event =>
  path = new Path(event.position)
  control.addMouseMoveObserver(moveObserver)
}

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.

Software User Interfaces

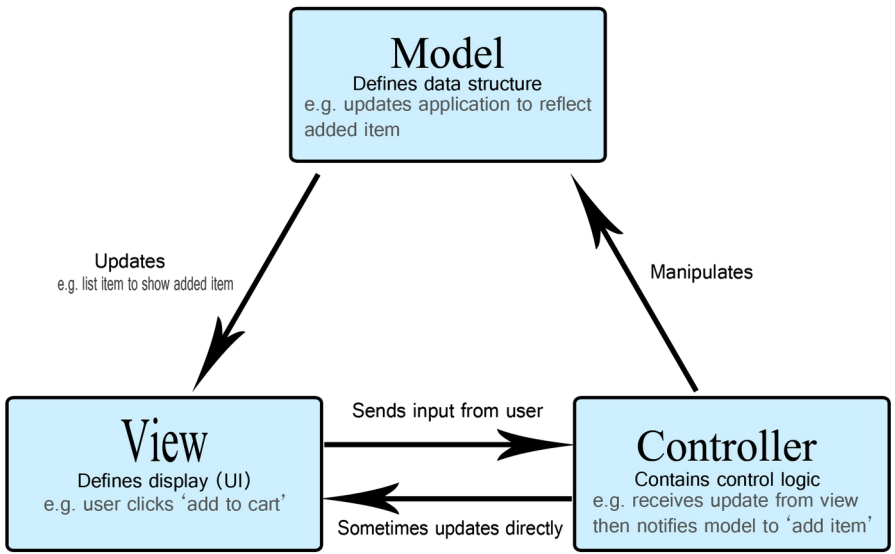
Design patterns can help us manage UI at the component level…but what happens when we have multiple components?



Software User Interfaces

Software User Interface Architecture!

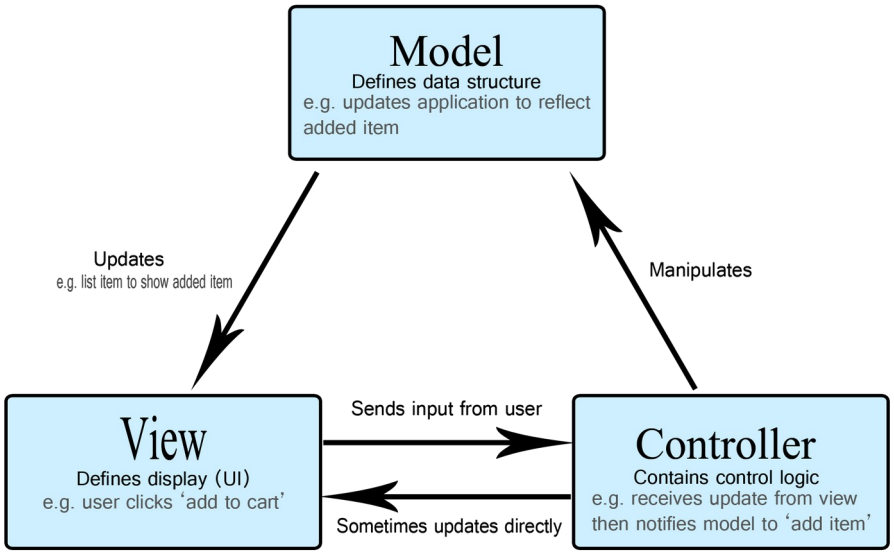
You have already built UI's using one architecture paradigm: MVC

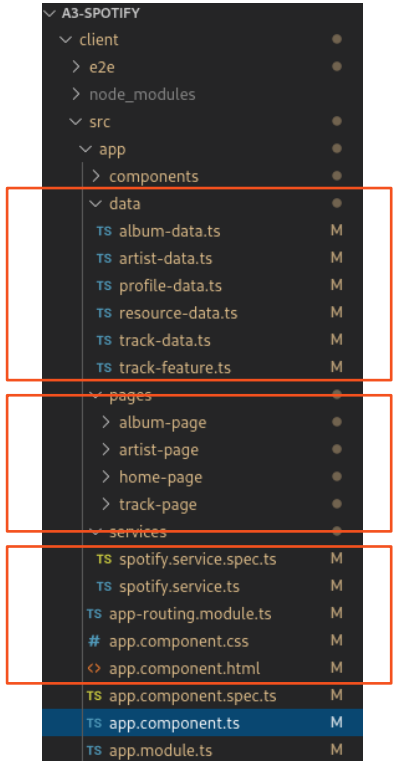


https://developer.mozilla.org/en-US/docs/Glossary/MVC

Software User Interfaces

Software User Interface Architecture!





Models

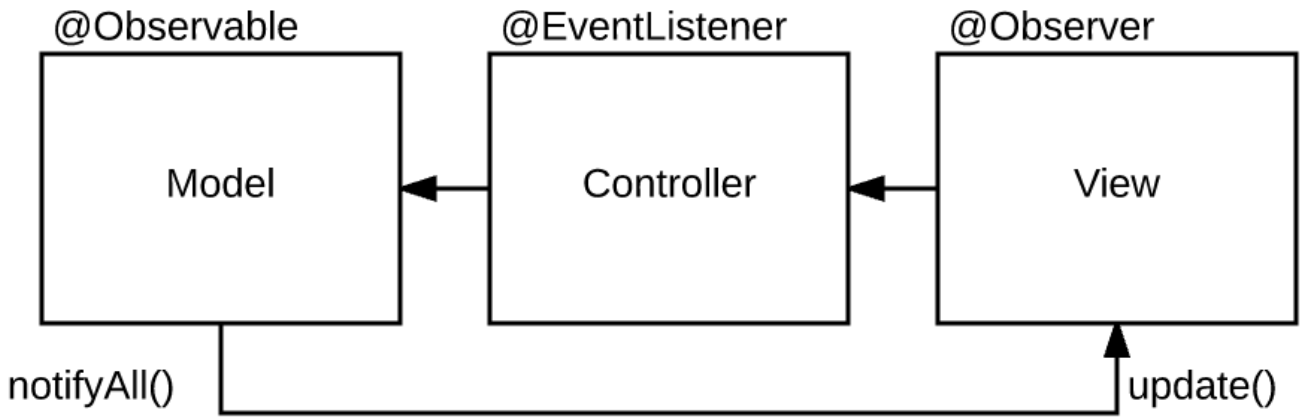
Views

Controllers

Software User Interfaces

MVC is similar to Observer

The Observer pattern forms the foundation for many MVC frameworks.



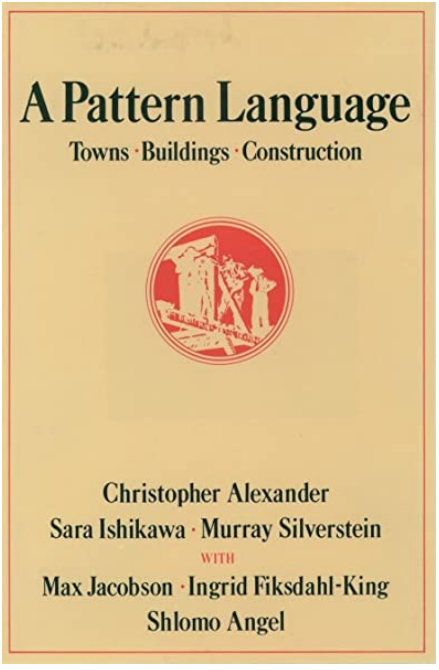
https://medium.com/@patrickackerman/the-observer-pattern-with-vanilla-javascript-8f85ea05eaa8

Software User Interfaces

User interfaces are hard to implement…

From a Design Perspective

“Provide a common language upon which designers and developers can reason about intent and function.”



Software User Interfaces

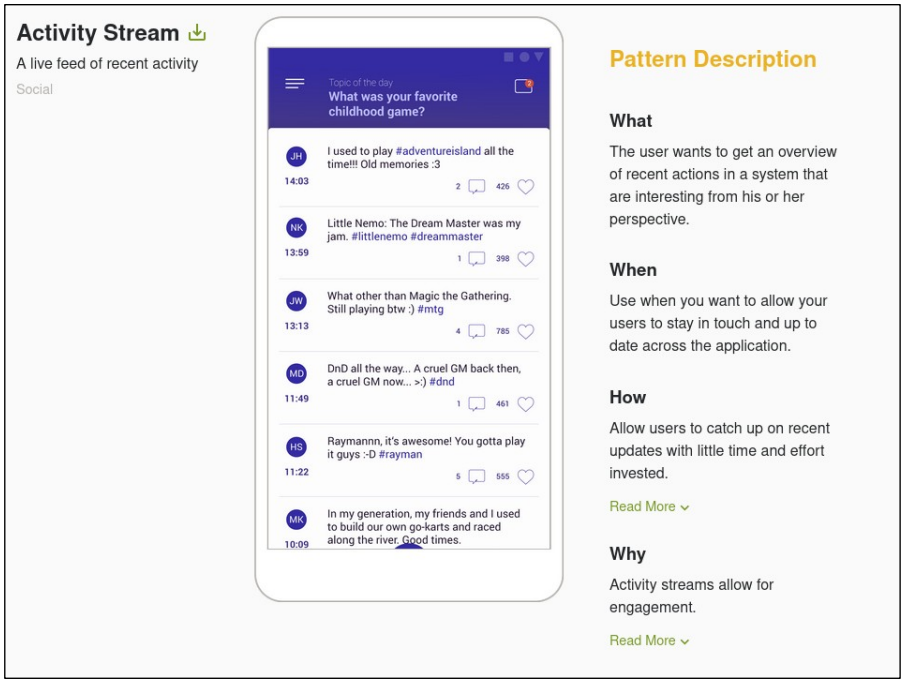
Design patterns can help…

https://uigarage.net/

http://ui-patterns.com/patterns

https://designwithkiwi.com/

https://pttrns.com/



Software User Interfaces

Design patterns can help…

Found that UI design patterns can reduce workload across multiple dimensions when compared to traditional UI design practices.

Impact of using UI Design Patterns on the Workload of Rapid Prototyping of Smartphone Applications: An Experimental Study

Sarah Suleri, Yeganeh Hajimiri, Matthias Jarke

ABSTRACT

UI design pattern-driven prototyping introduces the idea of utilizing UI design patterns to create rapid prototypes. In this paper, we investigated the impact of using UI design pattern-driven approach on the subjective workload of rapid prototyping. We conducted a between-group workload study (NASA-TLX) for the pattern-driven and traditional approach of rapid prototyping smartphone applications to compare the subjective workload, physical demand, mental demand, temporal demand, performance, effort, and frustration experienced by UI/UX designers. Within the pattern-driven approach, we explored the impact of using pattern libraries with and without pattern standard. Our results indicate that the subjective workload, physical demand, and effort experienced by UI/UX designers using the pattern-driven approach is significantly less than the traditional approach of rapid prototyping. However, there is no significant difference in subjective workload experienced while using UI design pattern libraries with and without pattern standard for documenting UI design patterns.

CCS CONCEPTS

• **Software and its engineering** → **Software prototyping**; *Design patterns*; *Patterns*; • **Human-centered computing** → **Interface design prototyping**; **Wireframes**; **Graphical user interfaces**; **User interface design**.

KEYWORDS

UI design patterns; UI design pattern-driven approach; prototyping; UI design; workload; NASA-TLX.

ACM Reference Format:

Sarah Suleri, Yeganeh Hajimiri, and Matthias Jarke. 2020. Impact of using UI Design Patterns on the Workload of Rapid Prototyping of Smartphone Applications: An Experimental Study. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '20 Extended Abstracts), October 5–8, 2020, Oldenburg, Germany.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3406324.3410718

1 INTRODUCTION & BACKGROUND

Rapid prototyping is a widely used technique to quickly create an interactive artefact that can be experienced and evaluated by experts and end-users. However, the traditional rapid prototyping process becomes tedious for UI/UX designers due to time constraints, scattered UI design knowledge, and inability of developers to reproduce the same quality of UI design [3]. To address these pain points, Suleri *et al.* proposed *UI design pattern-driven prototyping*. With this approach, they introduced the idea of utilizing UI design patterns to create rapid prototypes for agile development of smartphone applications [3].

To realize the UI design pattern-driven prototyping approach, Suleri *et al.* also introduced Kiwi[1] [3], a library of UI design patterns and guidelines that aims to consolidate UI design knowledge from various academic and non-academic sources and document it in a standard format. In Kiwi, each UI design pattern is described with information regarding **what** (problem), **when** (context of use), **how** (recommended solution) and **why** (rationale). In addition, each UI design pattern consists of a sample GUI and blueprint to outline its layout details. For the scope of this paper, we will refer to this library as a *library with pattern standard*.

Besides Kiwi, there are other web-based pattern libraries such as PatternTap[2], UIGarage[3], Nicelydone[4], Pttrns[5], and MobilePatterns[6] but these libraries do not document UI design patterns in any pattern standard format. These libraries organize patterns into numerous categories and provide a sample GUI and screen name for each pattern. For the scope of this paper, we will refer to these libraries as *libraries without pattern standard*.

In our research, we aim to investigate rapid prototyping of smartphone applications from the perspective of subjective workload. Here, *workload* refers to the perceived level of physical and cognitive burden experienced by the UI/UX designers during rapid prototyping of smartphone applications [1].

We conducted a between-group study of pattern-driven and the traditional approach of rapid prototyping to compare the subjective workload, physical demand, mental demand, temporal demand, performance, effort, and frustration experienced by UI/UX designers. Within the pattern-driven approach, we explored the impact of using pattern libraries with and without pattern standard.

Our results indicate that the subjective workload experienced by UI/UX designers using the pattern-driven approach is significantly less than the workload experienced using the traditional approach of rapid prototyping. Specifically, there is a significant decrease in physical demand and effort of rapid prototyping while using the pattern-driven approach. However, there is no significant difference

Suleri, S., Hajimiri, Y., & Jarke, M. (2020, October). Impact of using UI Design Patterns on the Workload of Rapid Prototyping of Smartphone Applications: An Experimental Study. In 22nd International Conference on Human-Computer Interaction with Mobile Devices and Services (pp. 1-5).

Software User Interfaces

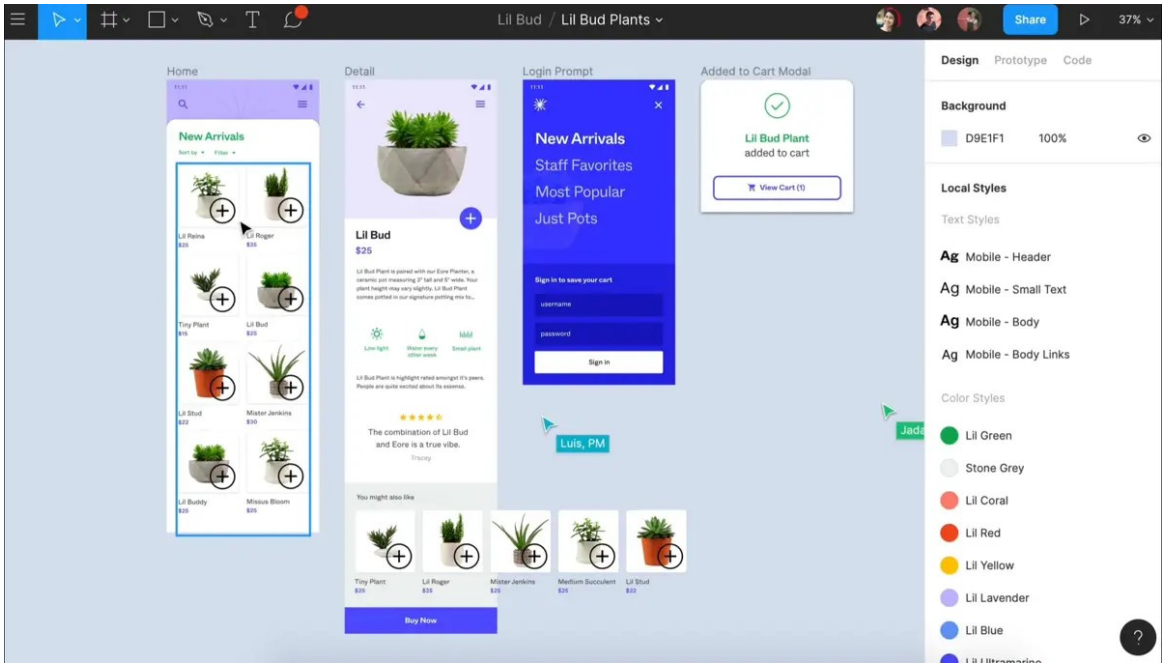
A third way emerges…

Figma?

What else?

- Qt Design Studio

- MS Visual Studio



Software User Interfaces

A third way emerges…

User interface tools:

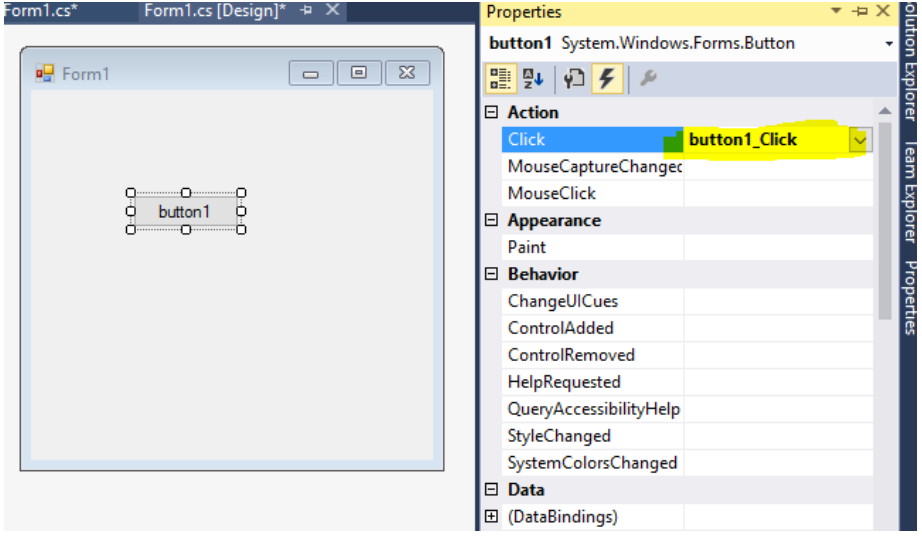
- Support rapid prototyping (pre-coding)
- Reusability (can apply to multiple platforms)
- Add consistency across platforms
- Bring designers, developers, and researchers together through a single tool.

Software User Interfaces

A third way emerges…

User interface tools (some types):

- Automate much of the coding process
- Replace programming steps with graphical configuration
- Lower level of expertise
- Raise level of reliability



Software User Interfaces

A third way emerges…

User interface tools that do both:

- Make creating UI easy and easy to use.
- Non-programmers can participate in the design and implementation process
- What else?

Software User Interfaces

A third way emerges…

User interface tools that do both:

- Make creating UI easy and easy to use.
- Non-programmers can participate in the design and implementation process
- What else?

What else?

- Automate testing?
- Validation?
- Drag and drop UI Design patterns?
- Intuitive/built-in support undo, error, accessibility?

Software User Interfaces

What can we learn from all of this?

- Input techniques and the interfaces that encapsulate them require patterns and architectures.

- Design patterns and architectures help us communicate and envision how to bring disparate elements together.

- But also set how testing and debugging will work.

- Tools can bring automation and cross-discipline expertise together.

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Wearable Technologies

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Augmented Reality

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Virtual Reality

Software User Interfaces

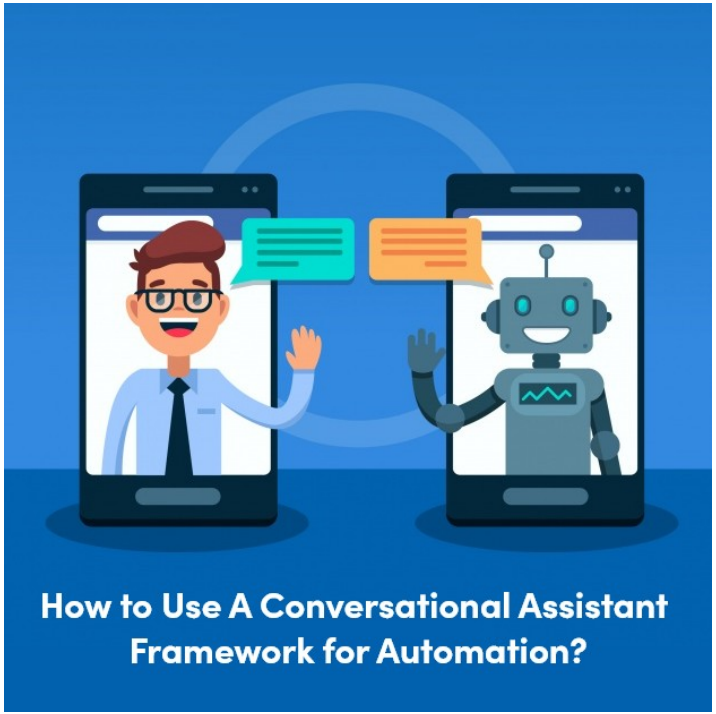
Can we apply this knowledge to other types of user interfaces?

Voice Assistants

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Conversational Agents

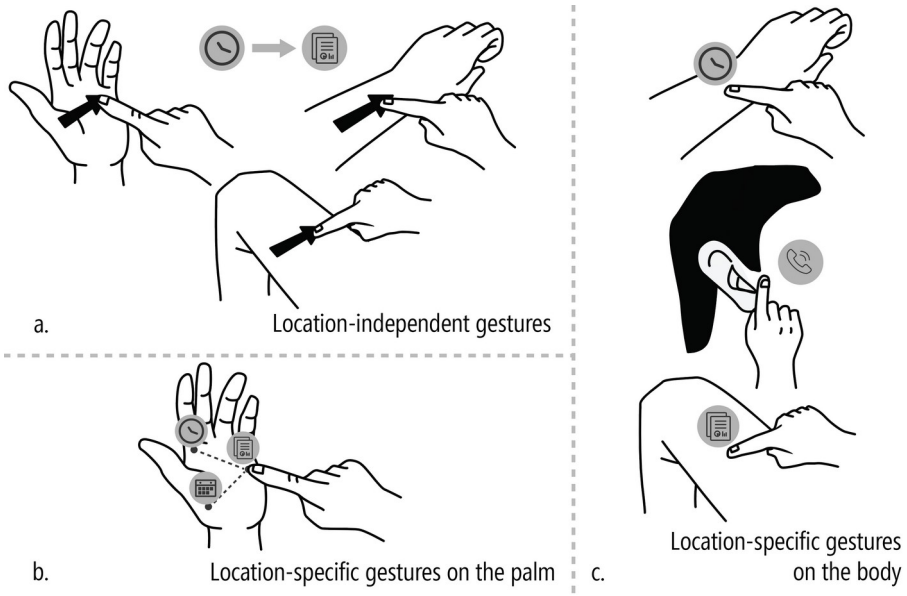


Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Future advances

- **Eyewear**
- **On-Body Interaction**
- **Projected Displays**

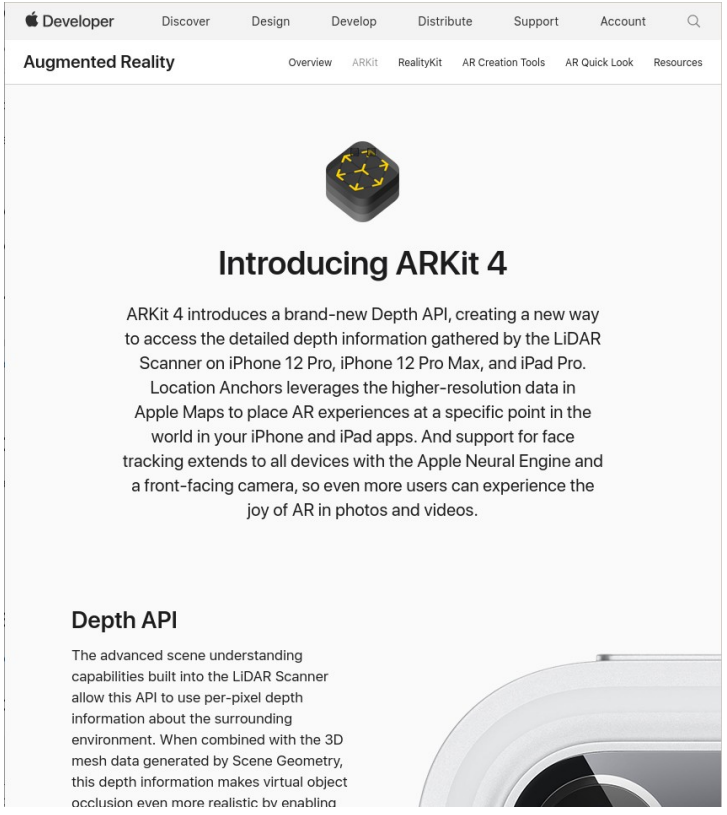


https://makeabilitylab.cs.washington.edu/project/onbodyinteraction/

Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

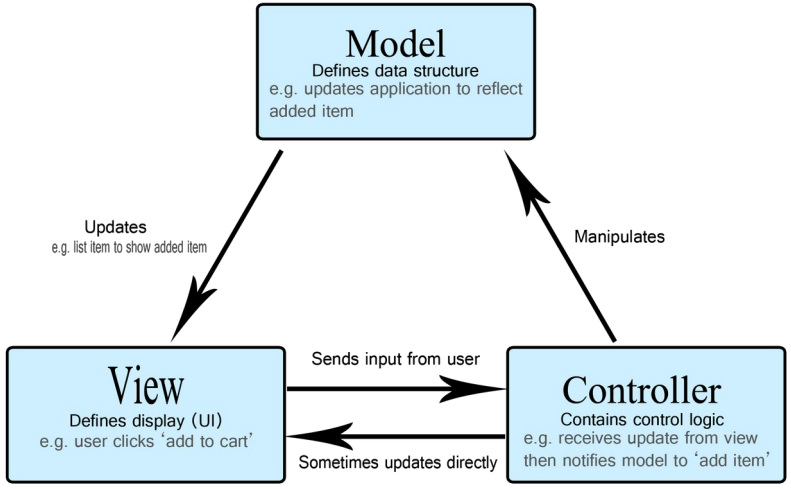
We are just getting started…



Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

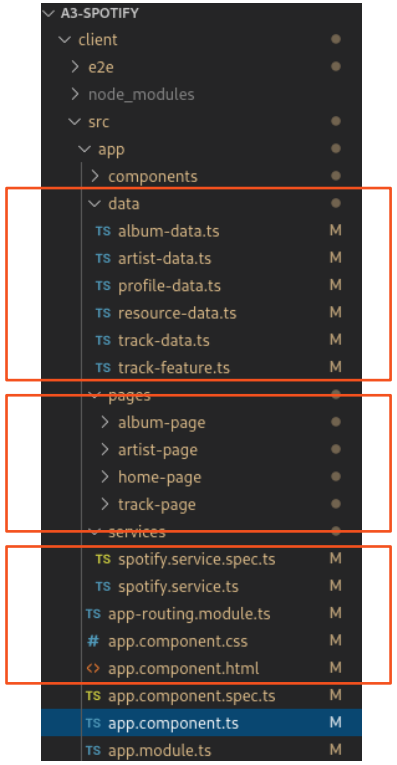
Does MVC work when your view interfaces with the auditory channel? Vibrotactile channel?



Software User Interfaces

Can we apply this knowledge to other types of user interfaces?

Can we still build interfaces through a single tool when they are spread across many different types of devices?



Models

Views

Controllers

Software User Interfaces

For Your Final Exam:

- Consider the device you have selected (Wearable, AR,VR,Voice, or Speculative)

- How does it fit into your scenario?

- What purpose does it server?

- How does it enhance the products that your team built?

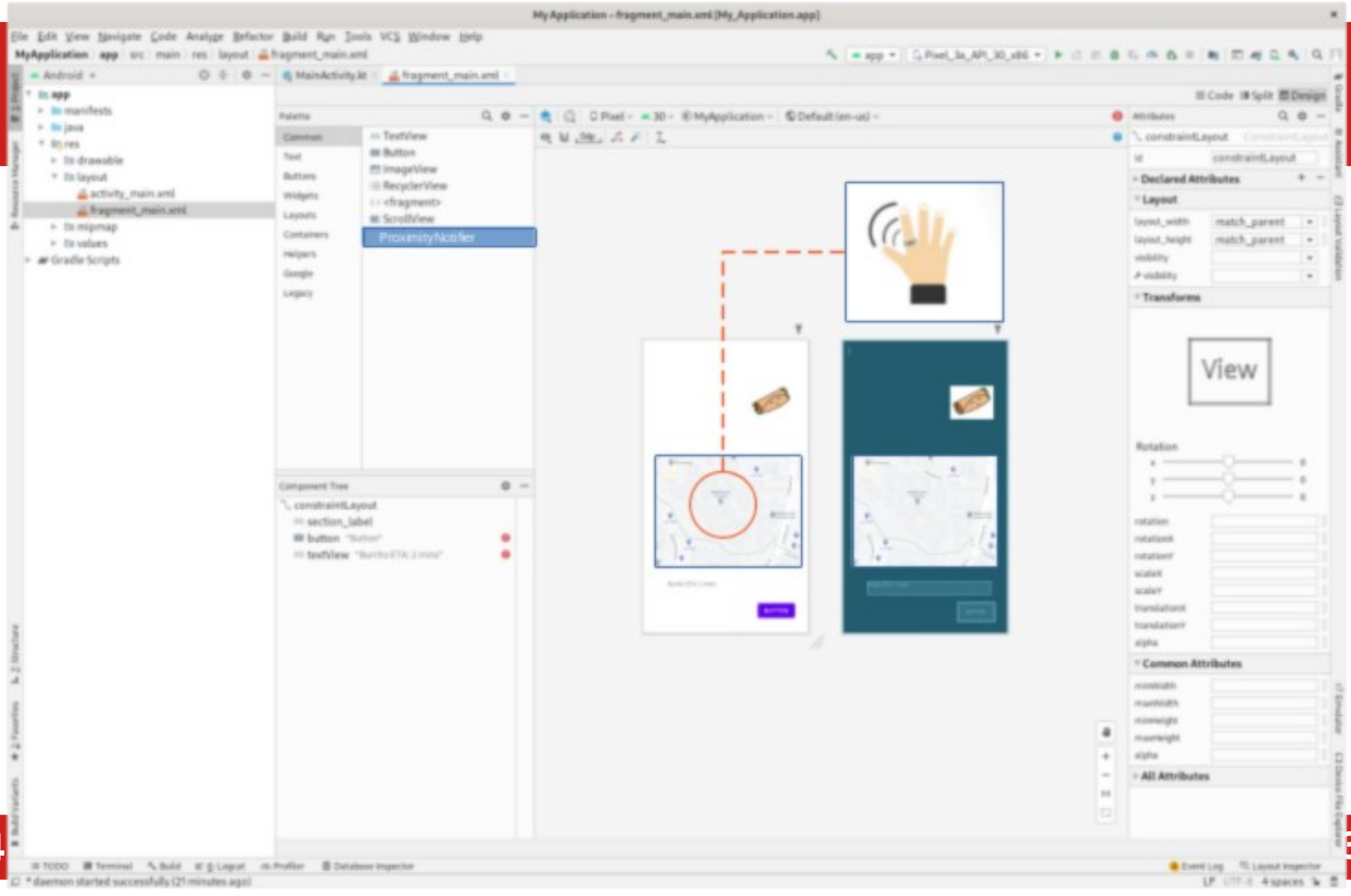
- What is the user interface and what are the user interactions?

Software User Interfaces

For Your Final Exam:

- You will submit:

- A 1-2 introductory paragraph overview of your device.
- At least 1 storyboard that explains how a person would use your device alongside your team product(s).
- Sketches, wireframes, or any other assets you feel will help communicate your idea.
- Modify the prototyping tool that your team used to build your prototypes to add support for your device. Include assets necessary to clearly support this modification.
- Assemble as either a slide deck or whitepaper and submit as PDF.


My Application – fragment_main.xml [My_Application.app]
ProximityNotifier
View

Software User Interfaces

Final Presentations

- 8 minutes per team:

- 5 minutes for presentation
- 3 minutes for discussion

- Nominate a speaker or take turns

- Prepare 2-5 process slides

- Spend about 1-2 minutes

- Demo your functional prototype

- Spend about 3-4 minutes

Minimum of 5	
Tuesday 3/9	Thursday 3/11
EcoStyle	AudioPod
ZotHealth	Mgr. Peter
Student Search	MusicHub
PeterPal	Point&Click
CLEM	Petr-Patter
	KAST
	MightByte
	ProjectCupid
	Mike + Spice Girls
	Visurely