# Theater Framework

Dario A Lencina-Talarico

November 30, 2015

# Agenda

1. Why building Theater Framework?
2. The Actor Model
3. **Exercise 1**: Ping Pong
4. **Exercise 2**: Greeting Actor
5. **Exercise 3**: Turnstile
6. Conclusions
7. How to get started
8. How to get involved

# Why building Theater Framework?

- Writing apps that perform complex tasks such as bank transactions and vehicle interactions often result in code that is “hard to understand” and packed with obscure bugs that take a long time to catch.

# Why building Theater Framework?

Complex ≠ Hard

A. Complex: composed of many interconnected parts; compound.

B. Hard: difficult to do or accomplish; fatiguing; troublesome.

# Why building Theater Framework?

- It is possible to create apps that accomplish complex tasks, but are easy to read and understand.

- To do that, we need to use the right tools and operate at the right abstraction level.

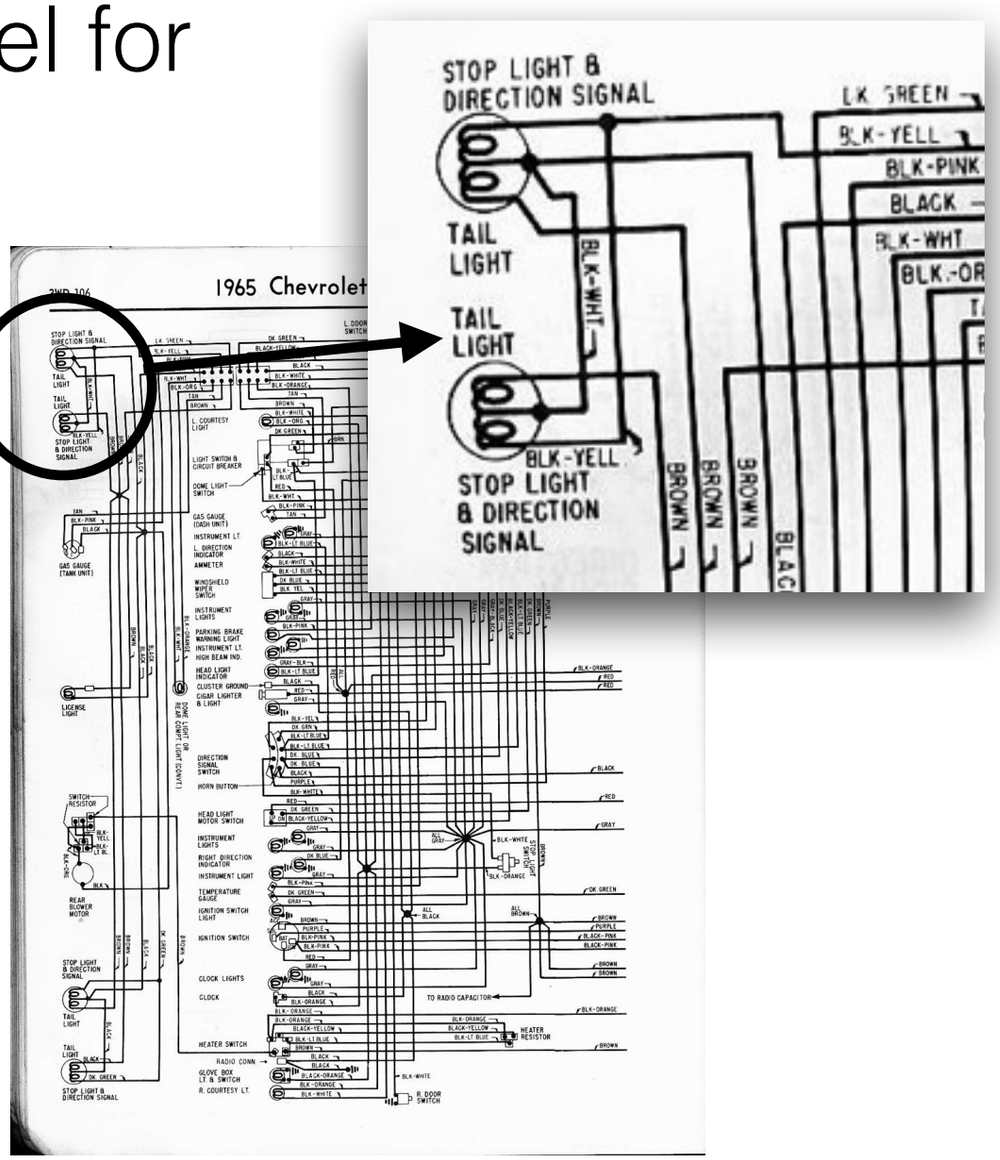
# Why building Theater Framework?

Let’s say that I just bought a 1965 Corvette…

# Why building Theater Framework?

What is the right abstraction level for driving a Corvette?

VS




**A**

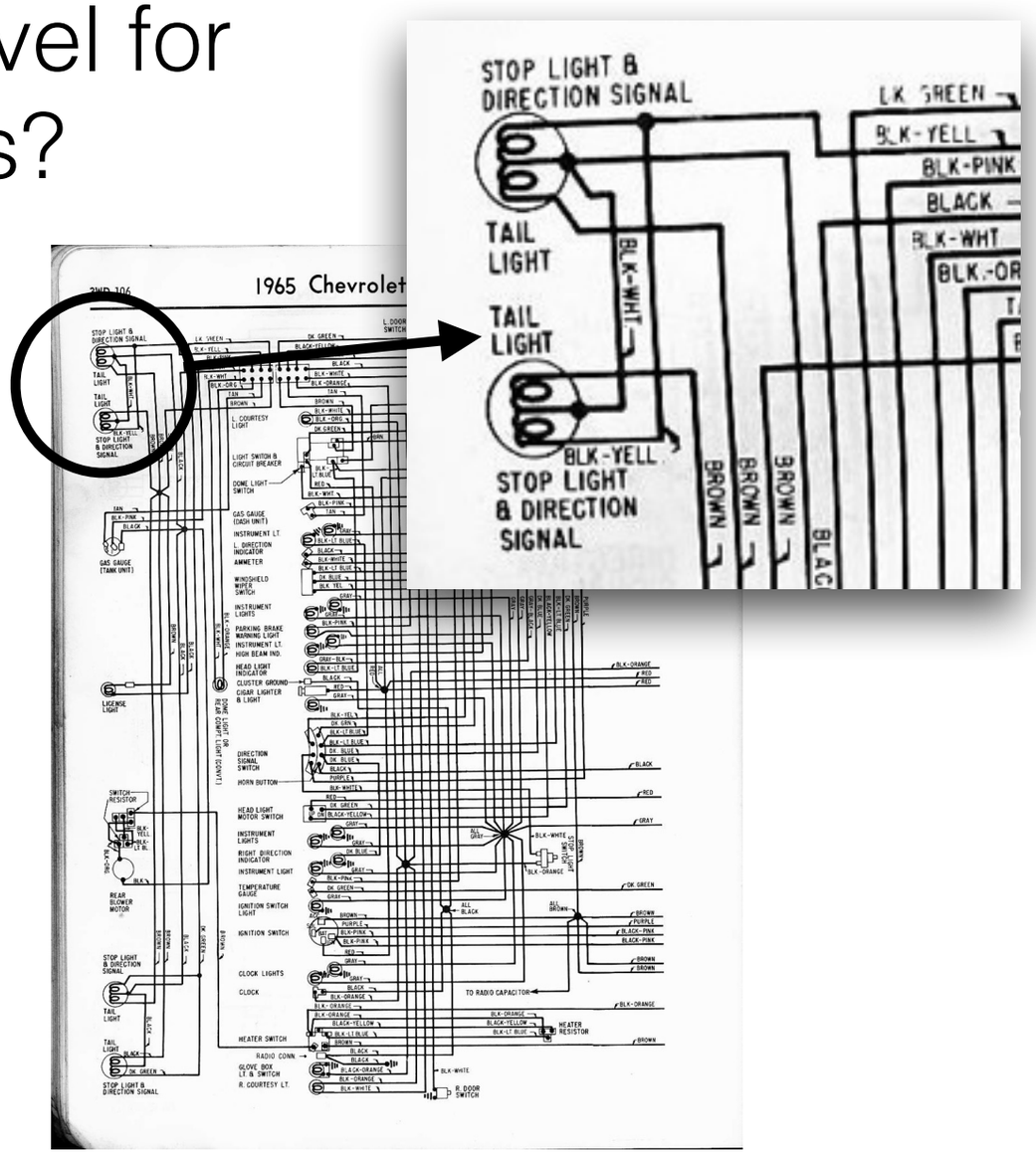
**B**

# Why building Theater Framework?

What is the right abstraction level for repairing the electrical systems?

A

VS




B

# Why building Theater Framework?

Many tasks in the existing iOS applications require the developer to get into the “wiring” of the SDK.

That creates bugs and frustration.

# Why building Theater Framework?

Examples of abstraction problems in iOS:

- Using the CoreBluetooth API's require the developer to define which dispatch_queue to use.

- If a variable might be accessed from two different threads, it is the developer's responsibility to handle the synchronization code and avoid the race conditions (locks, semaphores, etc).

# Why building Theater Framework?

- This problems have been fixed long time ago.

- It would be awesome if we could implement those solutions into the App development kits to make our life much easier and our Apps much better and expressive.

# The Actor Model

- Mathematical model of concurrent computation.
- Introduced by Carl Hewitt, Peter Bishop and Richard Steiger.
- **Paper**: A Universal Modular ACTOR Formalism: http://worrydream.com/refs/Hewitt-ActorModel.pdf
- This research was sponsored by the MIT Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research.

# The Actor Model

- Actors are like Java or Swift objects but:

1. Communications are asynchronous and Message based. (like people communicate)

2. Storage is share nothing. (An actor cannot access the internals of another actor)

3. Processing is single threaded, an actor can only handle 1 message at a time.

# The Actor Model

- Actor frameworks exist in other programming languages:

    1. Scala & Java have Akka.
    2. Erlang has a built-in library.
    3. .NET has akka.NET
    4. Objective-C has ActorKit

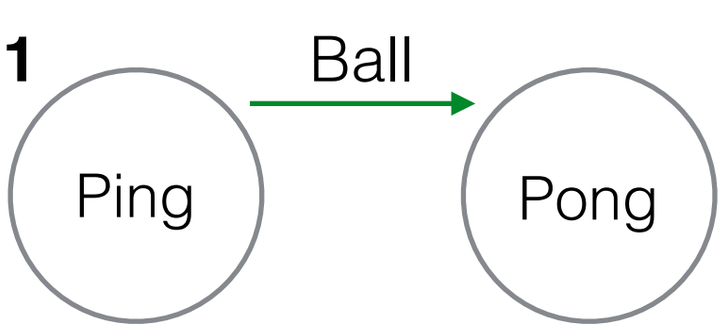
…

# Exercise 1: PingPong

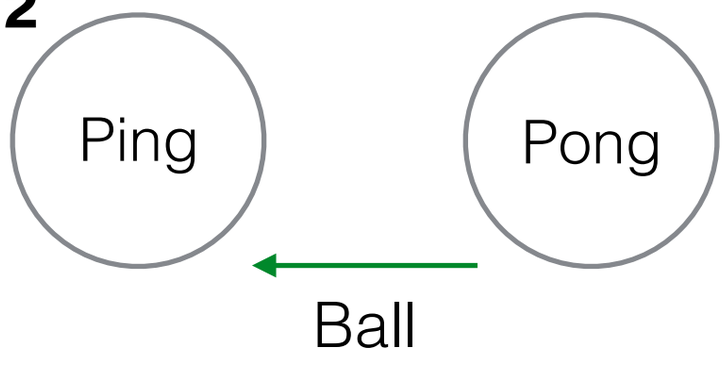
- Create two actors, Ping and Pong that pass the ball back and forth until the end of time:

  **loop {**

  **Step 1**




  **Step 2**



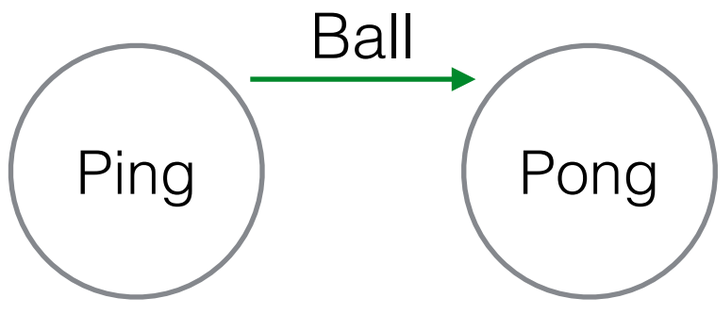

**}**

# Exercise 1: PingPong

- Actors live in actor systems, here's how you define the PingPong system:

```
public class PingPong {

    let system = ActorSystem(name: "pingpong")
    let ping : ActorRef
    let pong : ActorRef

    public init() {
        self.ping = system.actorOf(Ping.self, name: "ping")
        self.pong = system.actorOf(Pong.self, name: "pong")
        kickOffGame()
    }

    func kickOffGame() {
        pong ! Ball(sender: ping)
    }

}
```

=



# Exercise 1: PingPong

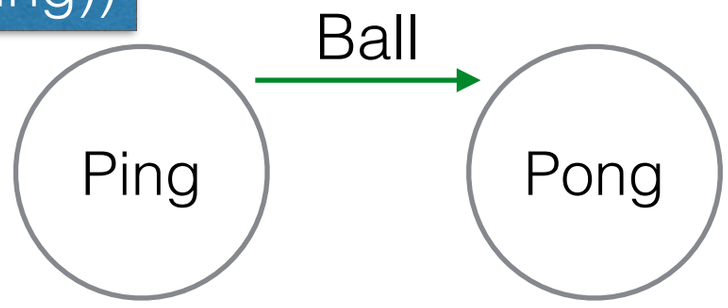
- Actors live in an actor system, here's how you define one:

```
public class PingPong {

    let system = ActorSystem(name: "pingpong")
    let ping : ActorRef
    let pong : ActorRef

    public init() {
        self.ping = system.actorOf(Ping.self, name: "ping")
        self.pong = system.actorOf(Pong.self, name: "pong")
        kickOffGame()
    }

    func kickOffGame() {
        pong ! Ball(sender: ping)
    }

}
```

ActorSystem constructor

Actor creation

this is equal to pong.tell(Ball(sender: ping))



# Exercise 1: PingPong

- Actor and message definition

```
class Ball : Actor.Message {}

class Pong : Actor {

    override func receive(msg: Actor.Message) {
        switch(msg) {
            case is Ball:
             msg.sender ! Ball(sender: this)

            default:
                super.receive(msg)
        }
    }
}
```

# Exercise 1: PingPong

- Actor and message definition

```
class Ball : Actor.Message {}

class Pong : Actor {

    override func receive(msg: Actor.Message) {
        switch(msg) {
            case is Ball:
             msg.sender ! Ball(sender: this)

            default:
                super.receive(msg)
        }
    }
}
```

Messages should subclass Actor.Message

Actors should subclass Actor

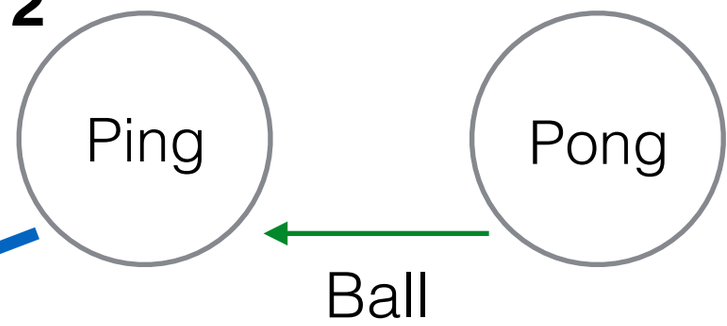
Actors must override this method to receive messages

Use a switch statement to unwrap and react to the message

# Exercise 1: PingPong

- Complete Pong actor implementation

**Step 2**

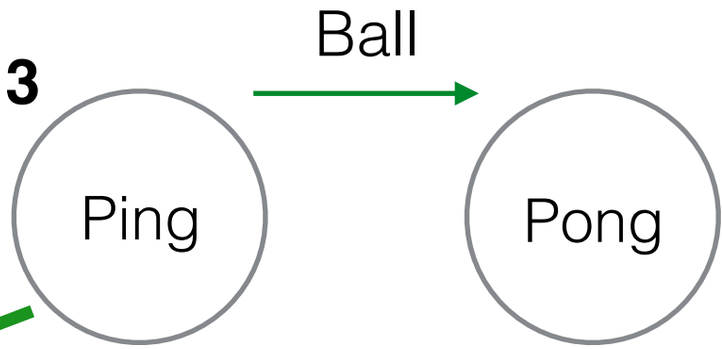



```
class Pong : Actor {

    override func receive(msg: Actor.Message) {
        switch(msg) {
            case is Ball:
                msg.sender ! Ball(sender: this)

            default:
                super.receive(msg)
        }
    }
}
```

# Exercise 1: PingPong

- Complete Pong actor implementation

**Step 3**




```
class Ping : Actor {

    override func receive(msg: Actor.Message) {
        switch(msg) {
            case is Ball:
                msg.sender ! Ball(sender: this)

            default:
                super.receive(msg)
        }
    }
}
```

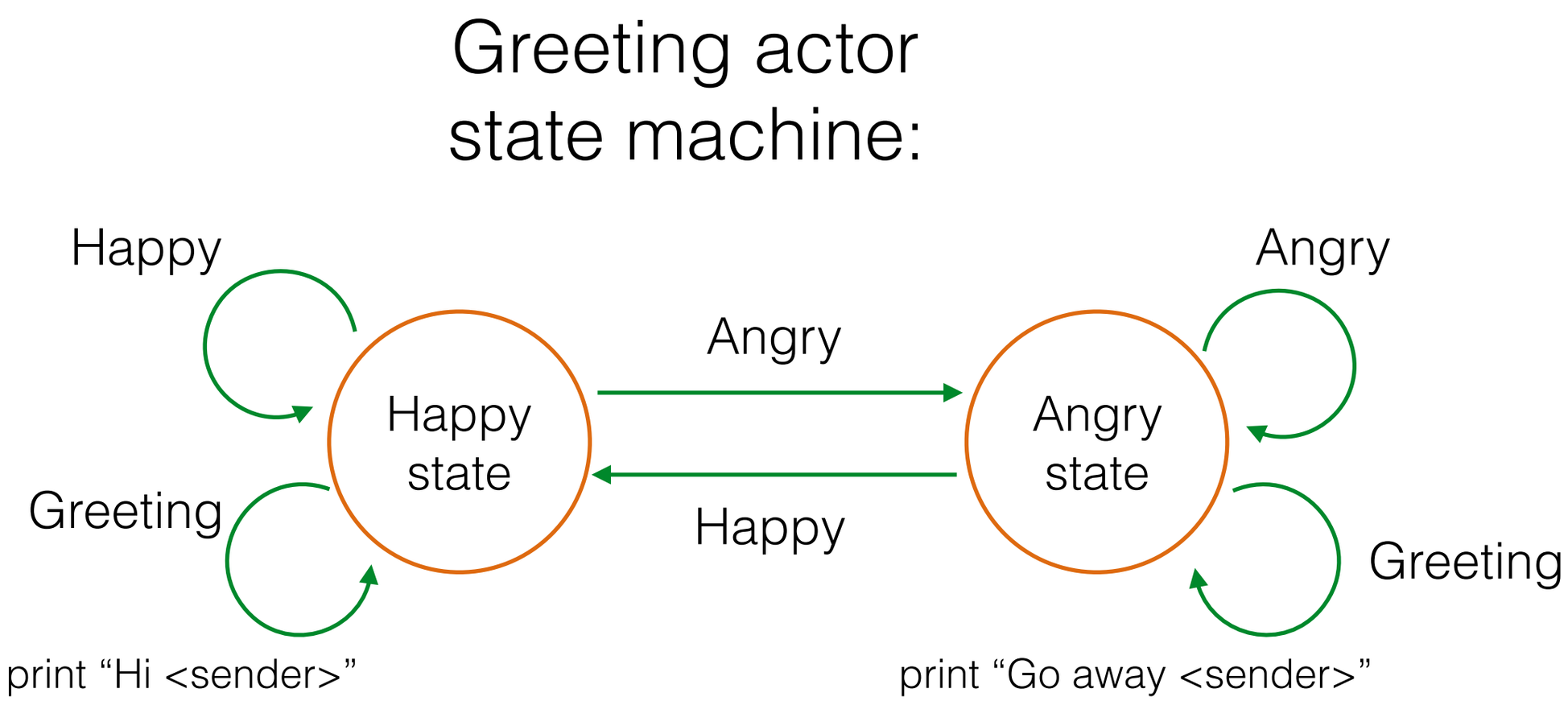
# Exercise 1: PingPong

- Sample output (Added some log statements)

```
Optional(“ping") said <Actors.Ball: 0x7ae24700> to pong
Optional(“pong”) said <Actors.Ball: 0x7aed8720> to ping
Optional("ping") said <Actors.Ball: 0x7af0ee60> to pong
Optional("pong") said <Actors.Ball: 0x7aed8720> to ping
Optional("ping") said <Actors.Ball: 0x7ae23f10> to pong
…
```

# Exercise 2:Greeting actor

- Actors can help us to represent state, consider the following:



# Exercise 2:Greeting actor

```
class GreetingActor: Actor {

override func receive(msg: Actor.Message) {
    return self.happy()(msg)
}

func happy() -> Receive {
   return {(msg : Message) in
      switch(msg) {
       case let g as Greeting:
          print("Hello \(g.sender)”)

       case is Angry:
          self.become("angry",
          state: self.angry())

       default:
          super.receive(msg)
      }
   }
}
```

```
func angry()  -> Receive {
    return {(msg : Message) in
        switch(msg) {
        case let g as Greeting:
            print("Go away \(g.sender)")

        case is Happy:
            self.become("happy",
             state: self.happy())

        default:
            super.receive(msg)
        }
    }
}
```

# Exercise 2:Greeting actor

```
class GreetingActor: Actor {

override func receive(msg: Actor.Message) {
    return self.happy()(msg)
}

func happy() -> Receive {
    return {(msg : Message) in
        switch(msg) {
        case let g as Greeti
            print("Hello \(g.sender)")

        case is Angry:
            self.become("angry",
            state: self.angry())

        default:
            super.receive(msg)
        }
    }
}
```

```
func angry()  -> Receive {
    return {(msg : Message) in
        switch(msg) {
        case let g as Greeting:
            print("Go away \(g.sender)")

        case is Happy:
            self.become("happy",
             state: self.happy())

        default:
            super.receive(msg)
        }
    }
}
```

State transition with “self.become”

# Exercise 3: Turnstile

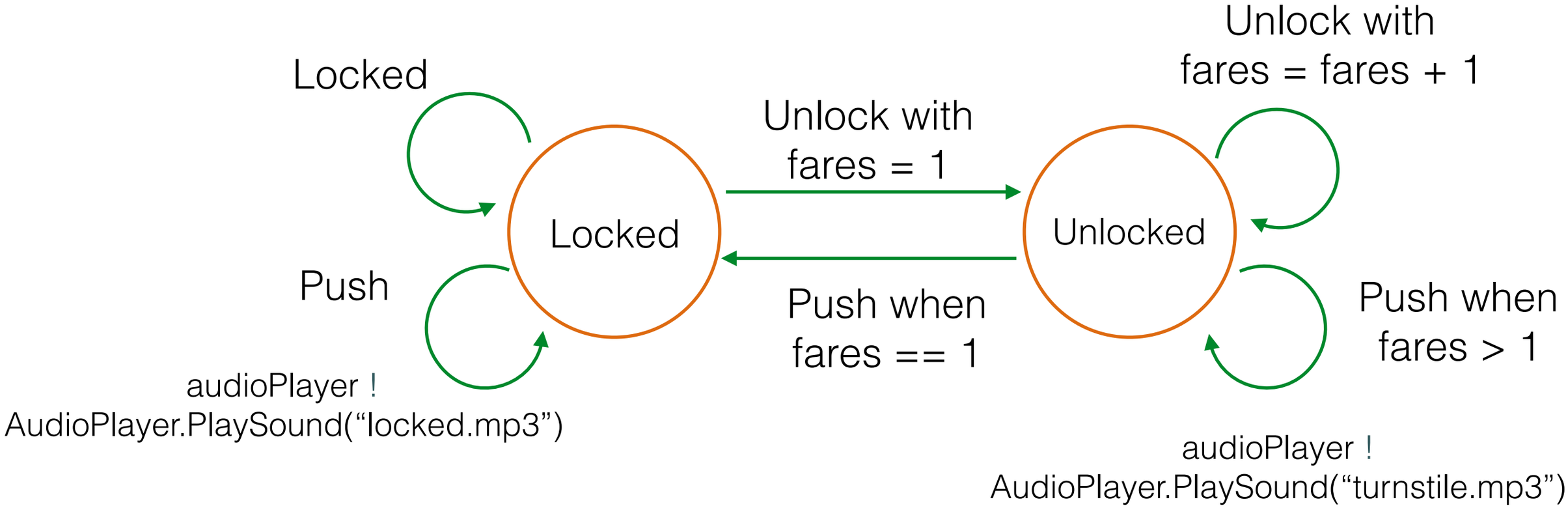
- Let's create a simple turnstile simulator using Theater.

# Exercise 3: Turnstile

```
SyncTurnstileViewController : UIViewController {
  let coinModule : ActorRef =
  AppActorSystem.shared.actorOf(CoinModule.self)

  let gate : ActorRef =
  AppActorSystem.shared.actorOf(Gate.self)

  @IBOutlet weak var status: UILabel!

  override func viewDidLoad() {
      super.viewDidLoad()
      gate ! SetViewCtrl(ctrl:self)
      coinModule ! SetViewCtrl(ctrl:self)
  }

  @IBAction func onPush(sender: UIButton) {
      gate ! Gate.Push(sender : nil)
  }

  @IBAction func onInsertCoin(sender: UIButton) {
      coinModule ! CoinModule.InsertCoin(sender : nil)
  }
}
```

# Exercise 3: Turnstile

## Gate State Machine



# Conclusions

- Presented a framework (Theater) that helps to write async and responsive iOS applications using Actors.

- Discussed how by raising the abstraction level it is easier to build correct, concurrent and clean code.

- Presented a different way to code state machines in iOS Apps.

# How to get started?

- Go to http://www.theaterframework.com

- Follow the tutorial using CocoaPods.

- Read the docs @ http://cocoadocs.org/docsets/Theater

# How to get involved?

- Go to http://www.theaterframework.com

- Grab some of the open issues or create some, fork the code, do your thing and create a pull request.

- Theater is distributed under the Apache 2 License.

- Follow us on Twitter @TheaterFwk