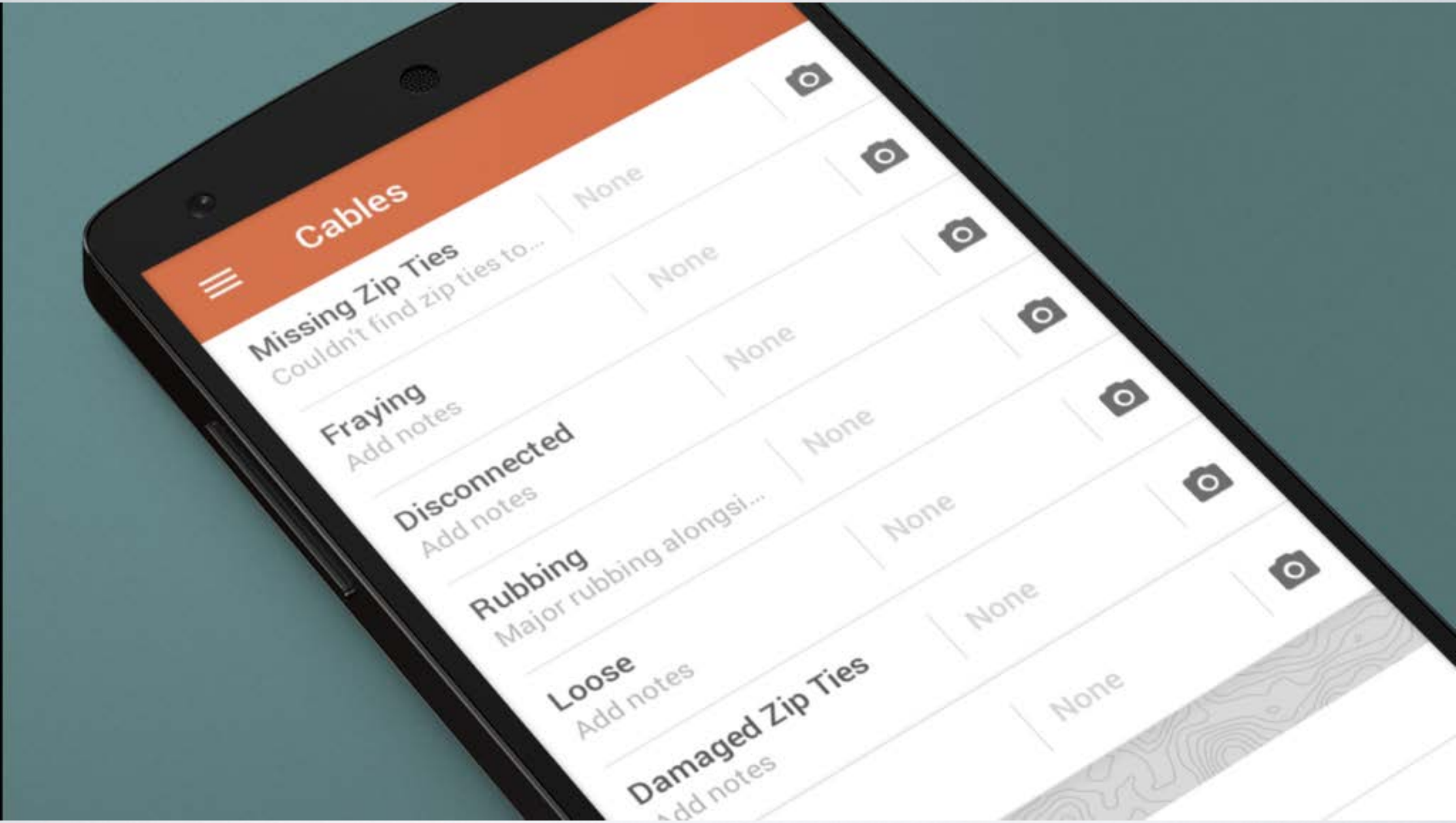
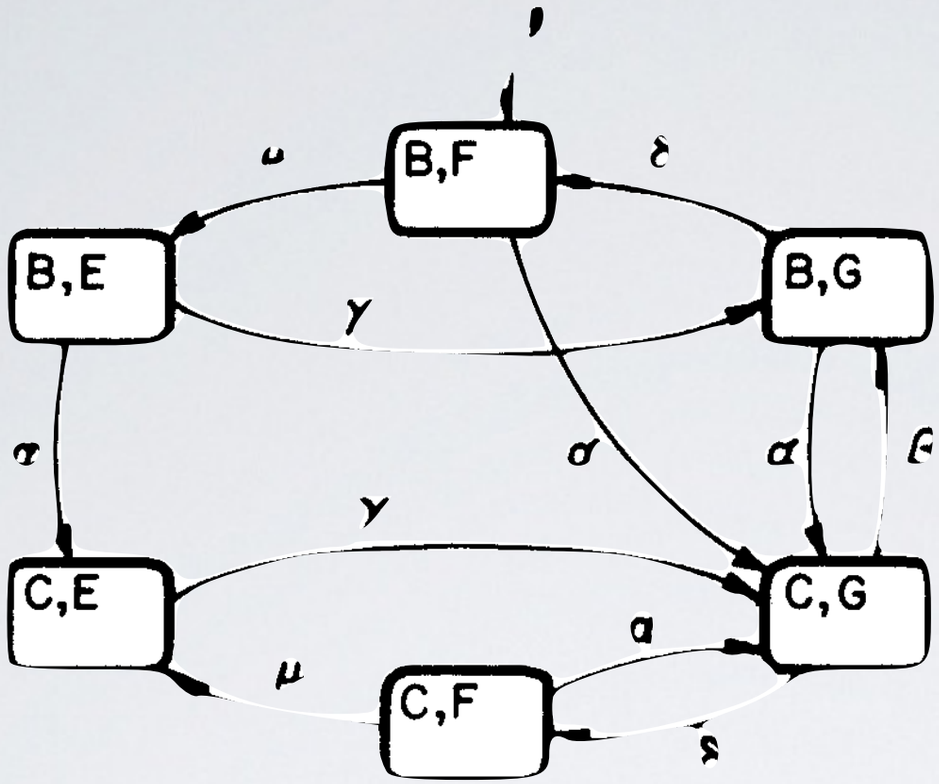
Using ClojureScript with heavy industrial equipment

Kevin Lynagh
Keming Labs
2015 May 22


110 V


Cables
Missing Zip Ties
Couldn't find zip ties to...
None
Fraying
Add notes
None
Disconnected
Add notes
None
Rubbing
Major rubbing alongsi...
None
Loose
Add notes
None
Damaged Zip Ties
Add notes
None

$$f(x) = x$$



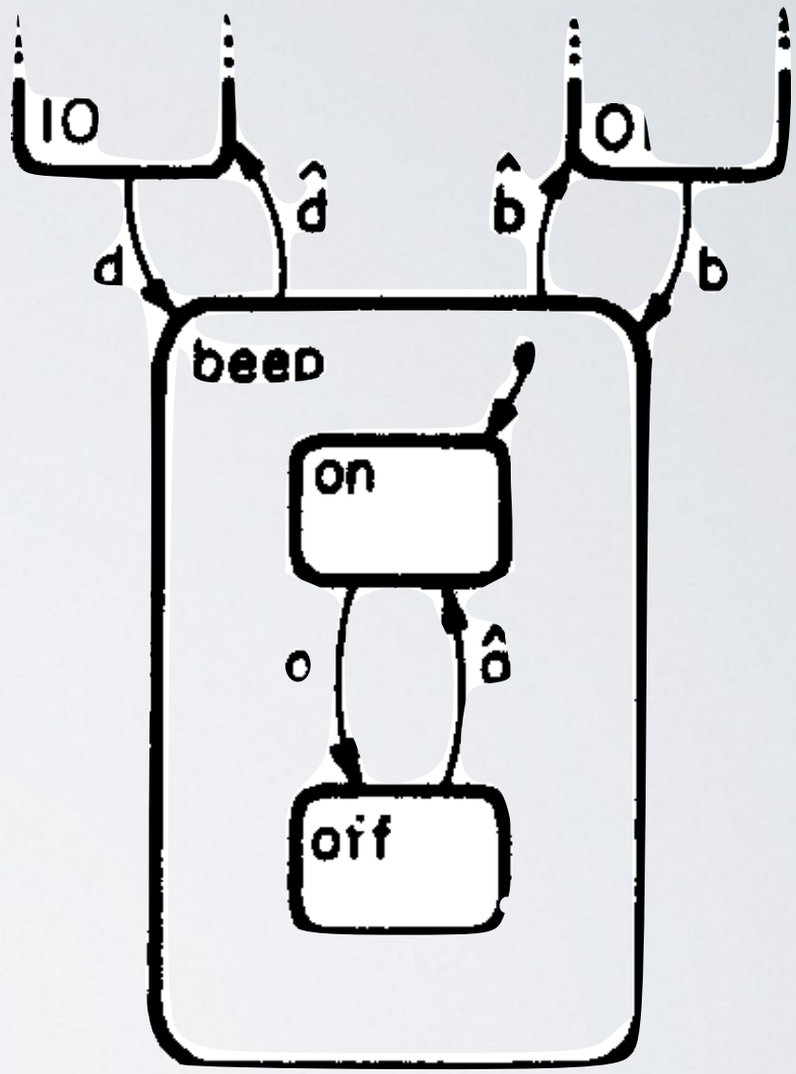


Fig. 32

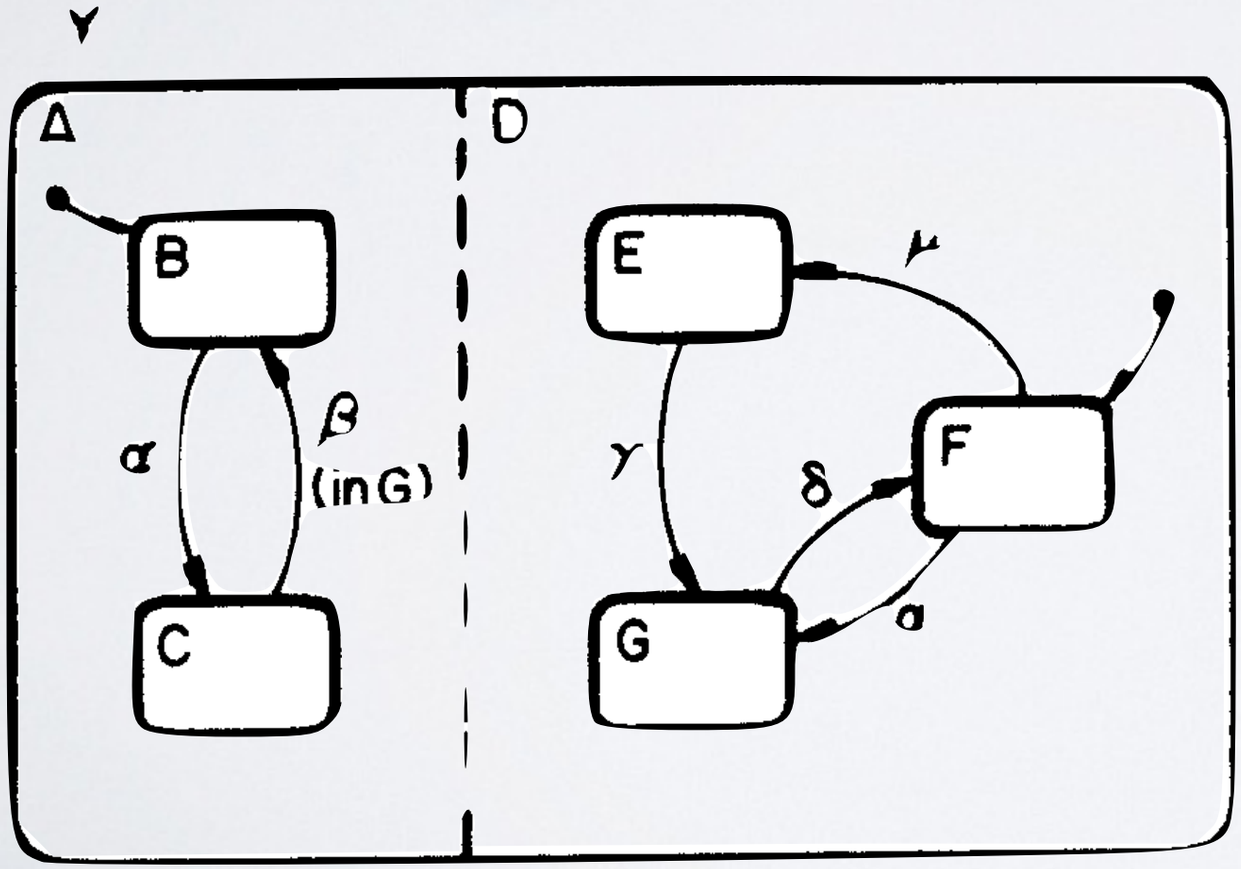


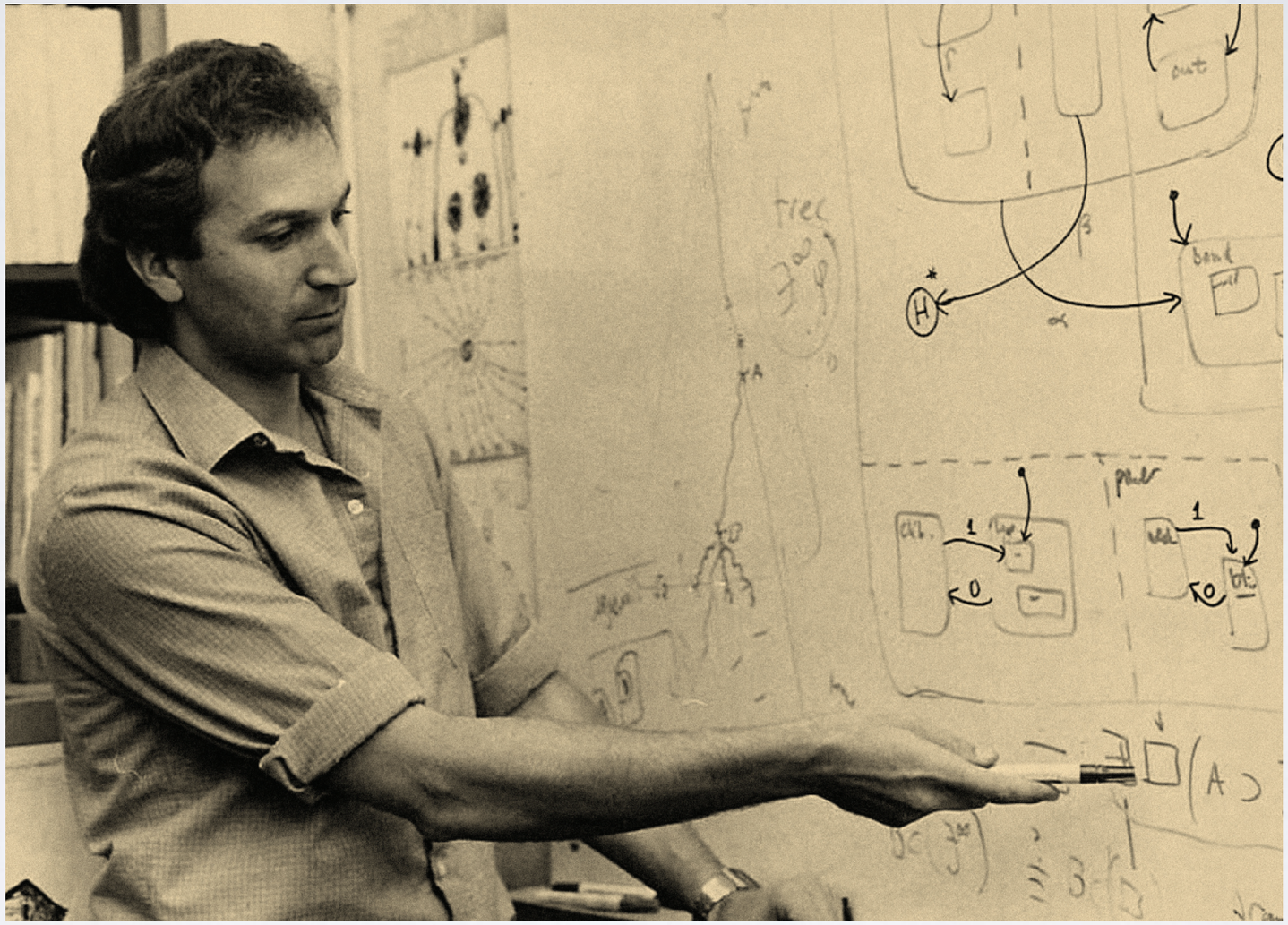
Fig. 33

Things you can take home

Things you can take home

```
(defn hello
  [name]
  (str "Hello " name "!"))

(hello "YOW") ;;=> "Hello YOW!"
```

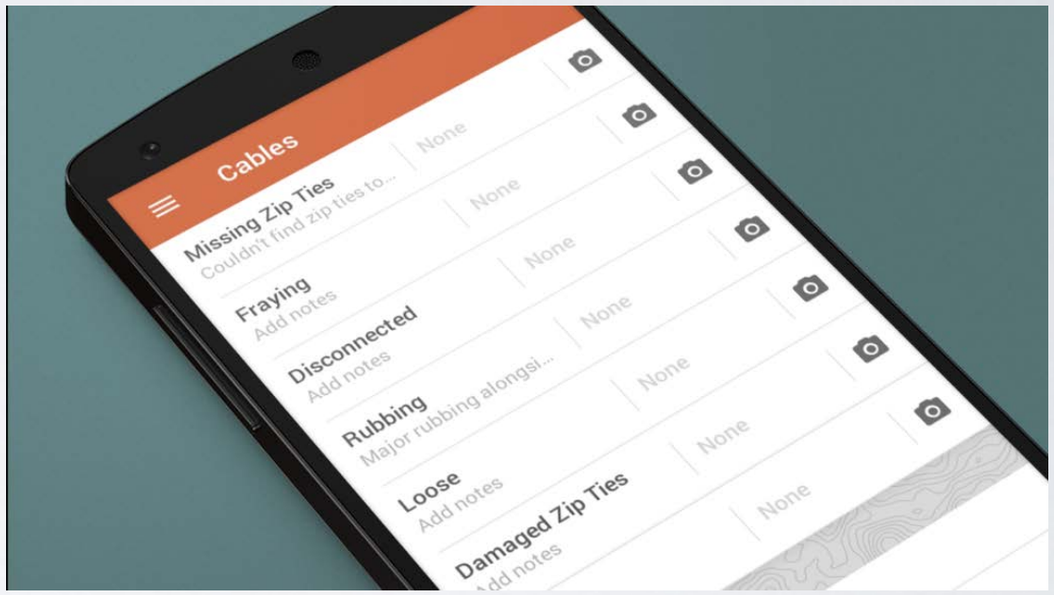
Things you can take home



This one weird trick Hacker News keeps off the frontpage

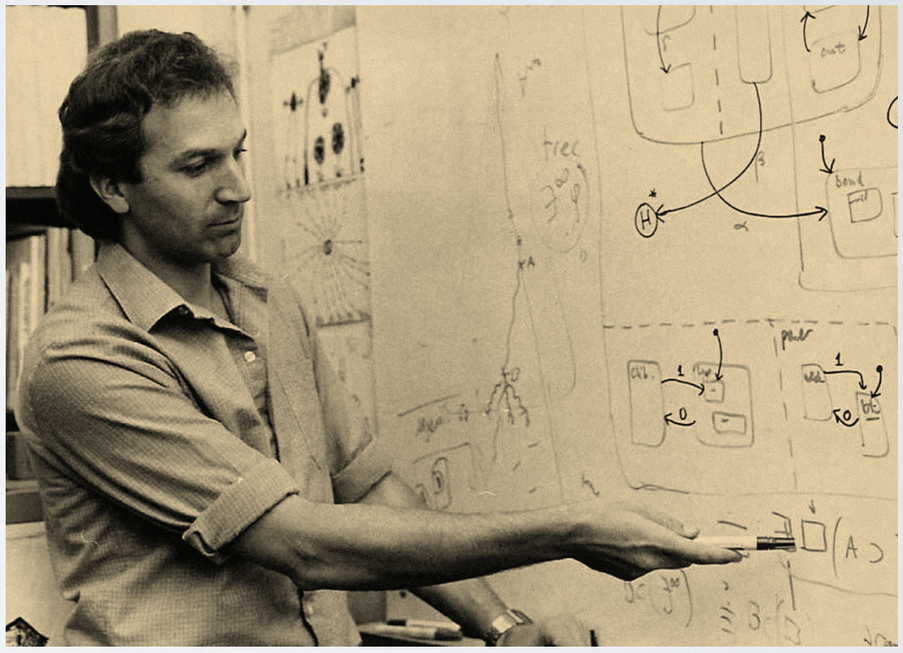
Things you can take home

$$f(x) =$$



Things you can take home

Talk outline



React

(optional)


110 V


110 V
HB29971109F
100
ONE HUNDRED DOLLARS


110 V


CHROME
VANADIUM
200MM


LIMITED
F150
Ford

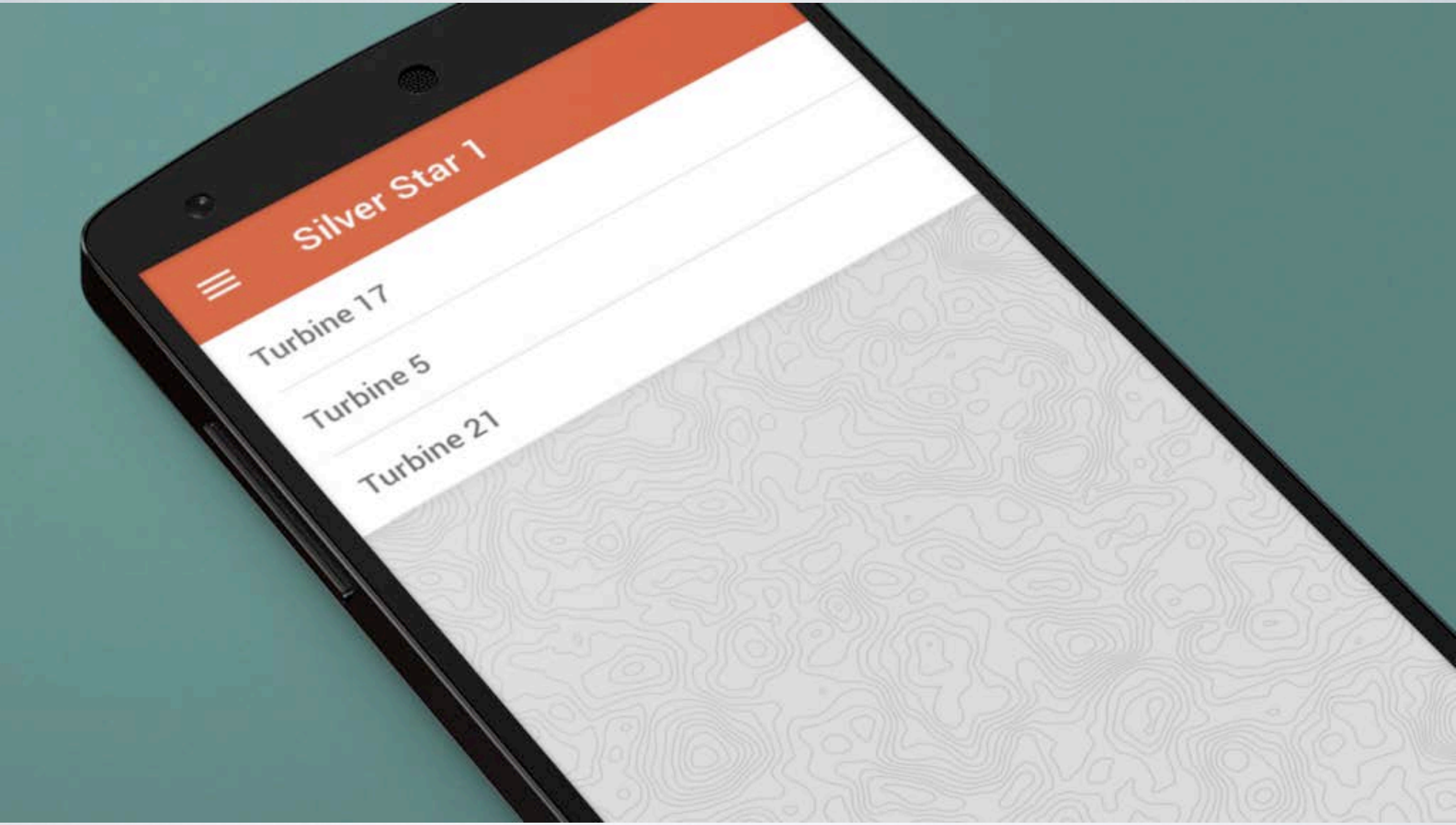

CHROME VANADIUM
200MM
110 V
HB29971109F
100

Needed: A better workflow

Paperwork is hard to sync, bulky

Enable preventative maintenance

Photos are easy + more detailed


Silver Star 1
Turbine 17
Turbine 5
Turbine 21

Complex details

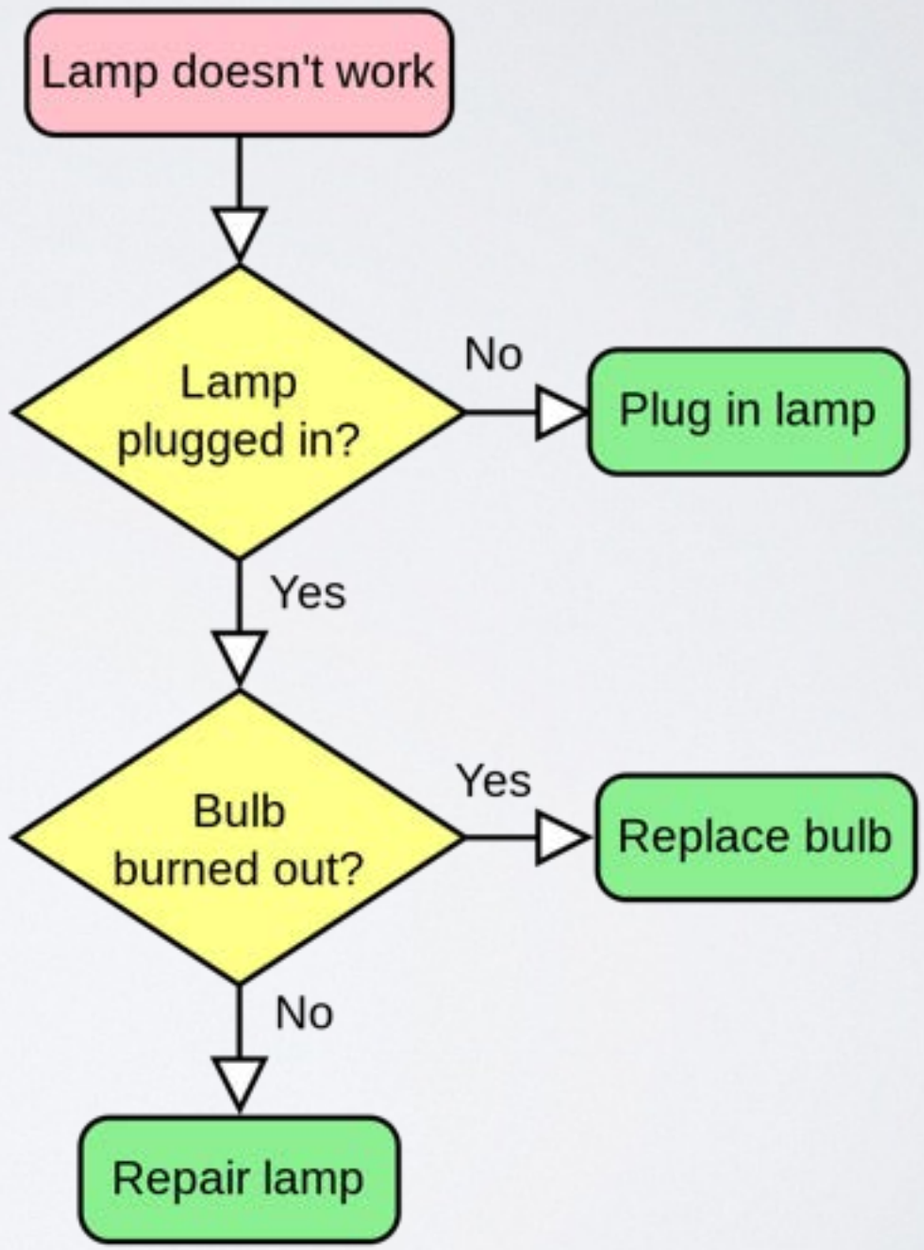
Specification techniques

Specification techniques

Discussion

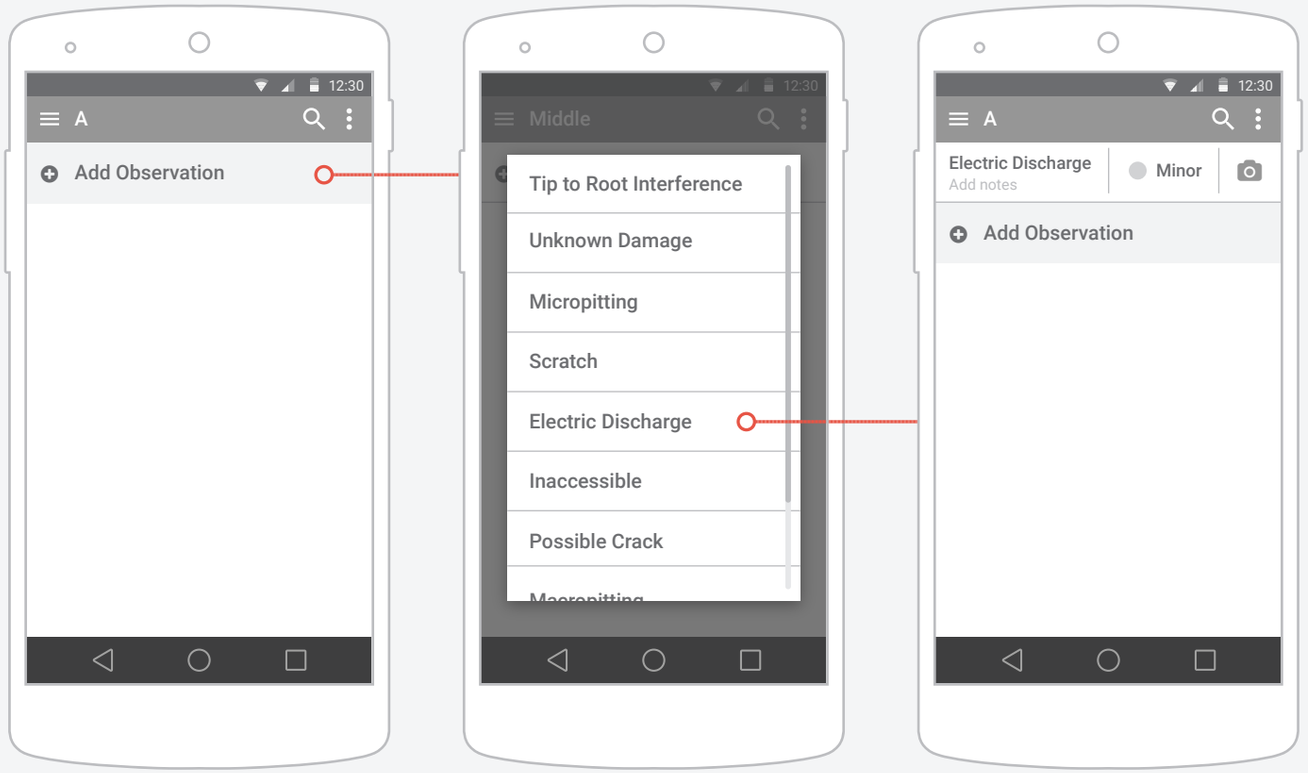
Specification techniques

Flow charts



Specification techniques

Screen flows



Add Observation
"Add Observation" button is available on components that allow open observations. Button always appears at the bottom of the observation list.

Select Observation Type
Modal overlay with scrolling list of open observations.

Specification techniques

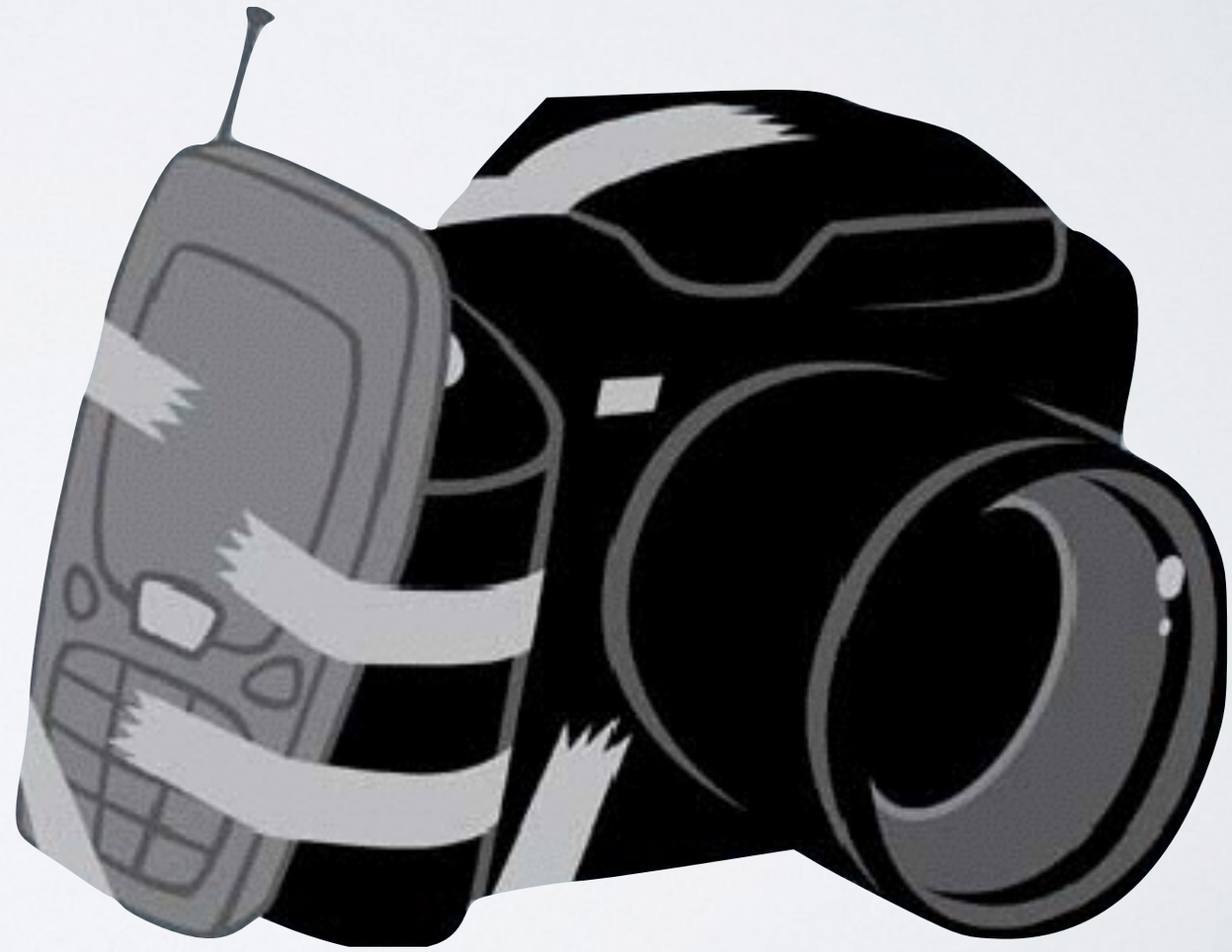
User stories

"Ted the technician wants to…"

"Mary the manager needs to…"

Specification techniques

Prototypes



Specification techniques

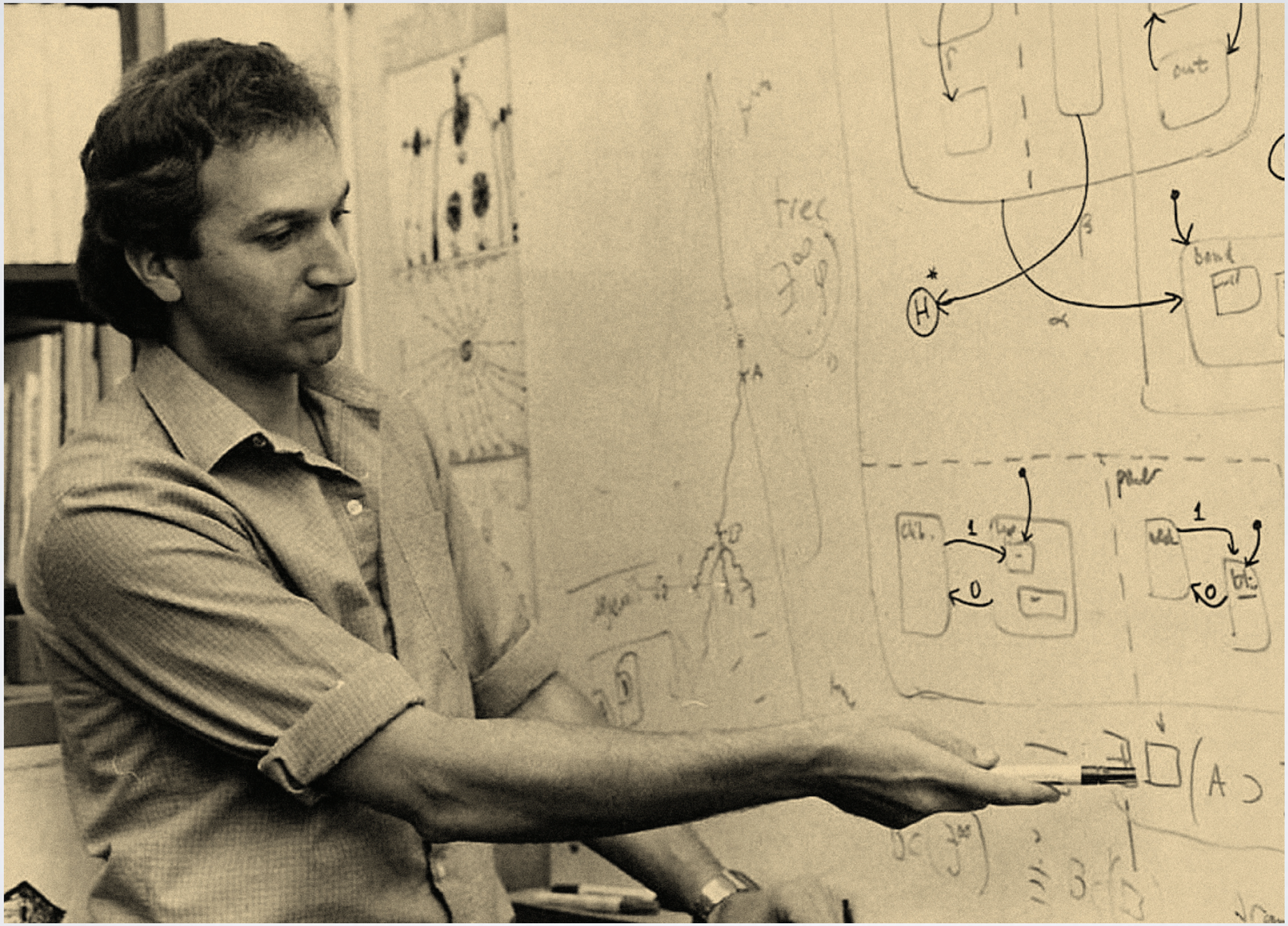
Specification techniques

Informal

Specification techniques

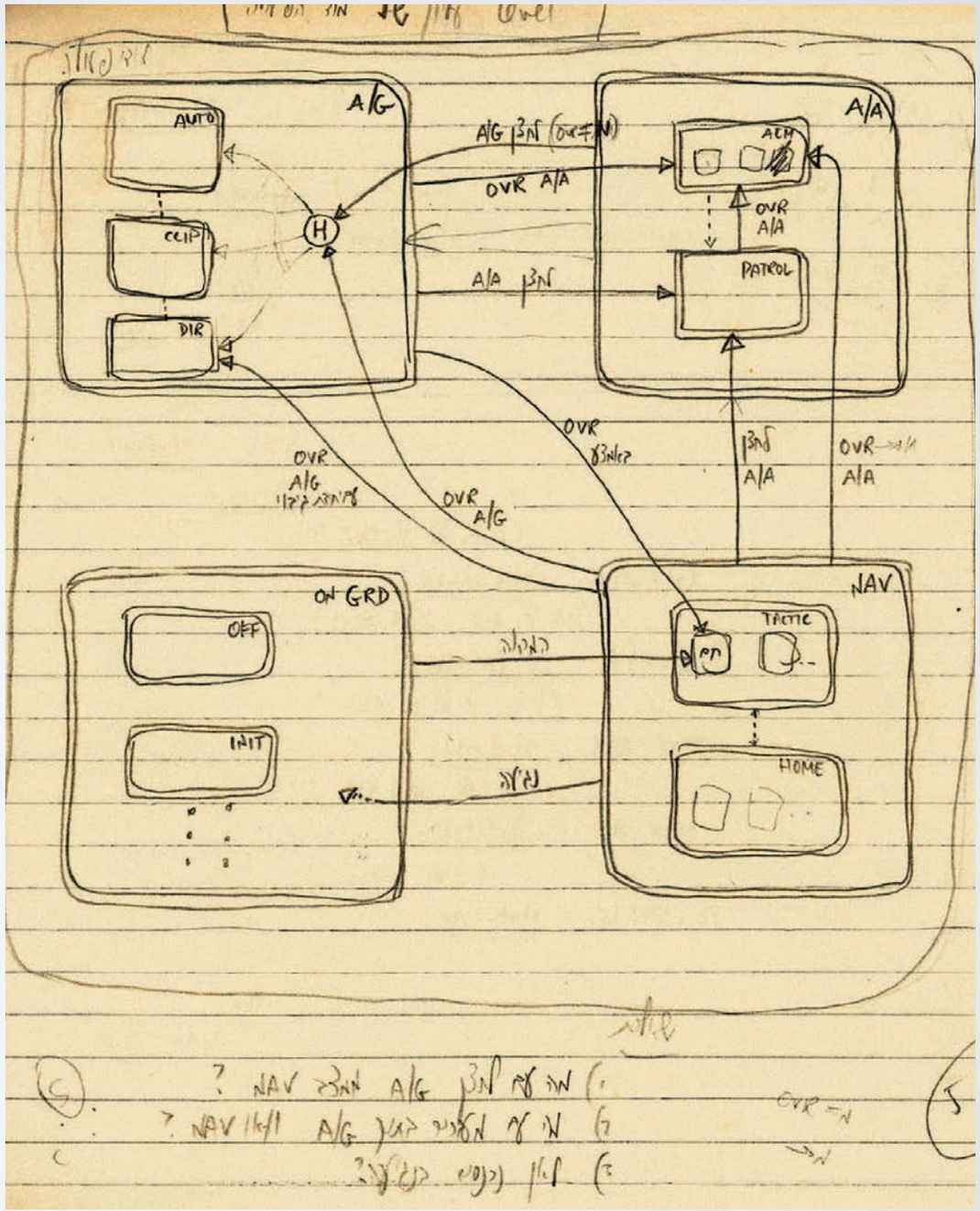
Formal

Harel's Statecharts



Back in the 80's

Harel's doodles



Science of Computer Programming 8 (1987) 231-274
North-Holland

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

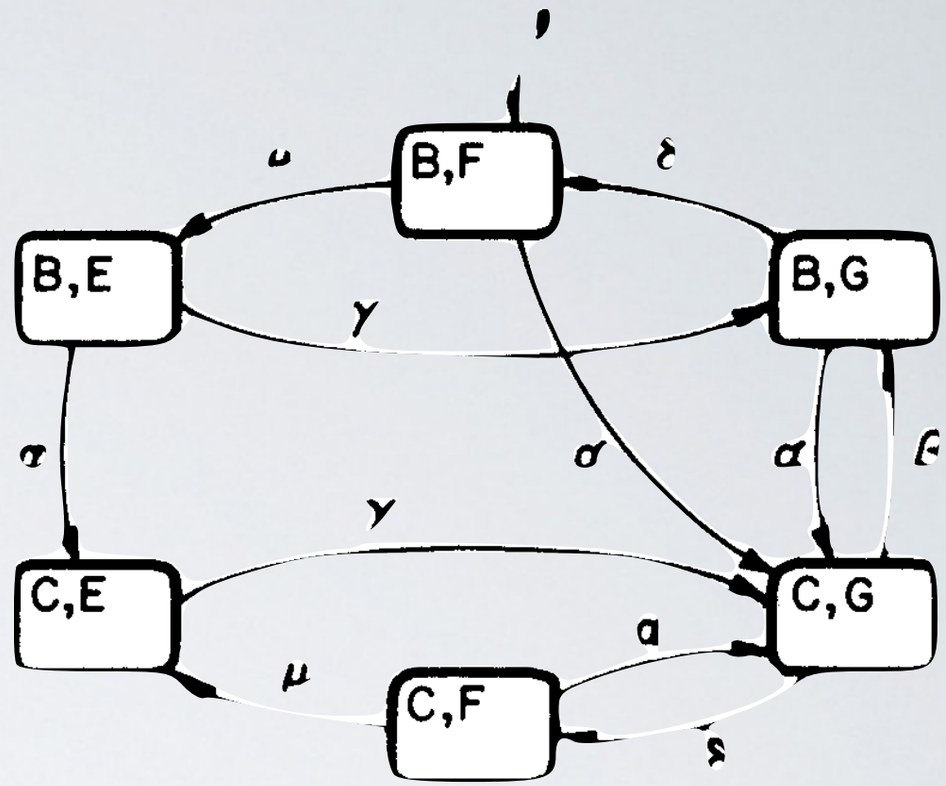
Communicated by A. Pnueli
Received December 1984
Revised July 1986

Abstract. We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call *statecharts*, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

1. Introduction

The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large and complex *reactive systems*. A reactive system (see [14]), in contrast with a *transformational system*, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and

* The initial part of this research was carried out while the author was consulting for the Research and Development Division of the Israel Aircraft Industries (IAI), Lod, Israel. Later stages were supported in part by grants from IAI and AD CAD, Ltd.



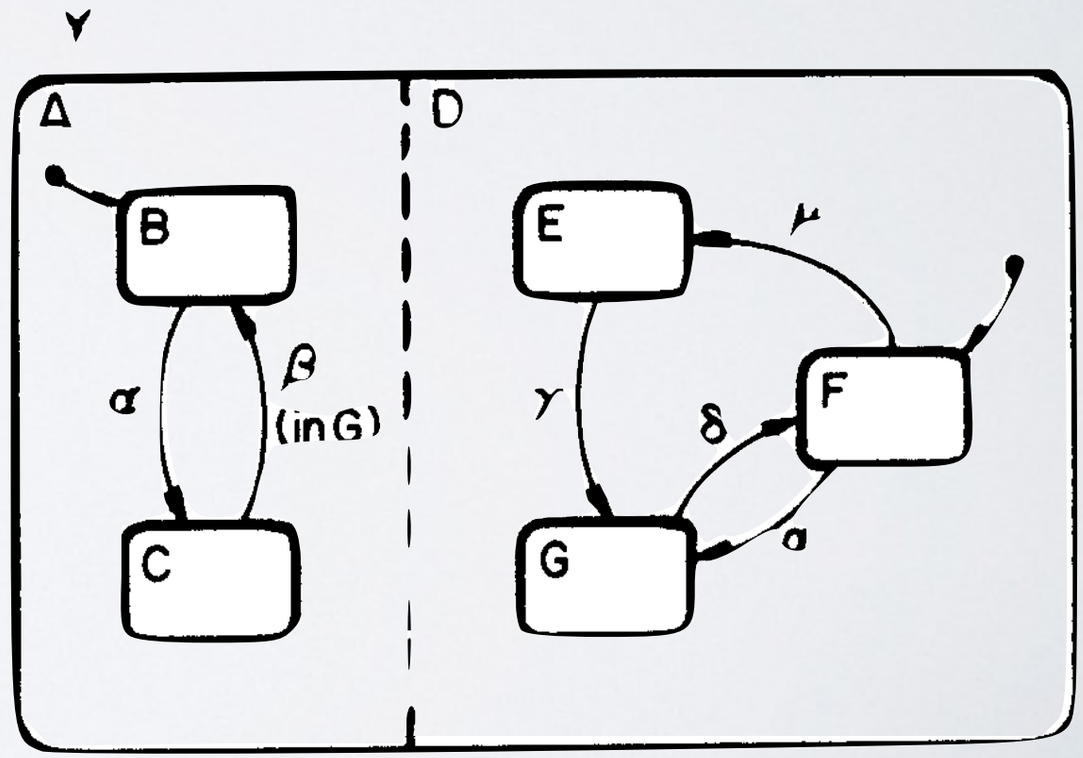
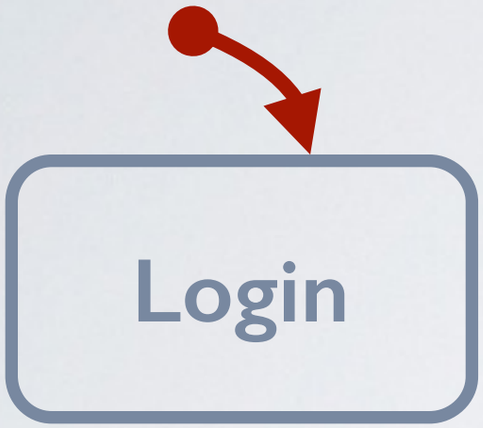


Fig. 10

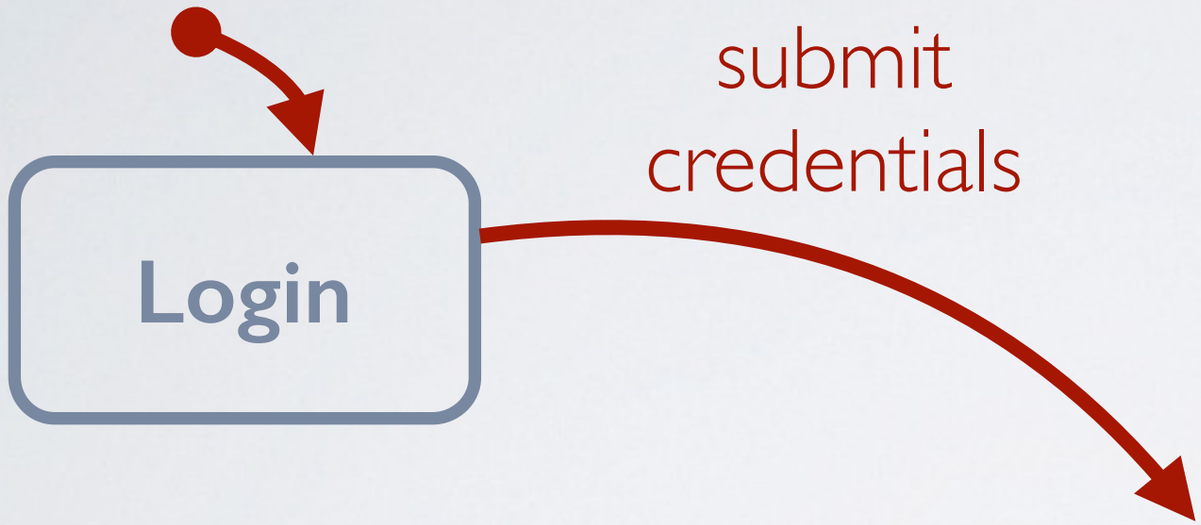
Statechart semantics

Statechart semantics

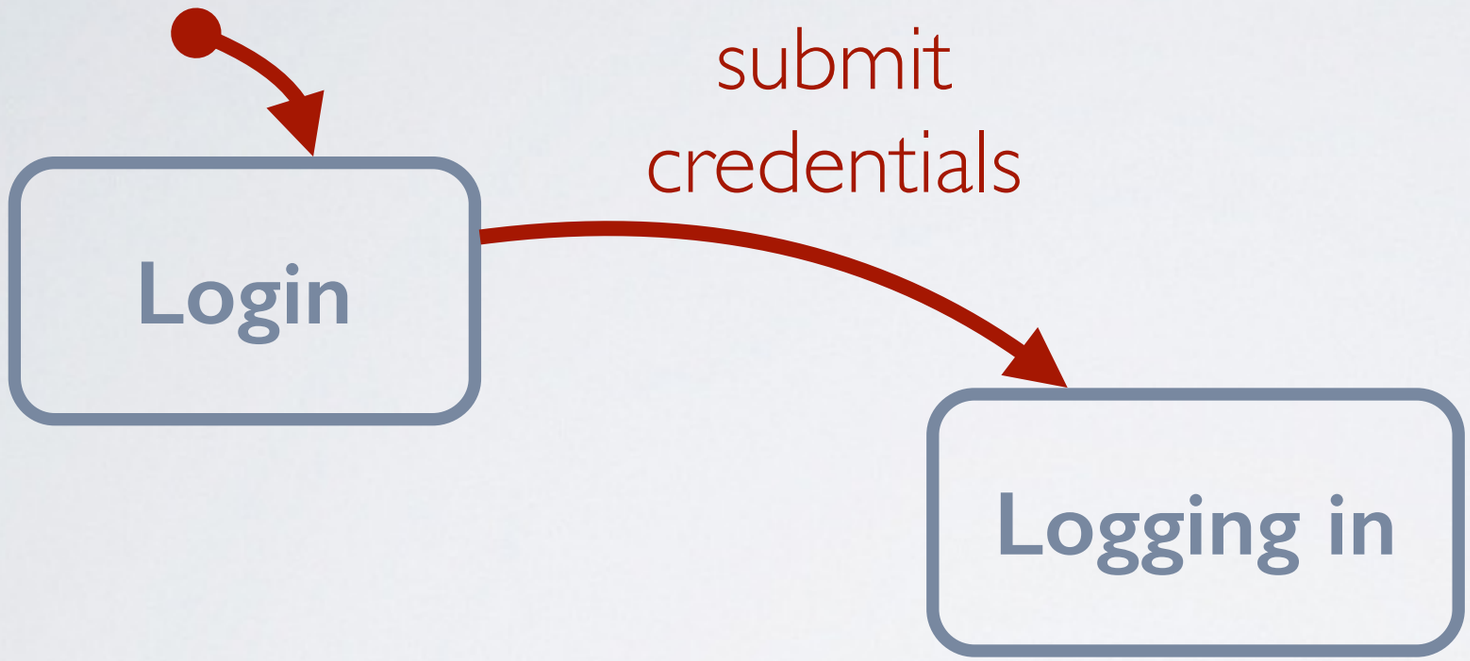
Statechart semantics



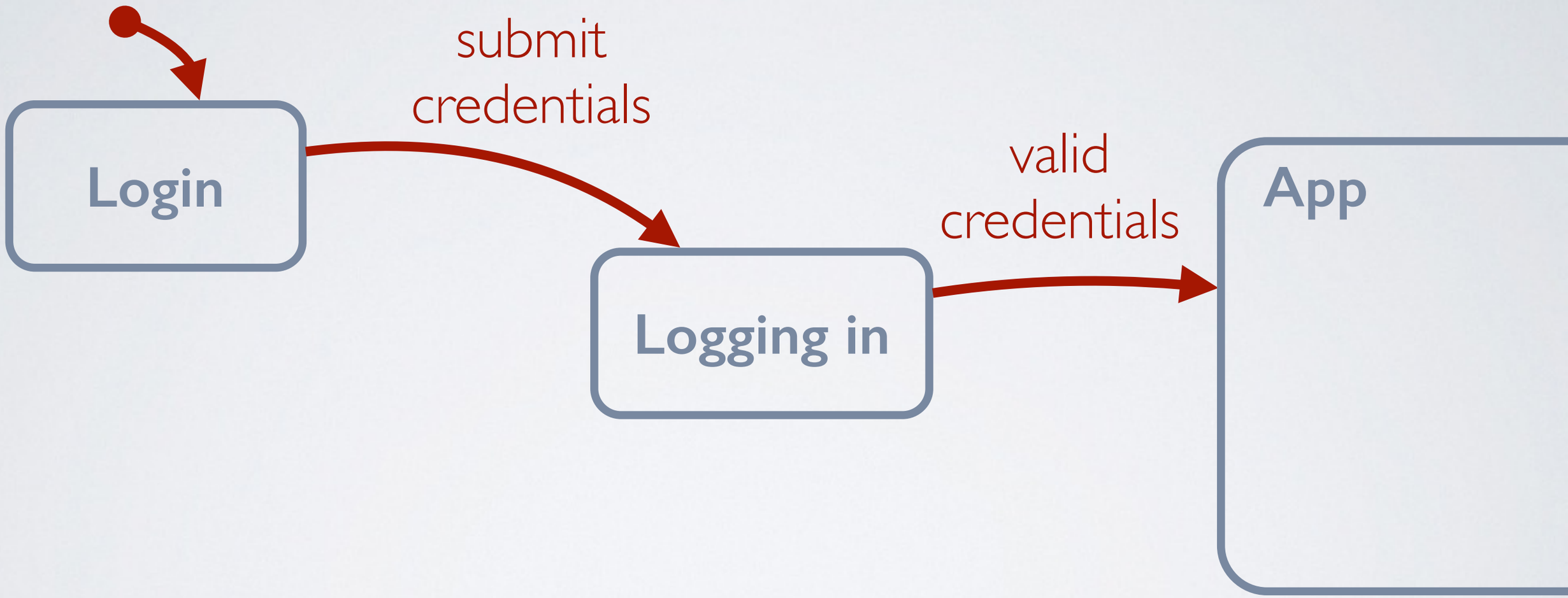
Statechart semantics



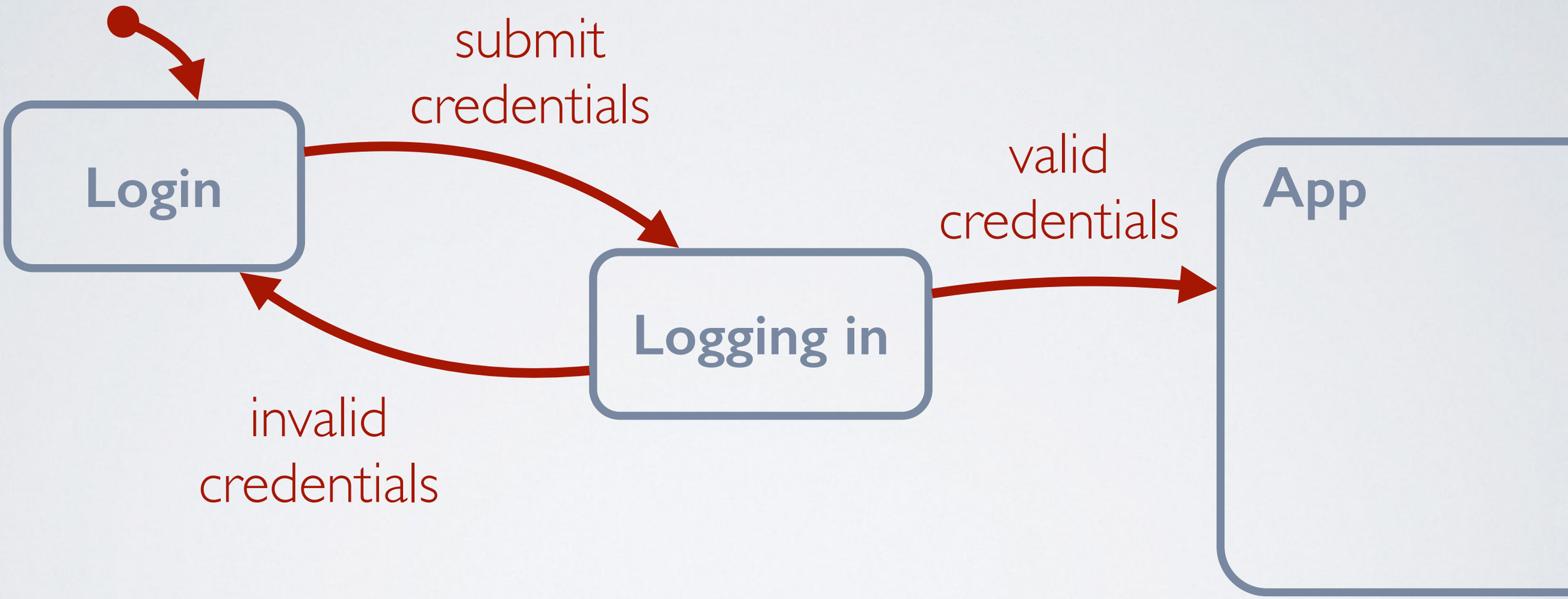
Statechart semantics



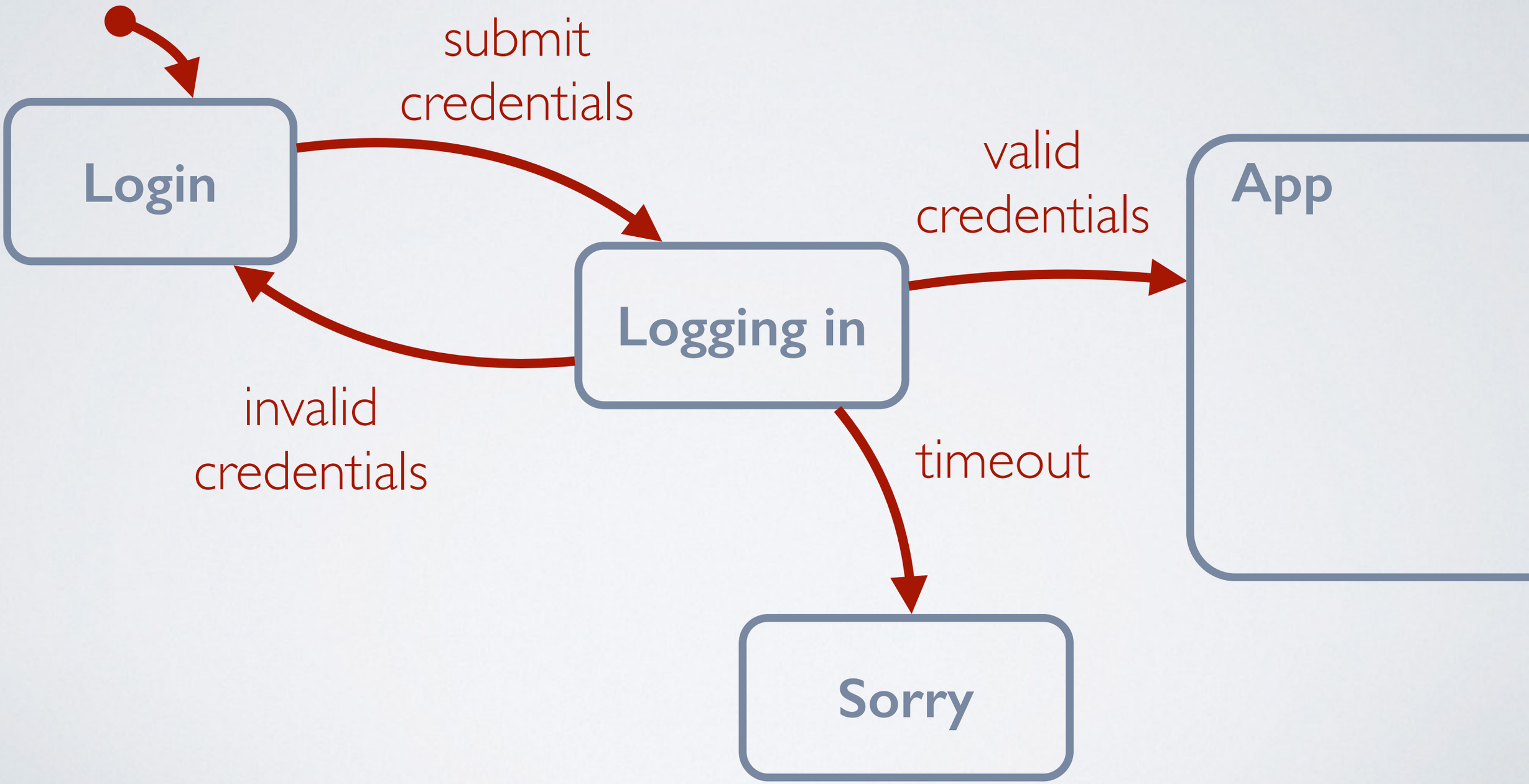
Statechart semantics



Statechart semantics



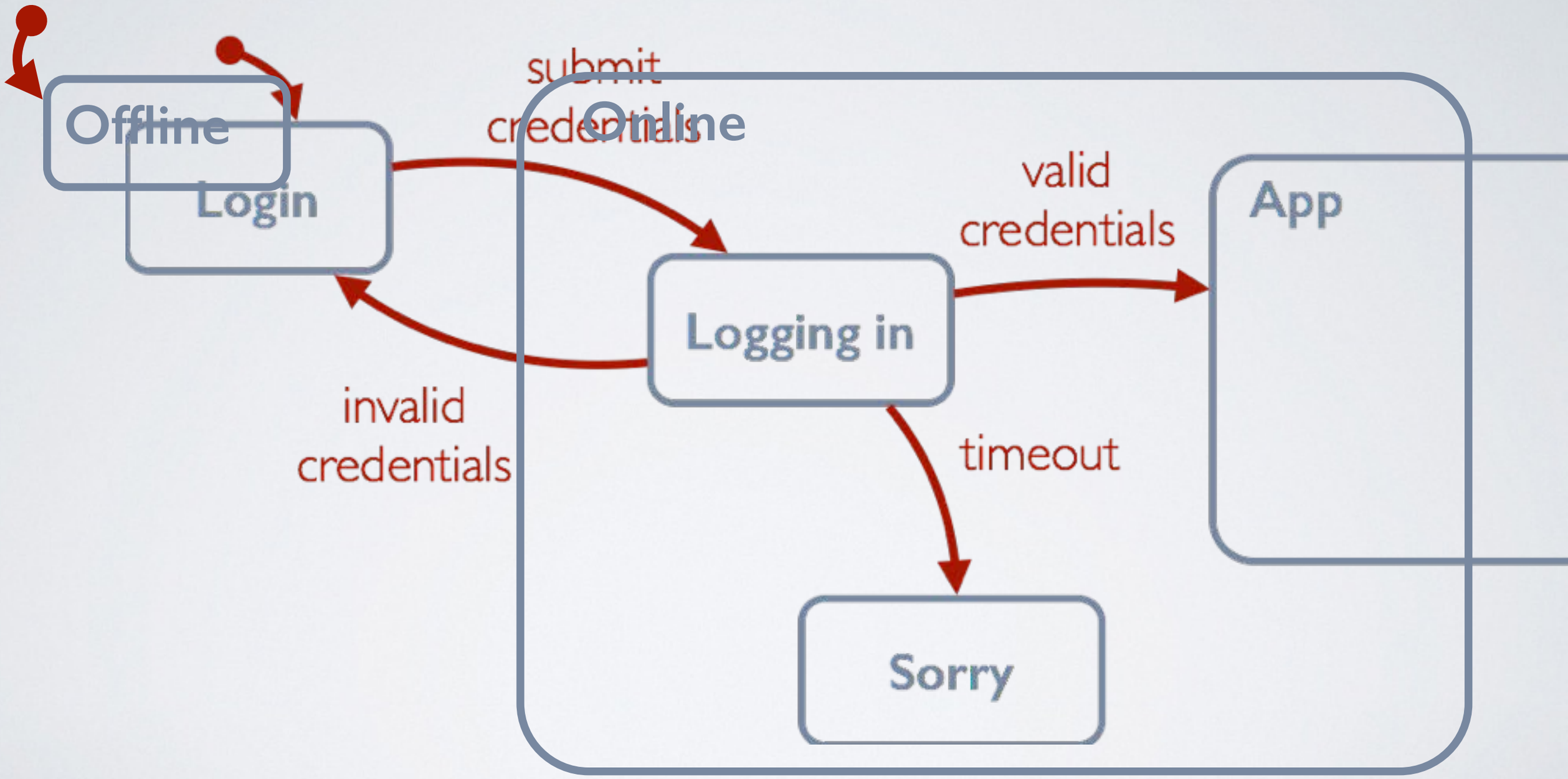
Statechart semantics



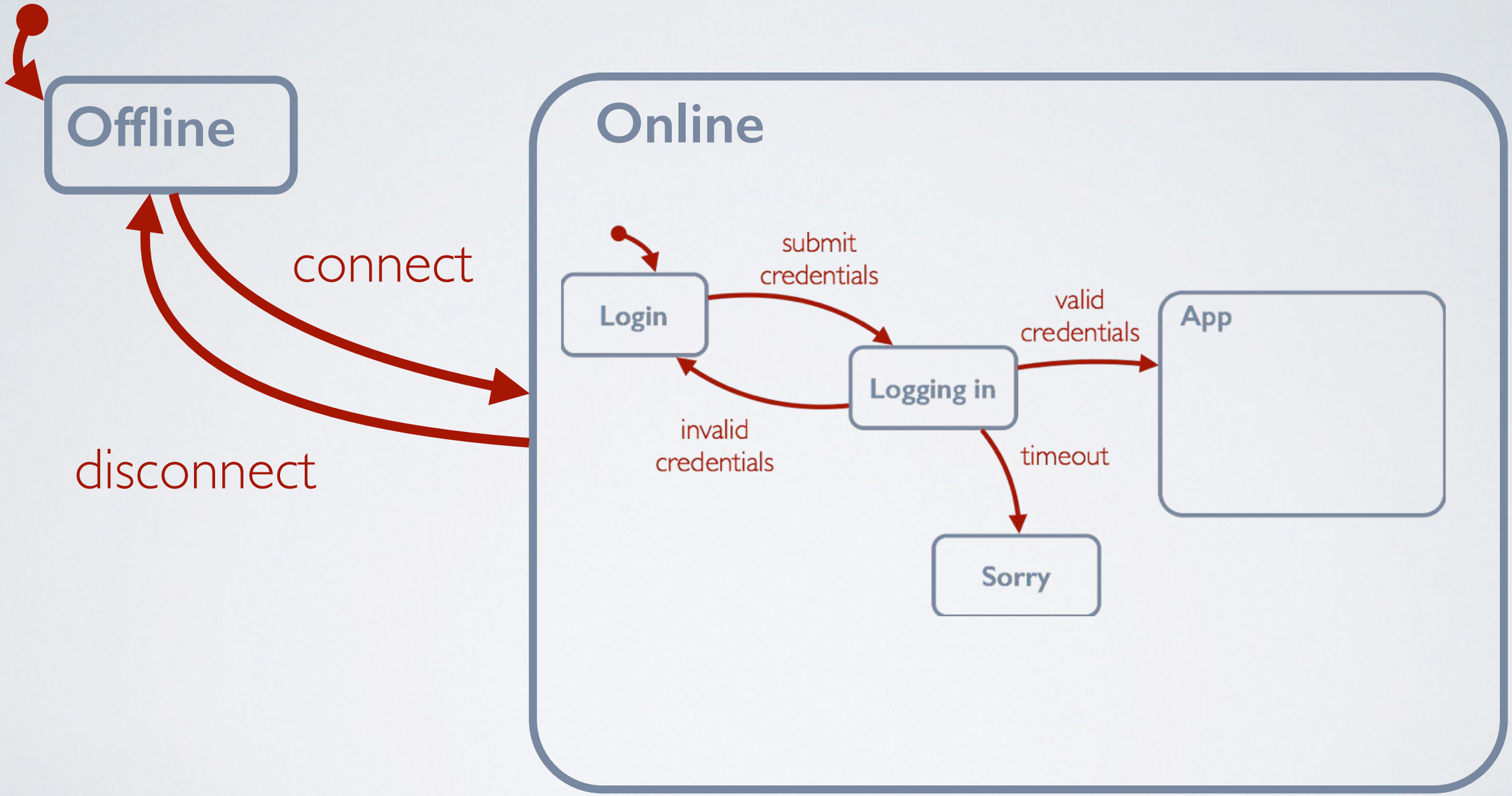
Statechart semantics

The user starts on the login screen.
They can enter some credentials and then hit the button to submit the credentials.
The user is now "logging in" and we are waiting for a response from the server.
If the server replies that the login is valid, the user is sent into the app.
If the server replies that the login is not valid, the user is sent back to the login screen with an error message.
There is also a 5 second timeout, so if we don't hear back from the server by then, the user is sent to a screen that explains that they need to be online to use this app.

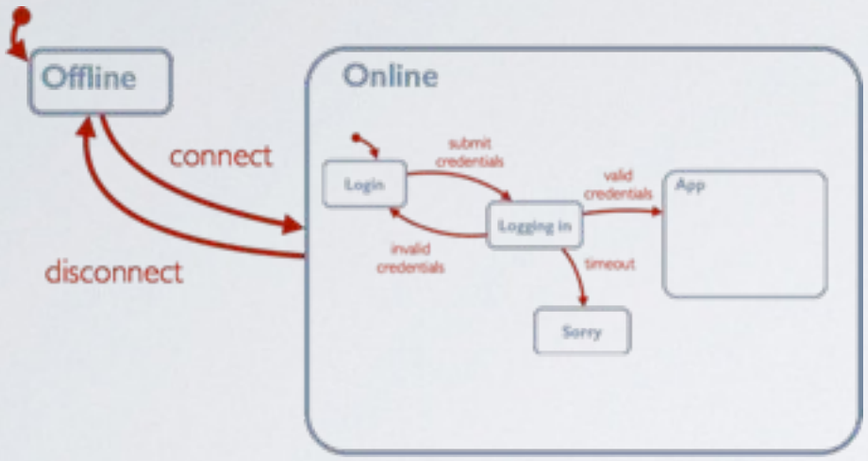
Statechart semantics



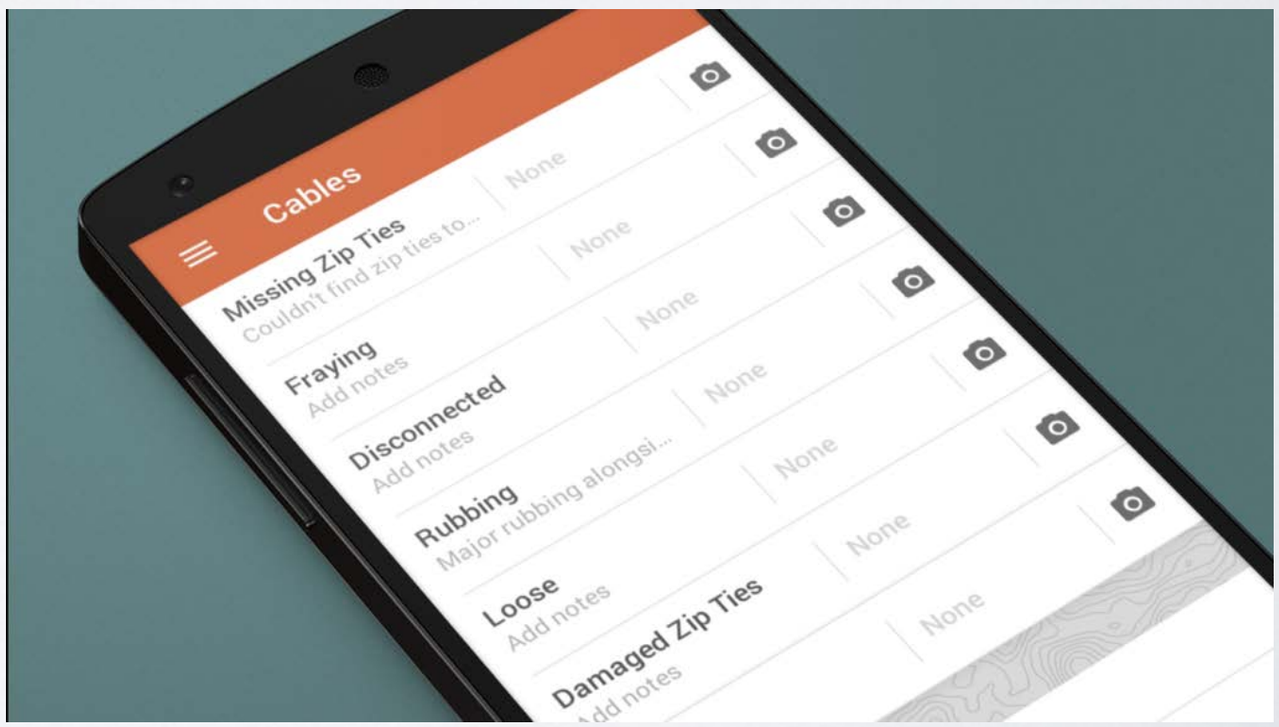
Statechart semantics



Implementing our app

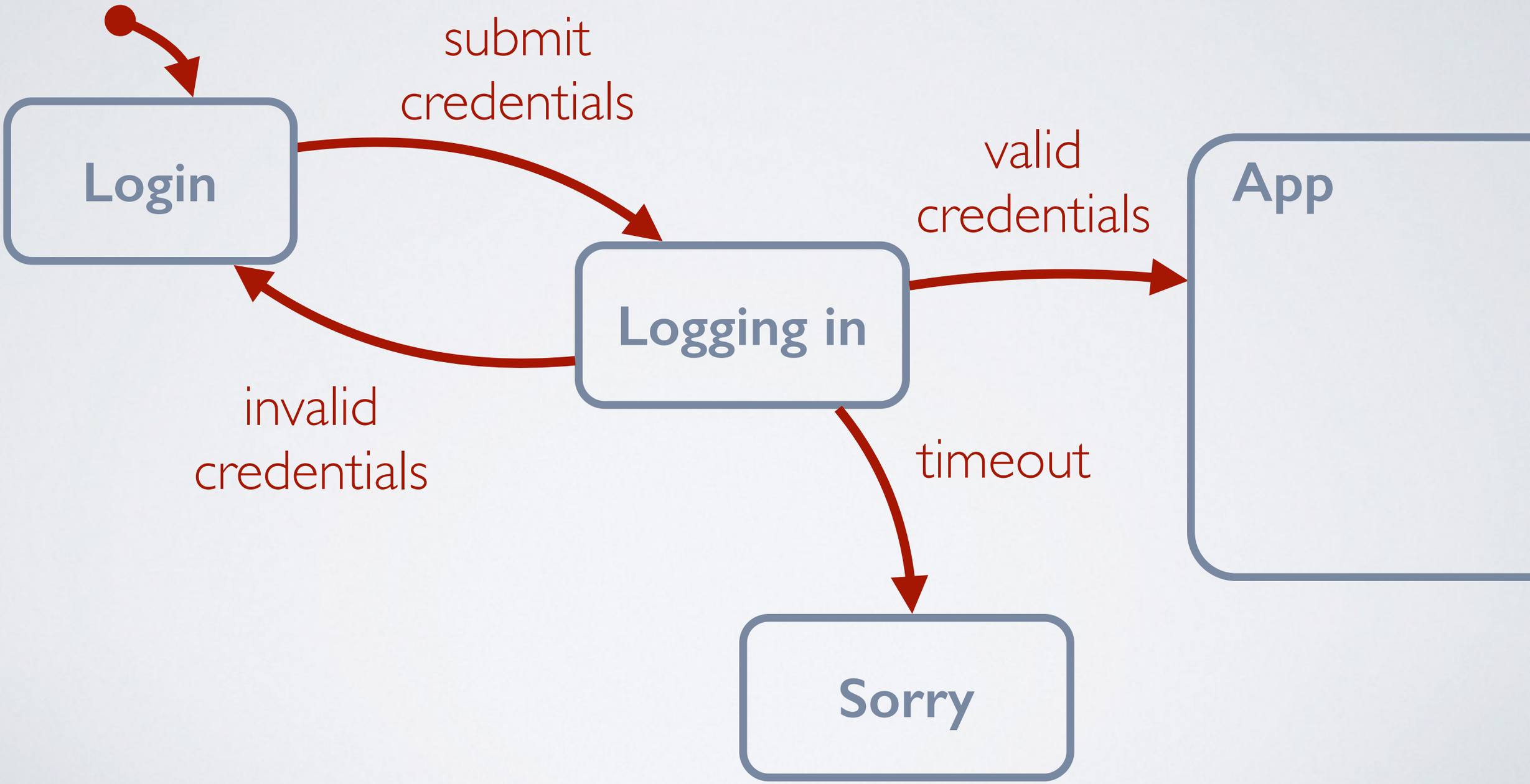


?



Implementation

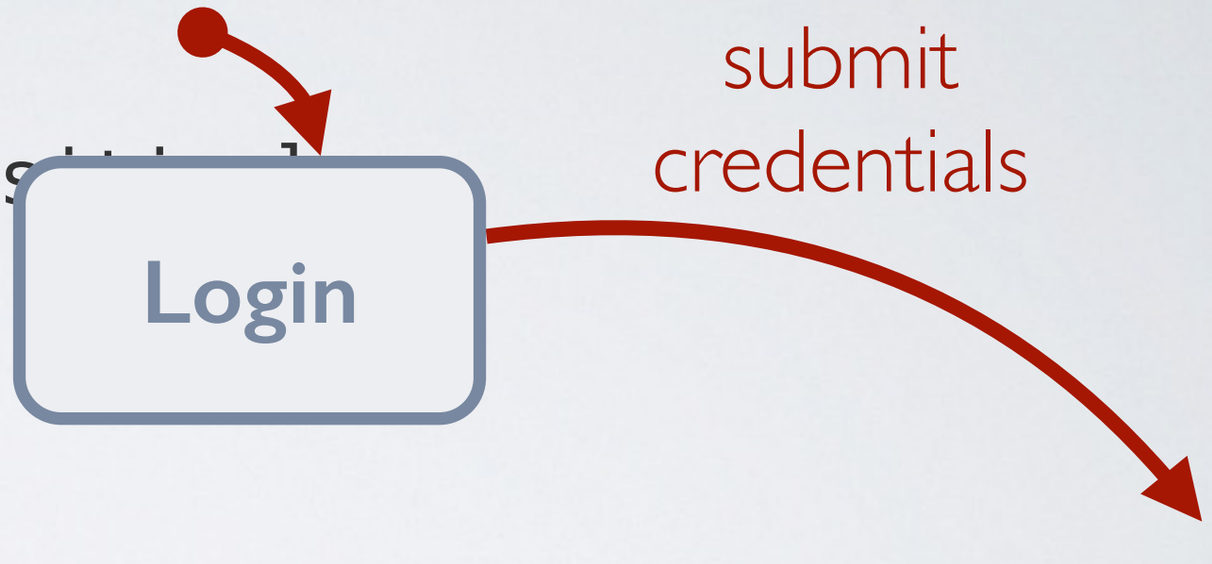
Lets start at “Login”



Login screen

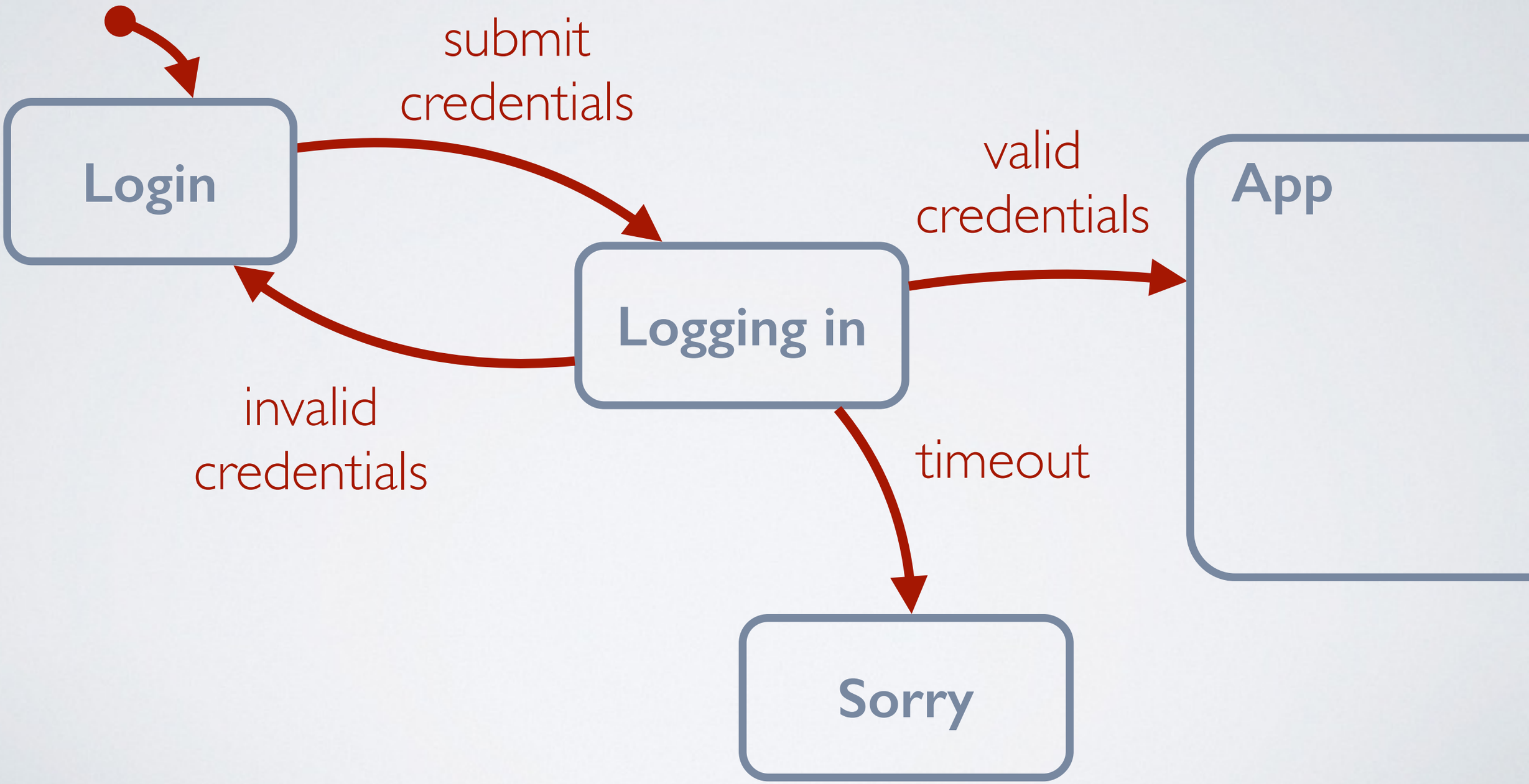
```
(defn login-template
  [username password transition]

  [:div
    [:h1 "Login Plz"]

    [:form
      {:on-submit (fn [e]
                    (transition "submit credentials"
                                username password))}

      [:input {:type "text" :value username}]
      [:input {:type "password" :value password}]
      [:input {:type "submit"} "Login"]]])
```

Transition fn from statechart




```
(defn login-template
  [username password trans
   [:div
     [:h1 "Login Plz"]
     [:form
       {:on-submit (fn [e]
                     (transition "submit credentials"
                                 username password))}
       [:input {:type "text" :value username}]
       [:input {:type "password" :value password}]
       [:input {:type "submit"} "Login"]]]])
```

Similar states share templates



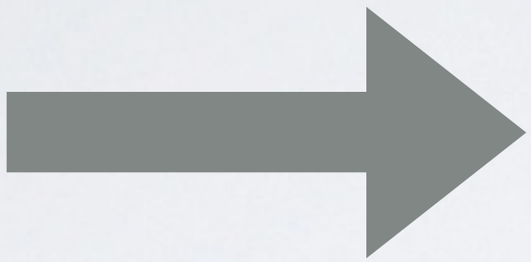
Similar states share templates

```
(defn login-template
  [state username password transition]

  [:div
    [:h1 "Login Plz"]

  [:form
    {:on-submit (fn [e]
                  (transition "submit credentials"
                              username password))}

    [:input {:type "text" :value username
             :disabled (when (= "logging in" state) true)}]
    [:input {:type "password" :value password
             :disabled (when (= "logging in" state) true)}]
    [:input {:type "submit"
             :disabled (when (= "logging in" state) true)}
      state]]])
```

Similar states share templates

```
(defn login-template
  [state username password transition]

  [:div
    [:h1 "Login Plz"]

    [:form
      {:on-submit (fn [e]
                    (transition "submit credentials"
                                username password))}

      [:input {:type "text" :value username
               :disabled (when (= "logging in" state) true)}]
      [:input {:type "password" :value password
               :disabled (when (= "logging in" state) true)}]
      [:input {:type "submit"
               :disabled (when (= "logging in" state) true)}
        state]]])
```

“Templates” = Functions



Reuse + abstraction

Things you can take home

Designers love it!

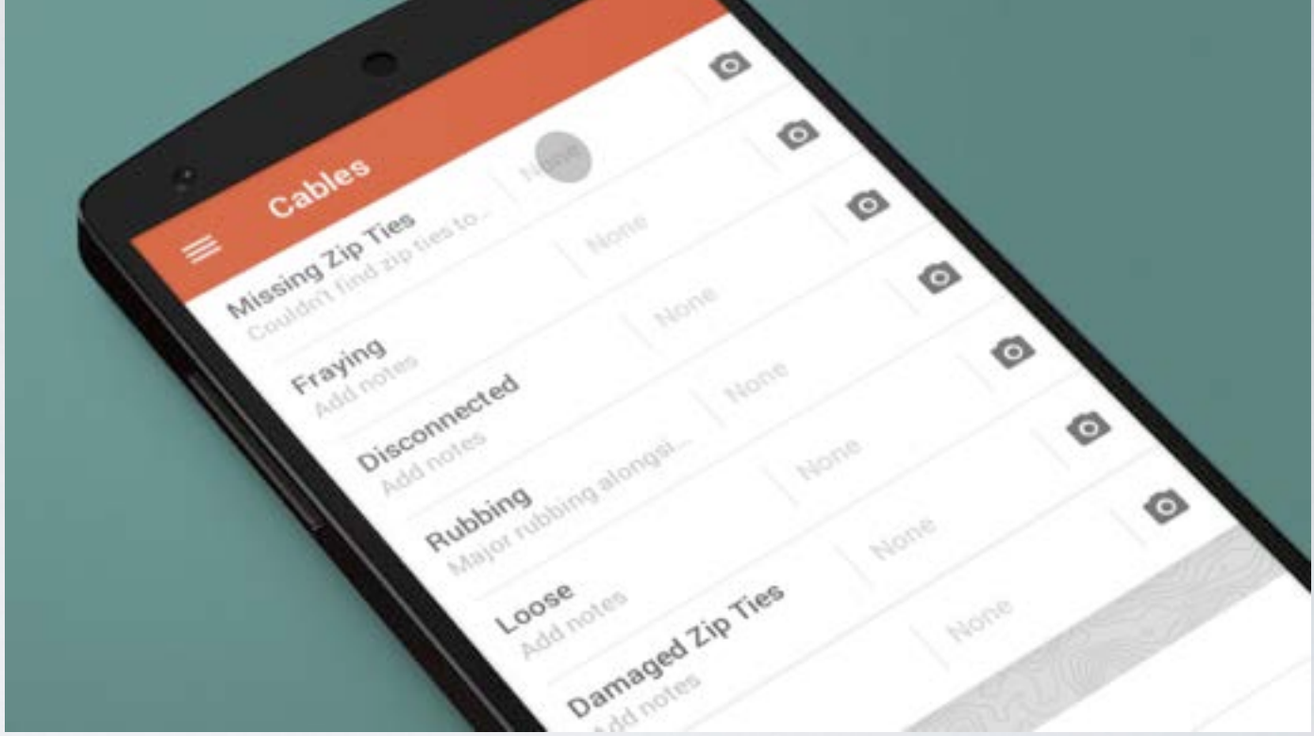
```
[:html
  [:body
    (login-template "login"         "Mr. Shortname"                "******")
    (login-template "logging in"    "Mr. Shortname"                "******")
    (login-template “login failed" "Mr. Shortname"                 "******")
    (login-template "login"         "Mrs. very super long name" "******")
    (login-template "login"         ""                             "")]]
```

Designers love it!

```
[:html
  [:body
    (login-template "login"         "Mr. Shortname"                "******")
    (login-template "logging in"    "Mr. Shortname"                "******")
    (login-template “login failed" "Mr. Shortname"                 "******")
    (login-template "login"         "Mrs. very super long name" "******")
    (login-template "login"         ""                             "")]]
```

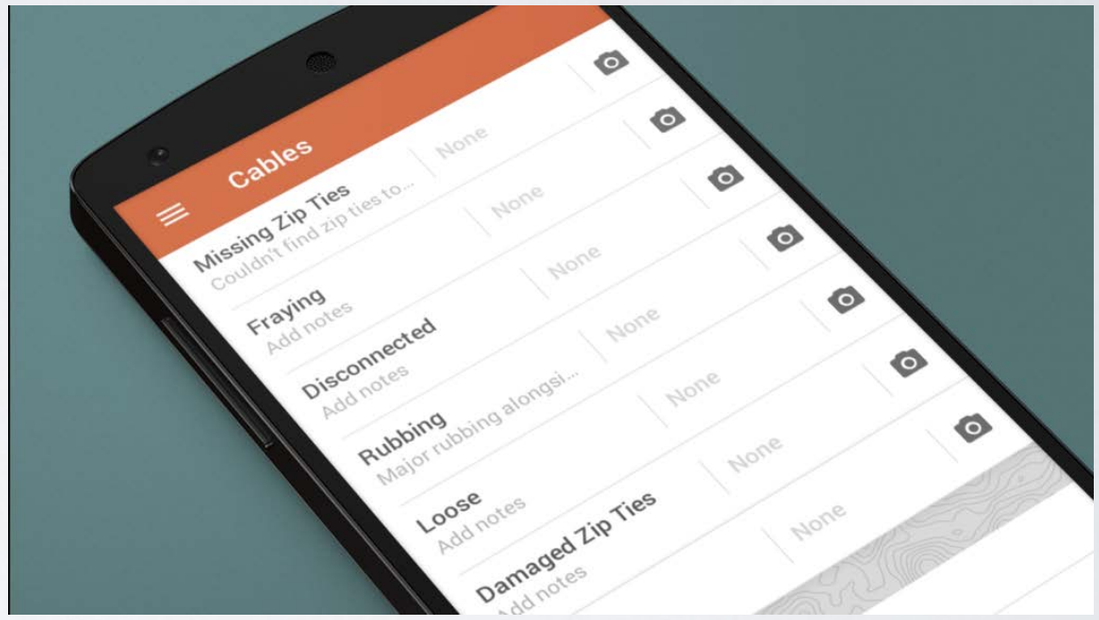
Composition

```
(defn checklist-item
  [{:keys [path name note value]
    :as op}
   ctrl]

  [:li
    [:p.name name]
   [:p.note
    {:on-click #(ctrl "expand note" {:path path :note note})}
    (or note "Add notes")]

   [:.value
    (observation-point-selector op ctrl
      #(ctrl "update" {:path (get-in @op [:observation :path])
                       :update %}))]

   (photo-icon ctrl path op)]
```



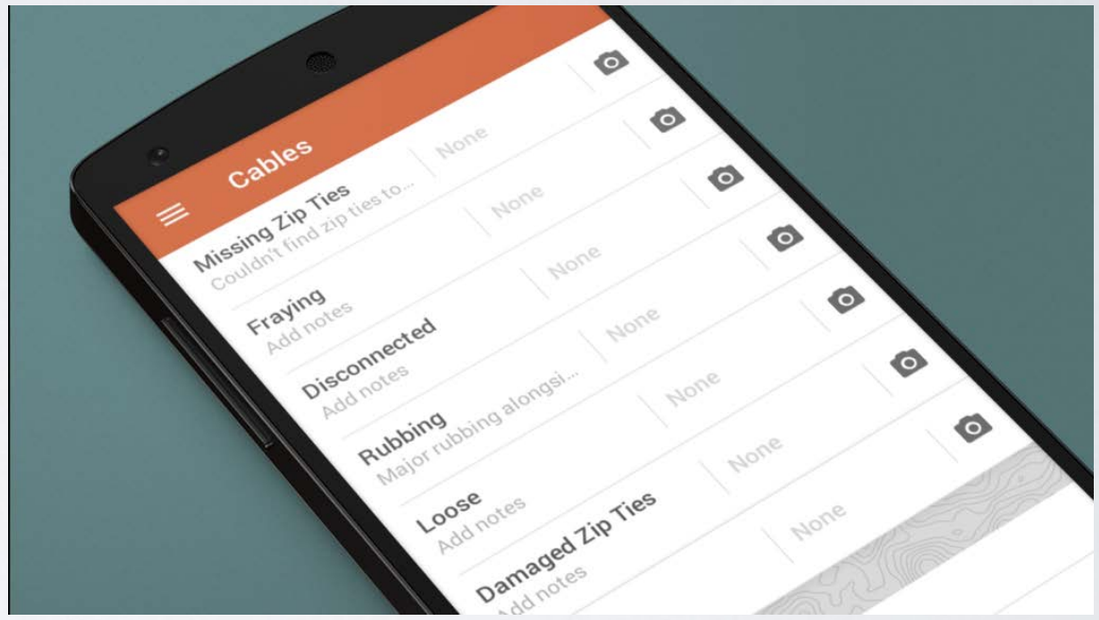
Functions are real nice

$$(f\ x) =$$



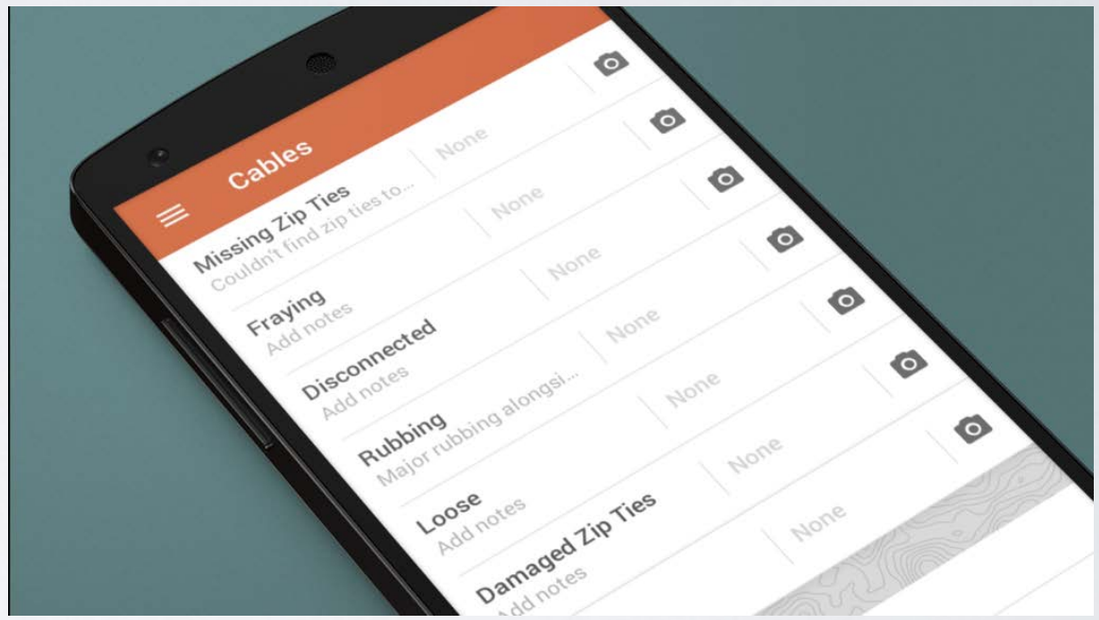
The entire UI can be a single function

$$(f\ x) =$$



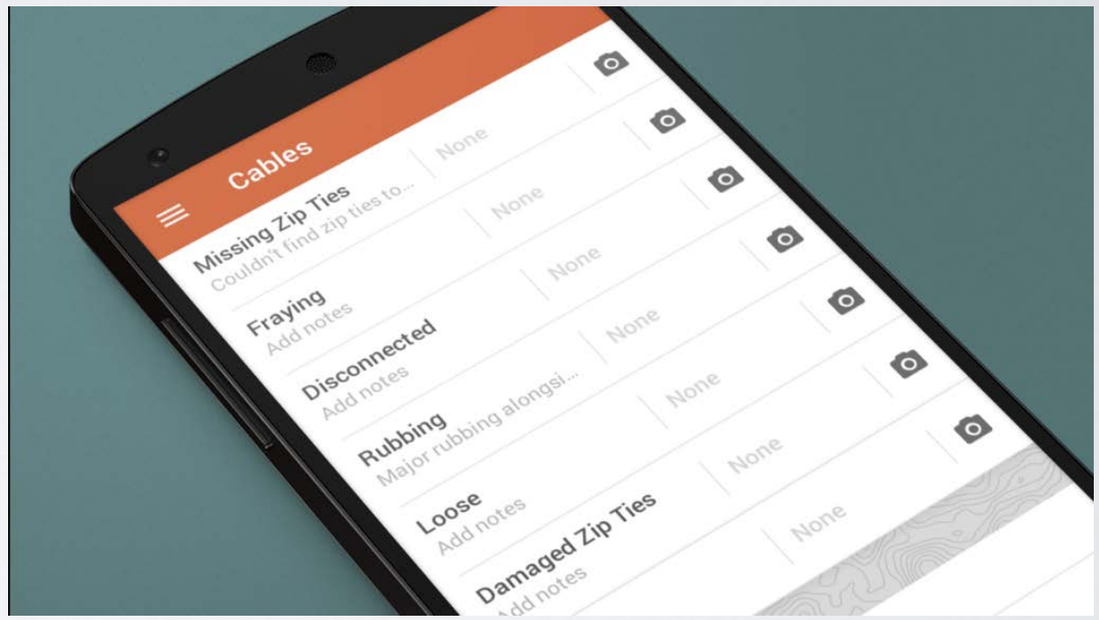
“Immediate mode” UI

(render x) =



Explicit application state

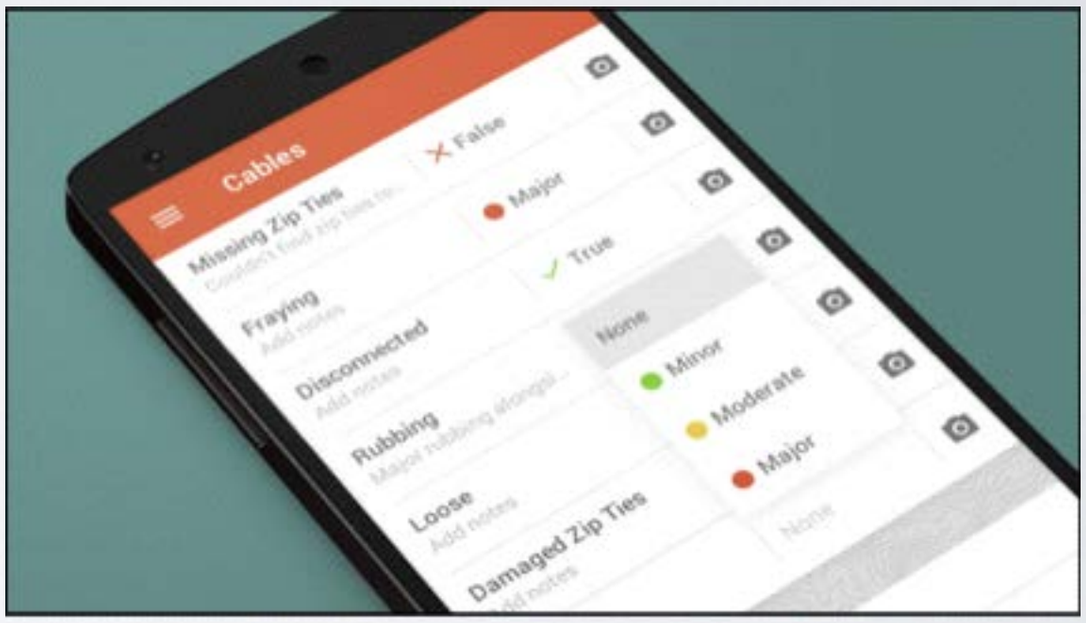
(render x) =



?

Explicit application state

(render x′) =



?

Explicit application state

```
{:state "login"
 :login {:username "foo"
         :password "bar"}
 :inspecting {…}
 …}
```

Explicit application state

An immutable graph database

Benefits of explicit app state

Benefits of explicit app state

Viewable

```
{:state "login"
 :login {:username "foo"
         :password "bar"}
 :inspecting {…}
 …}
```

Benefits of explicit app state

Serializable

```
{:state "login"
 :login {:username "foo"
         :password "bar"}
 :inspecting {…}
 …}
```

Benefits of explicit app state

Reloadable

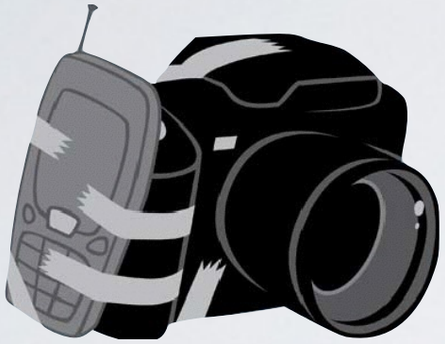
Benefits of explicit app state

Undo

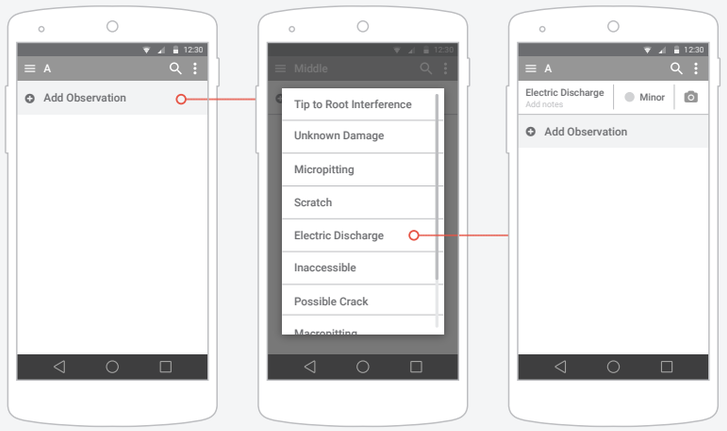
Wrap up

Specs are tricky…

Prototype



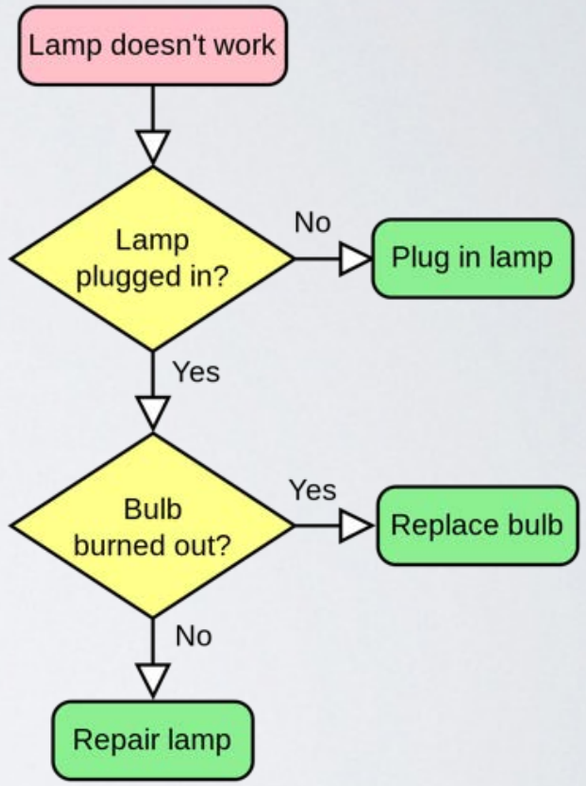
Screen flows



Add Observation
"Add Observation" button is available on components that allow open observations. Button always appears at the bottom of the observation list.

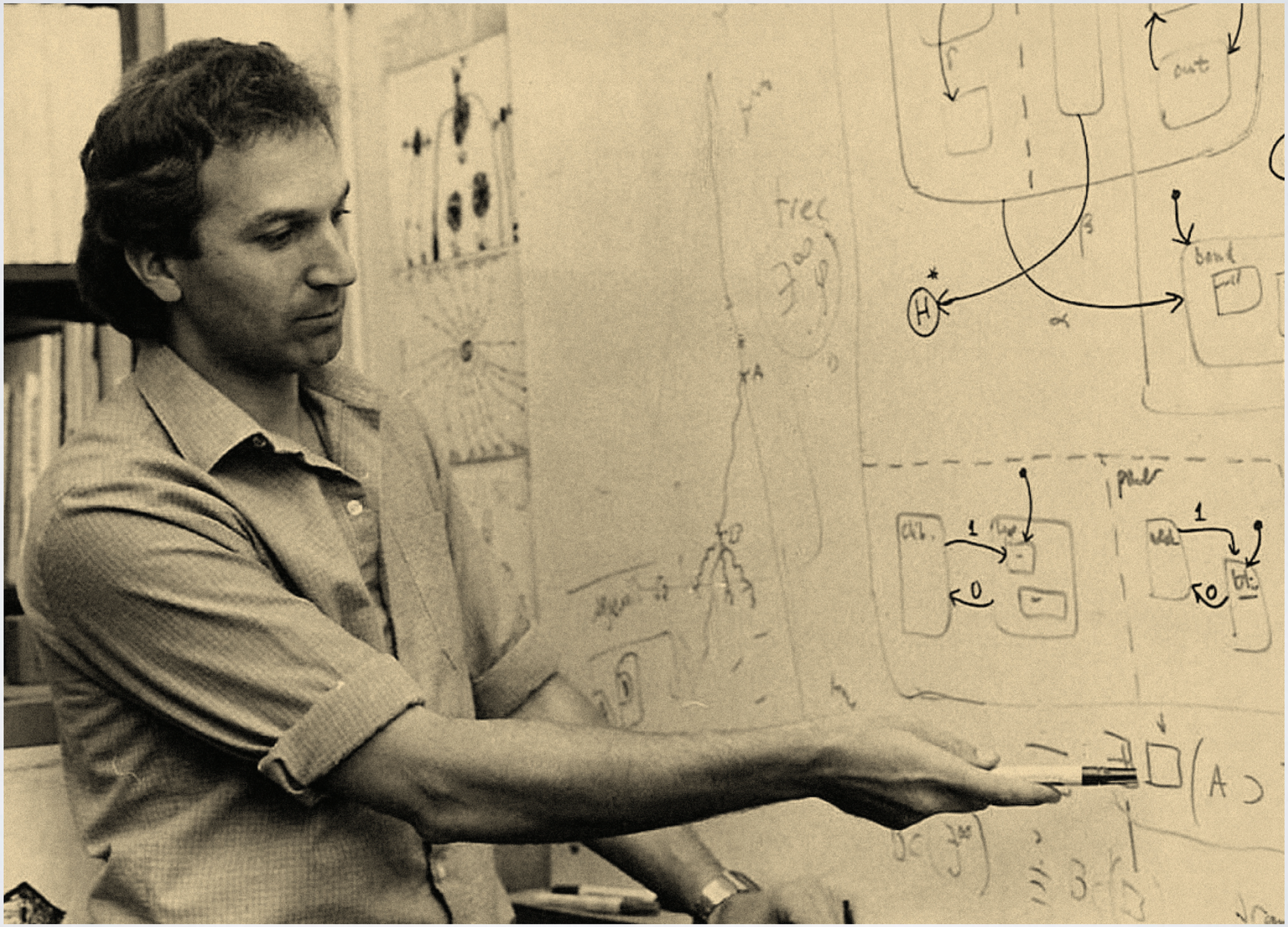
Select Observation Type
Modal overlay with scrolling list of open observations.

Flow charts

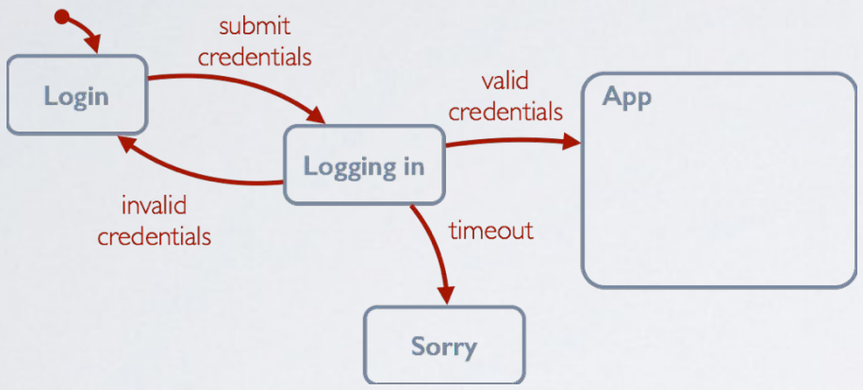


Discussions

Statecharts = formal specs



Model state explicitly

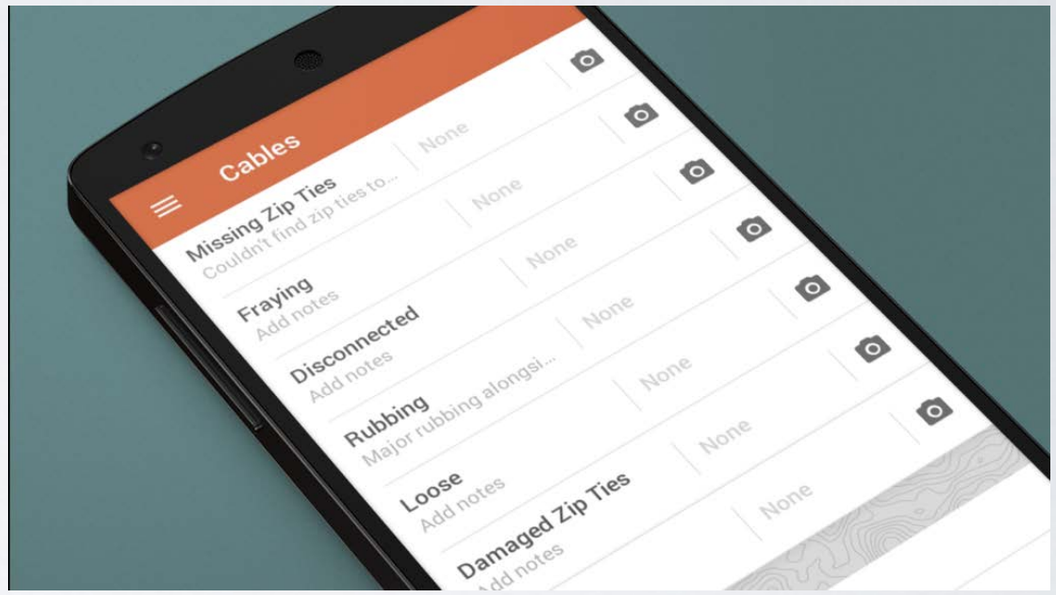


$\equiv$ *transition*

$$(transition\ x\ \text{“some event”}) \Longrightarrow x'$$

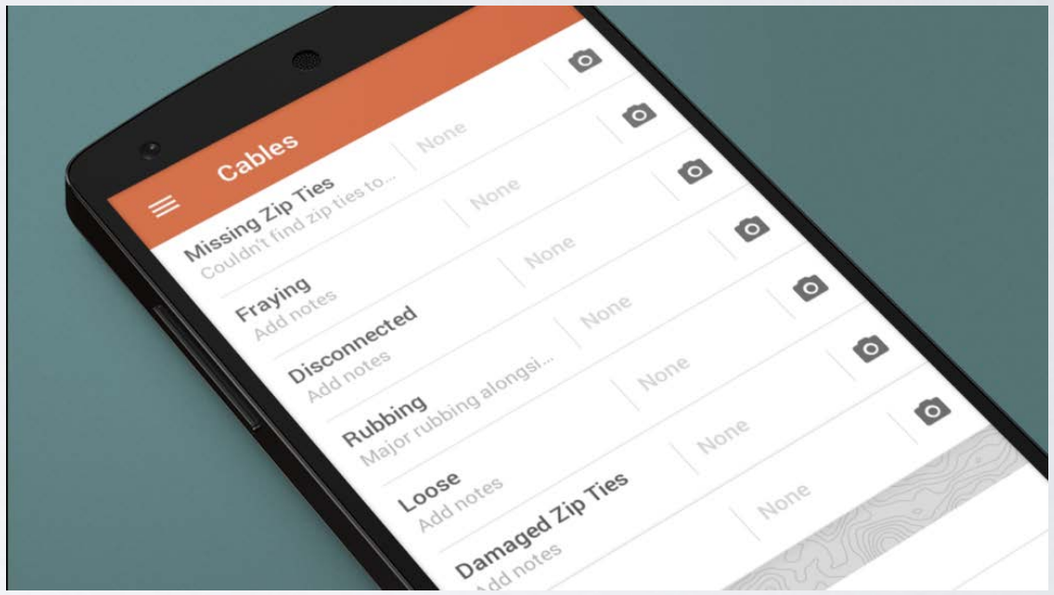
Build your UI as a function

(render x) =



Functions are awesome!

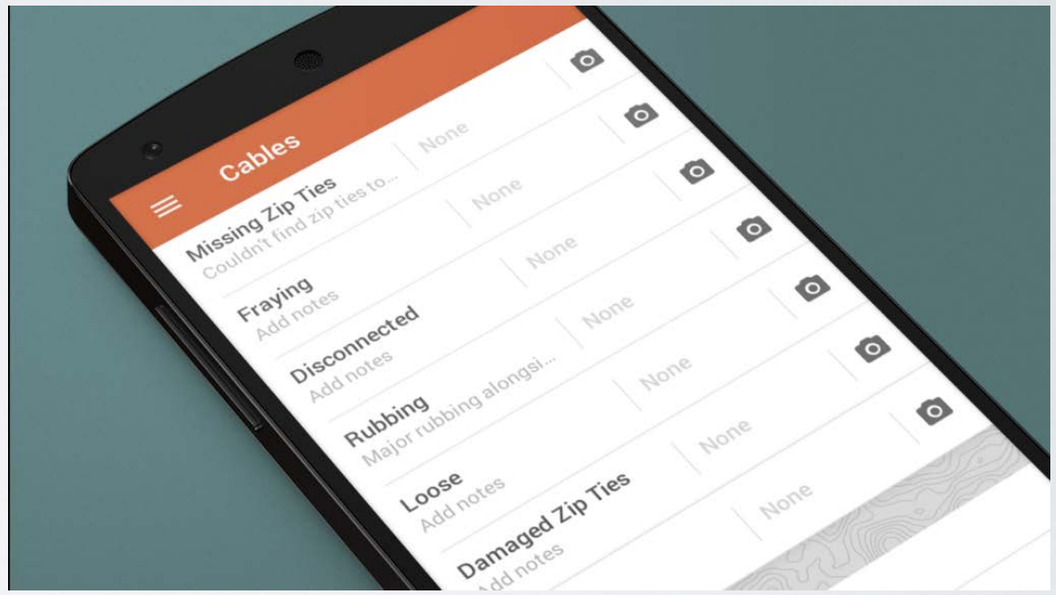
(render x) =



- **Reuse**
- **Composition**
- **Testing**

Explicit data are awesome!

(render x) =



- **Introspection**
- **Serializability**
- **Reloadable**
- **Undo**

Thanks!

Kevin Lynagh **@lynaghk**
Keming Labs
https://keminglabs.com/talks/