Let’s play a little guessing game

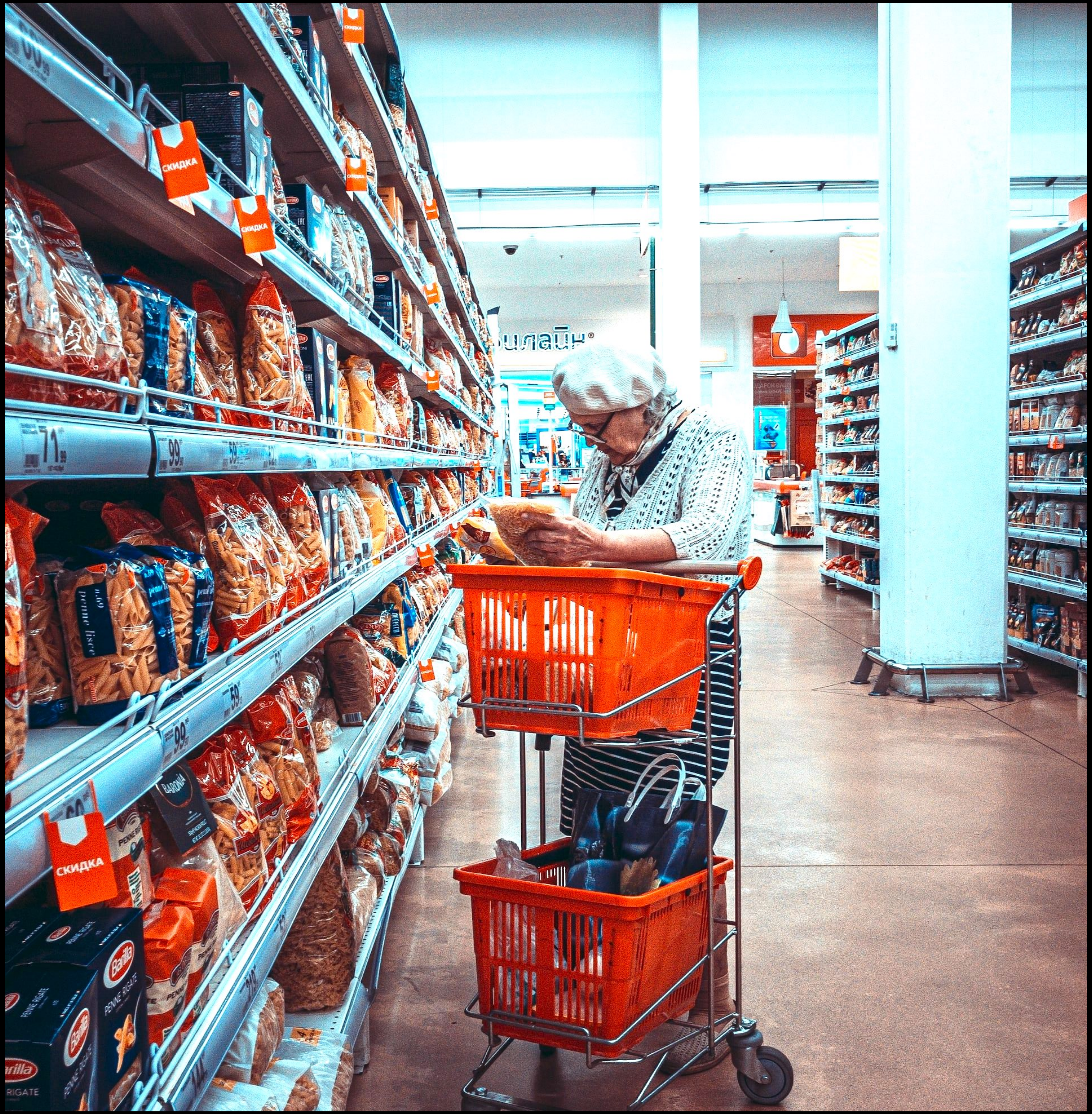
What’s this?

An old lady?

A shopping cart?

Pastas?

A supermarket?

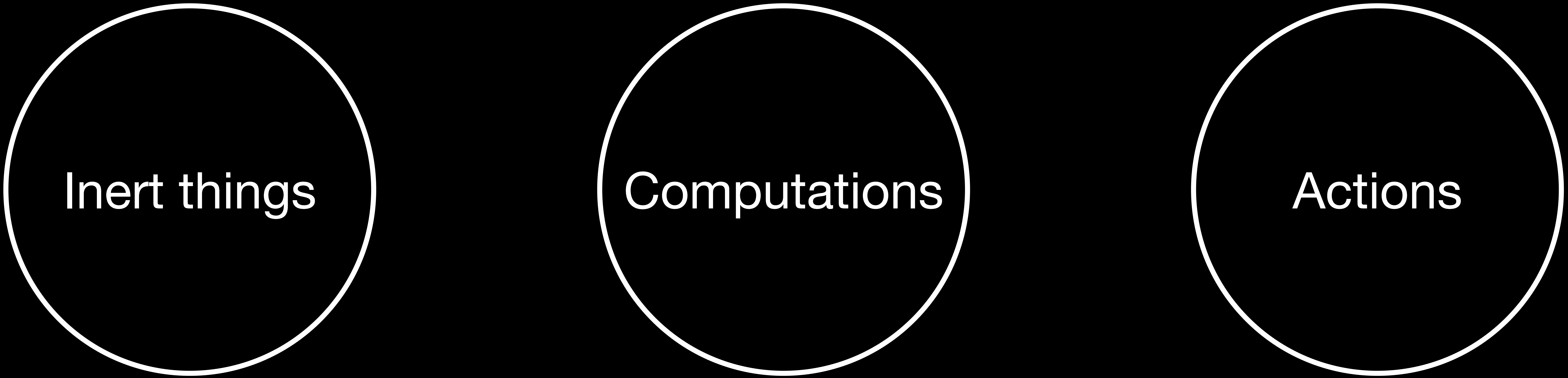


An iOS developer at the end of the day?

It’s Functional Programming

Functional Programming divides systems into 3 categories

Functional Programming divides systems into 3 categories



Inert things

Computations

Actions

Inert things

A shopping cart

Pastas

Raw material that can do nothing by itself

Computing the total price

Computations

Adding an item to the cart

Always have the same result no matter how many times we do it

Paying the bill

Finding a parking spot

Actions

The outcome depends on when and how many times you do it

Functional Programming

♥

Inert things & computations

(Because they are safe to use, predictable and highly testable)

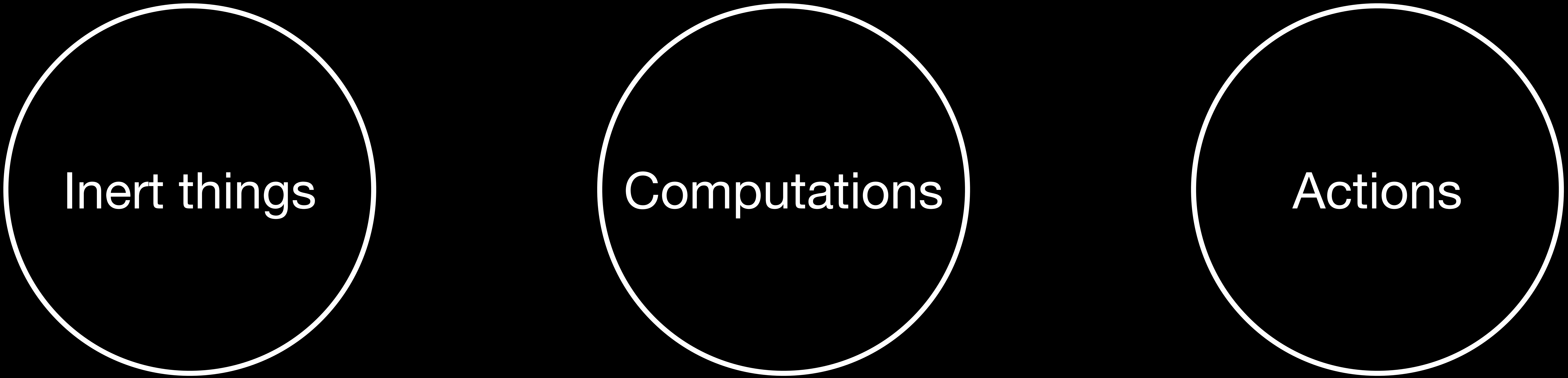
Functional Programming

△

Actions

(Because they are more unpredictable and we will have to manage them)

Functional Programming divides systems into 3 categories



Inert things

Computations

Actions

State machines are closely related to Functional Programming

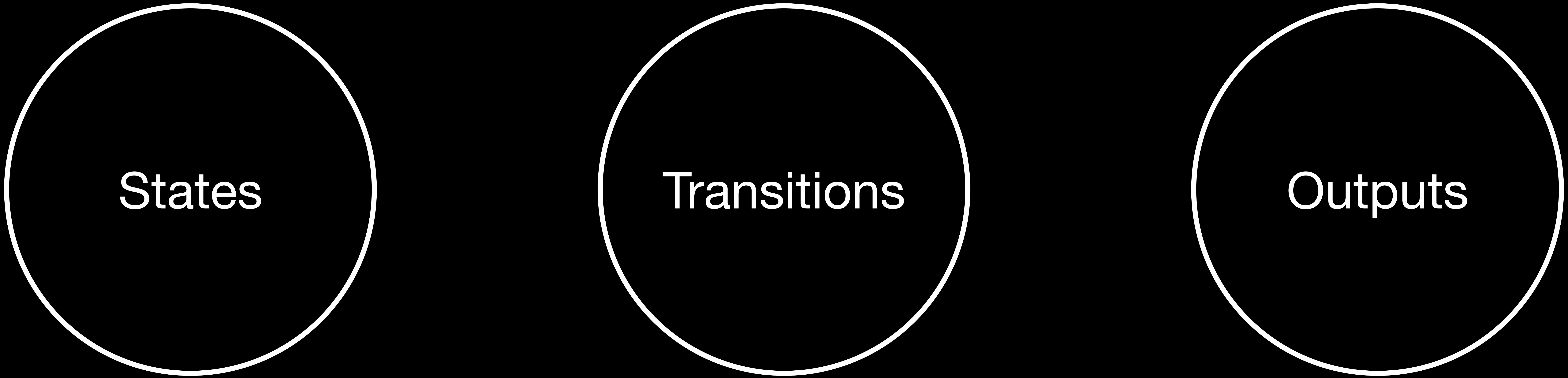
Thibault Wittemberg

FrenchKit 2022

Swift concurrency and state machines

The path to modern and reliable features

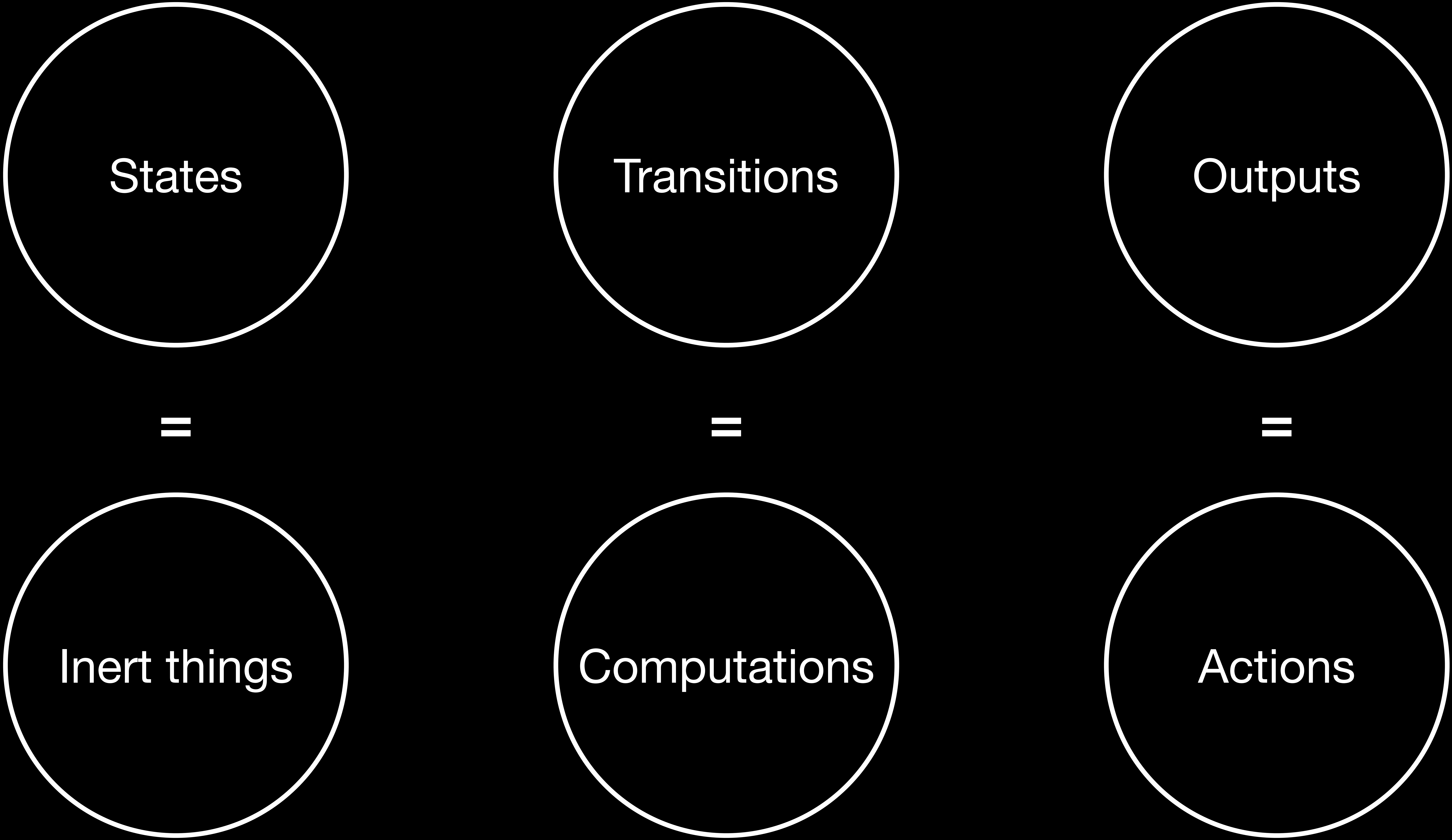
State machines also divide systems into 3 categories



States

Transitions

Outputs



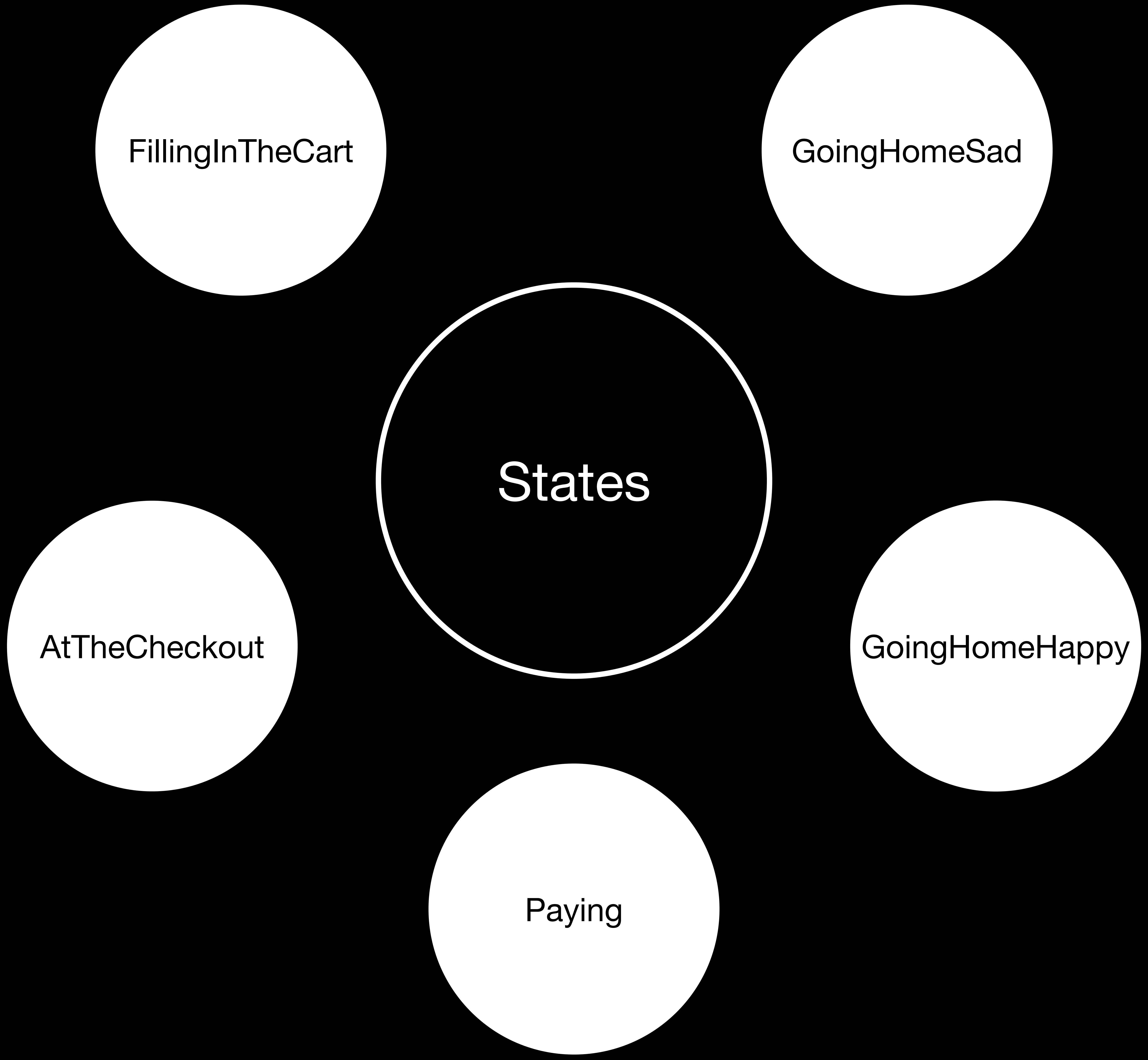
States = Inert things

Transitions = Computations

Outputs = Actions

States

Finite set of mutually exclusive values



FillingInTheCart

GoingHomeSad

States

AtTheCheckout

GoingHomeHappy

Paying

States

Immutable data

```
struct Item {
  let price: Price
  let name: String
}

struct ShoppingCart{
  var items: [Item]
}
```

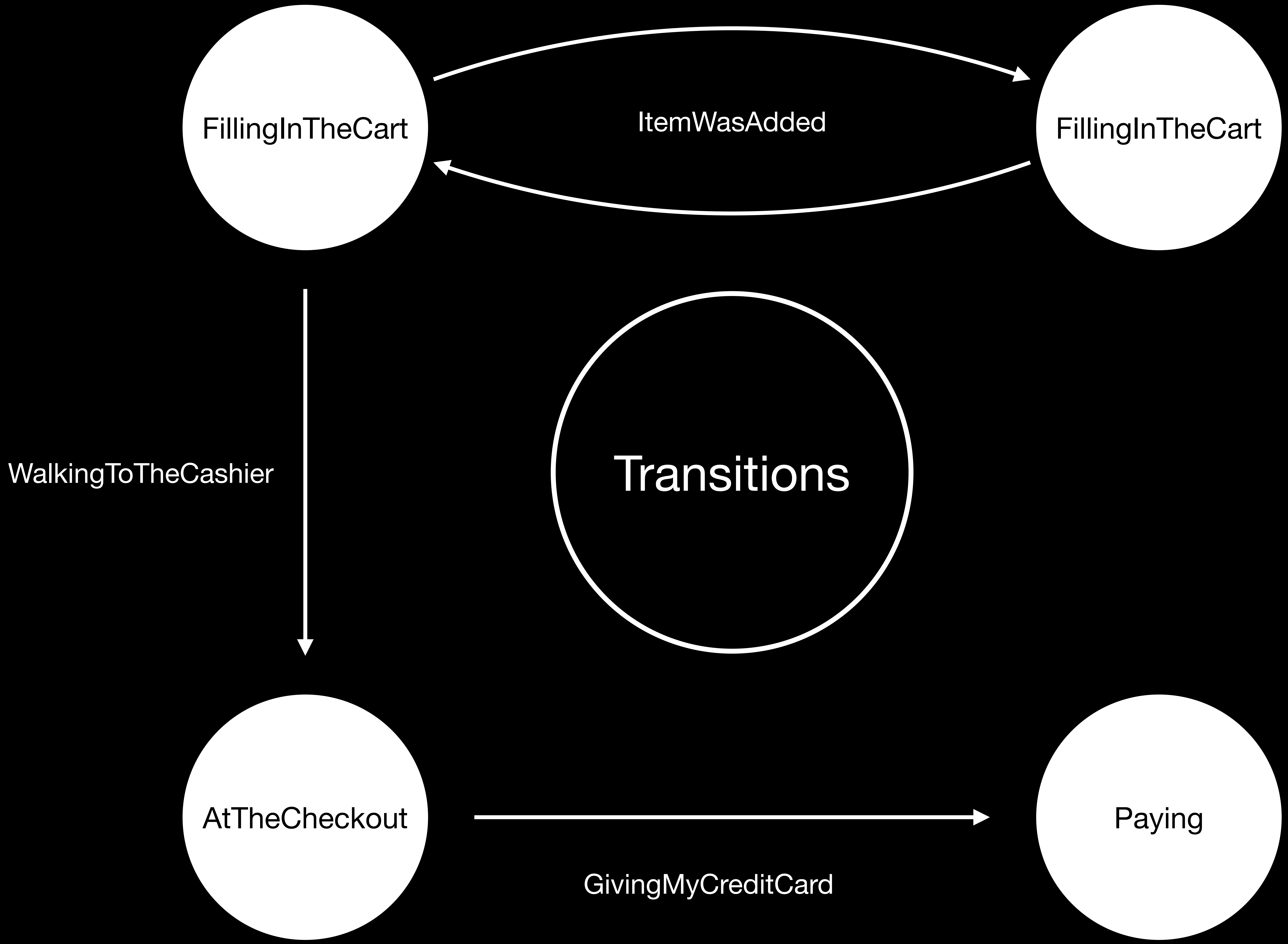
```
enum SupermarketState {
  case fillingInTheCart(ShoppingCart)
  case atTheCheckout(ShoppingCart)
  case paying(ShoppingCart, CreditCard, Price)
  case goingHomeHappy(ShoppingCart)
  case goingHomeSad(ShoppingCart)
}
```

Transitions

Pure functions* that drive the passage from one state to another

*Pure functions are side effect free. They cannot access a shared state

Transitions



FillingInTheCart

ItemWasAdded

FillingInTheCart

WalkingToTheCashier

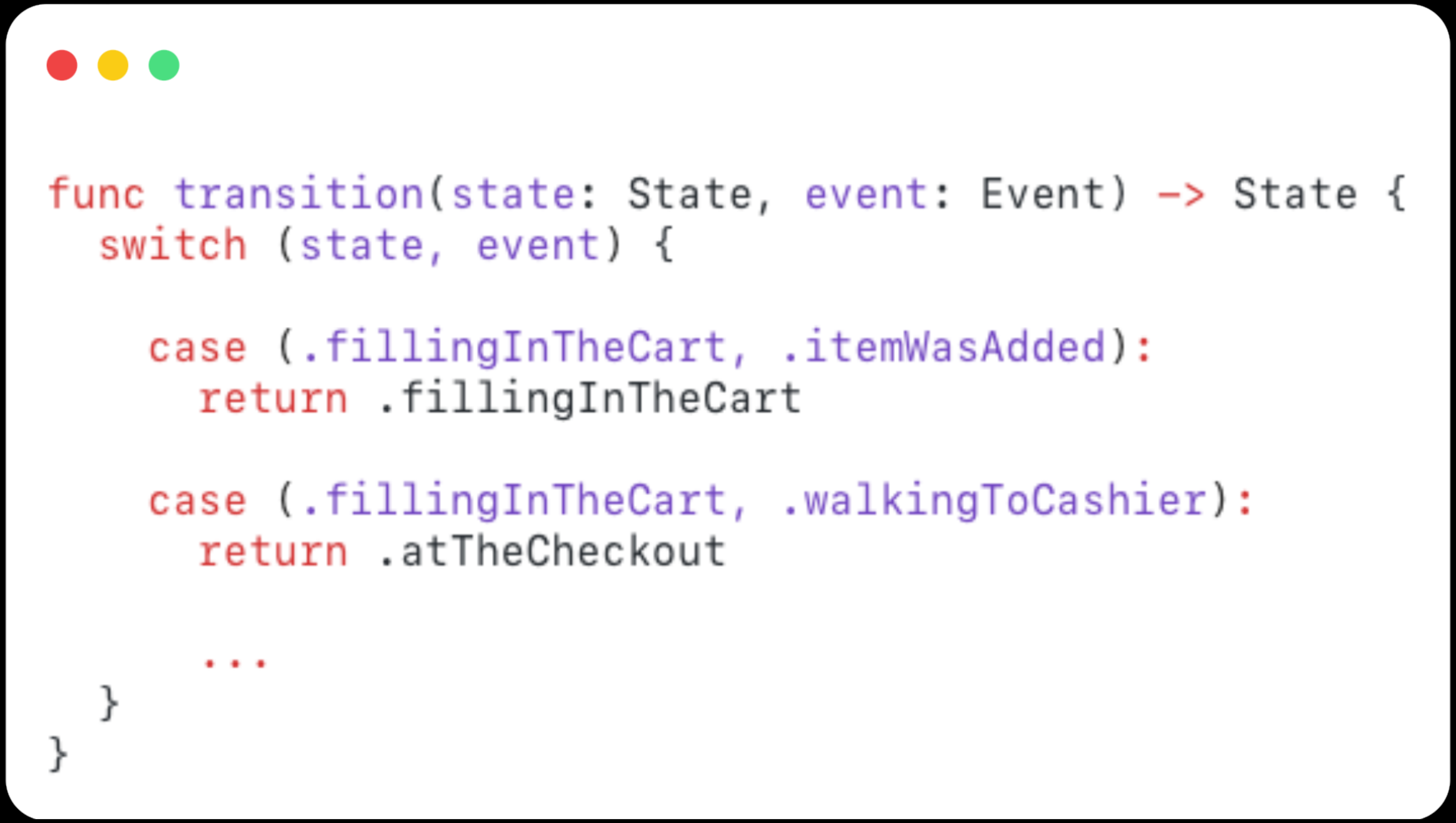
AtTheCheckout

GivingMyCreditCard

Paying

Transitions

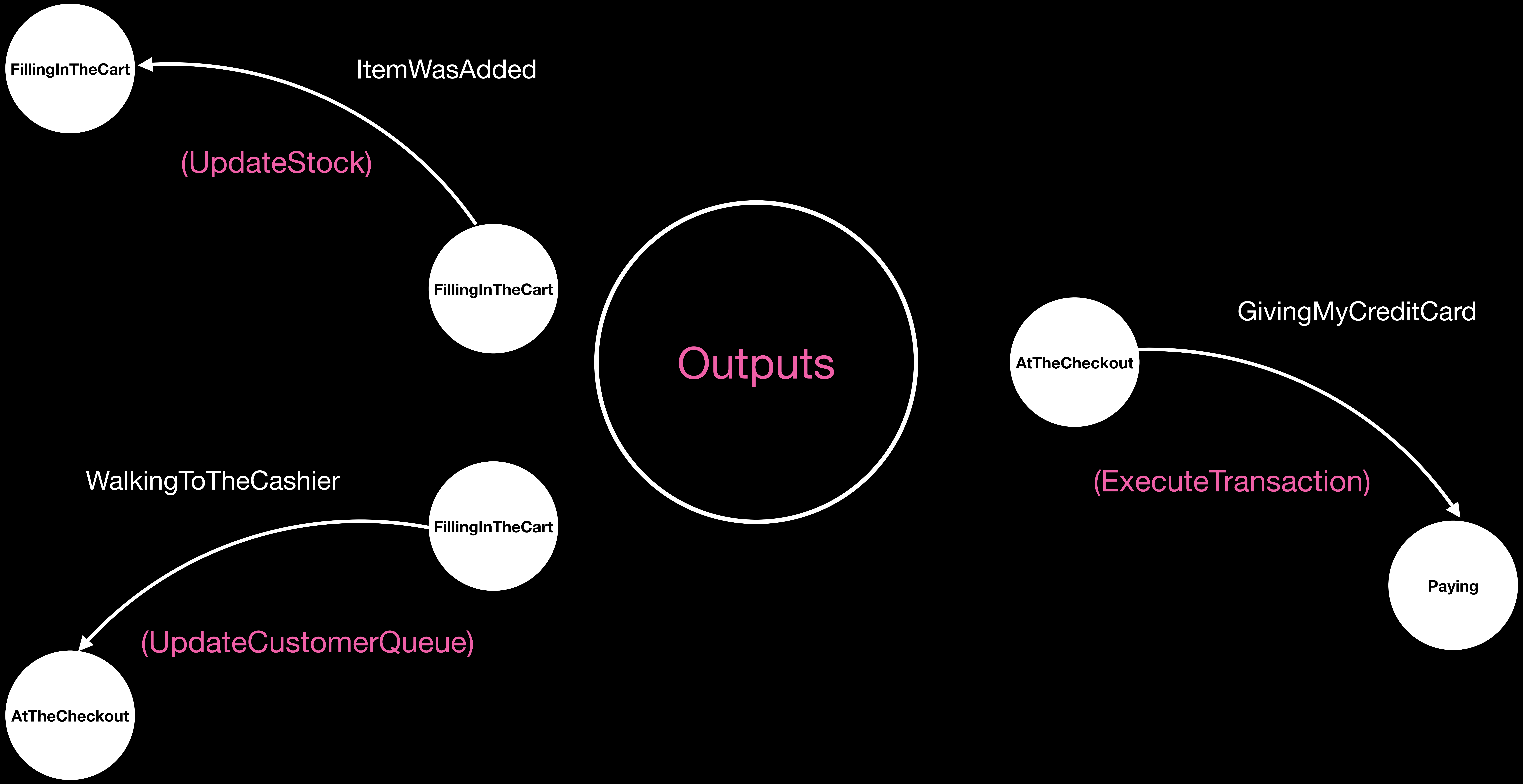
Pure functions



```swift
func transition(state: State, event: Event) -> State {
  switch (state, event) {

    case (.fillingInTheCart, .itemWasAdded):
      return .fillingInTheCart

    case (.fillingInTheCart, .walkingToCashier):
      return .atTheCheckout

      ...
  }
}
```

Outputs

Side effects that depend on the current state and an event



Outputs

FillingInTheCart

ItemWasAdded

(UpdateStock)

FillingInTheCart

WalkingToTheCashier

FillingInTheCart

(UpdateCustomerQueue)

AtTheCheckout

AtTheCheckout

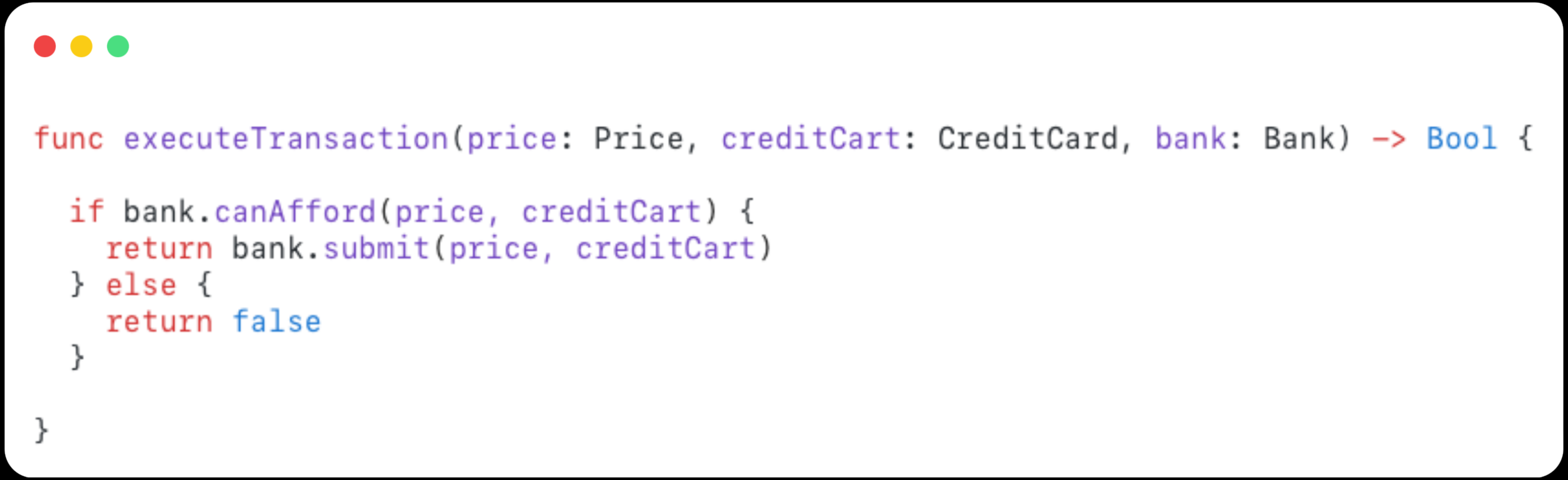
GivingMyCreditCard

(ExecuteTransaction)

Paying

Outputs

Side Effects



```
func executeTransaction(price: Price, creditCart: CreditCard, bank: Bank) -> Bool {

  if bank.canAfford(price, creditCart) {
    return bank.submit(price, creditCart)
  } else {
    return false
  }

}
```

Why state machines and FP?

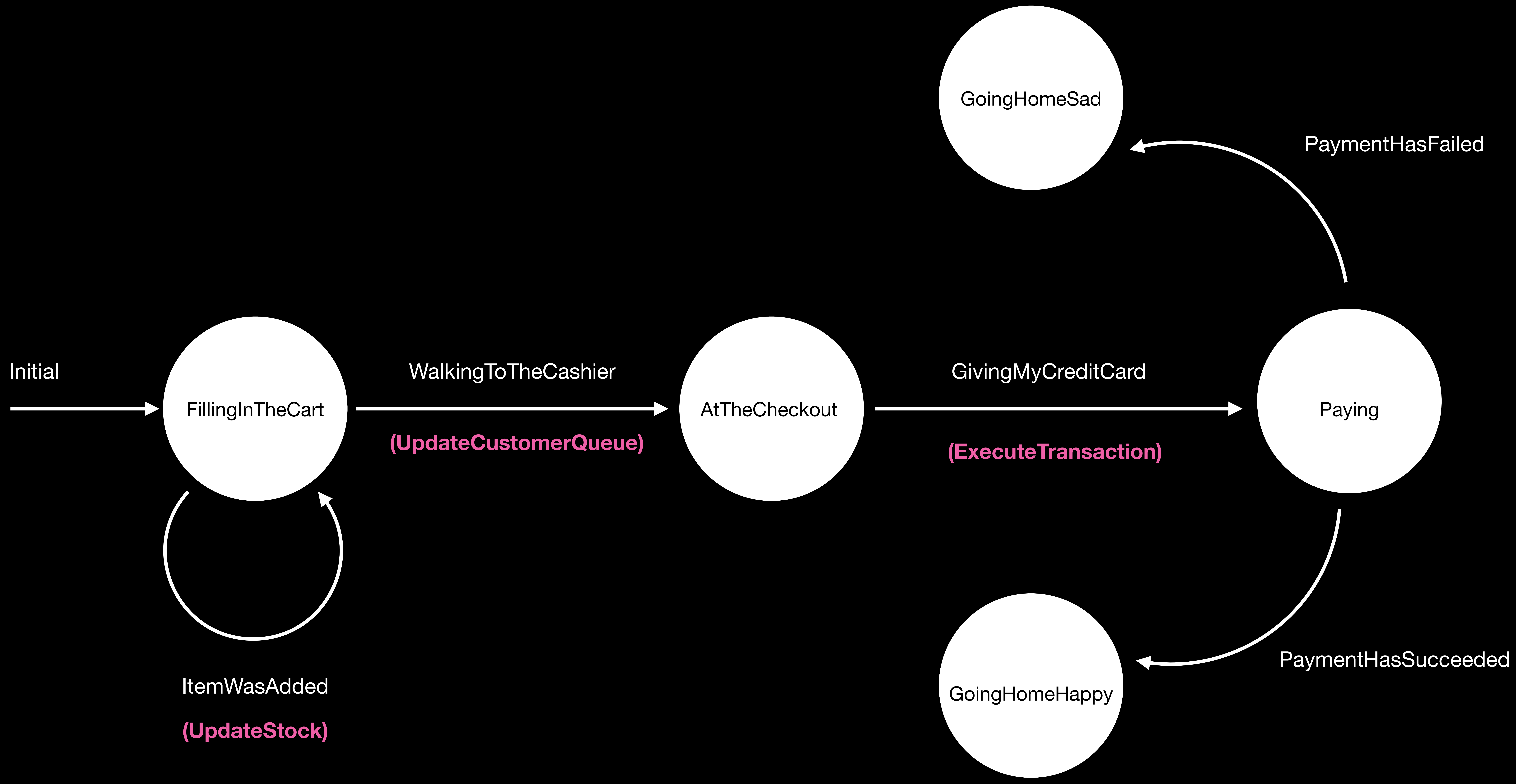
Applications are about state whether you want it or not, let’s make it EXPLICIT

Help increase the code coverage by leveraging pure functions

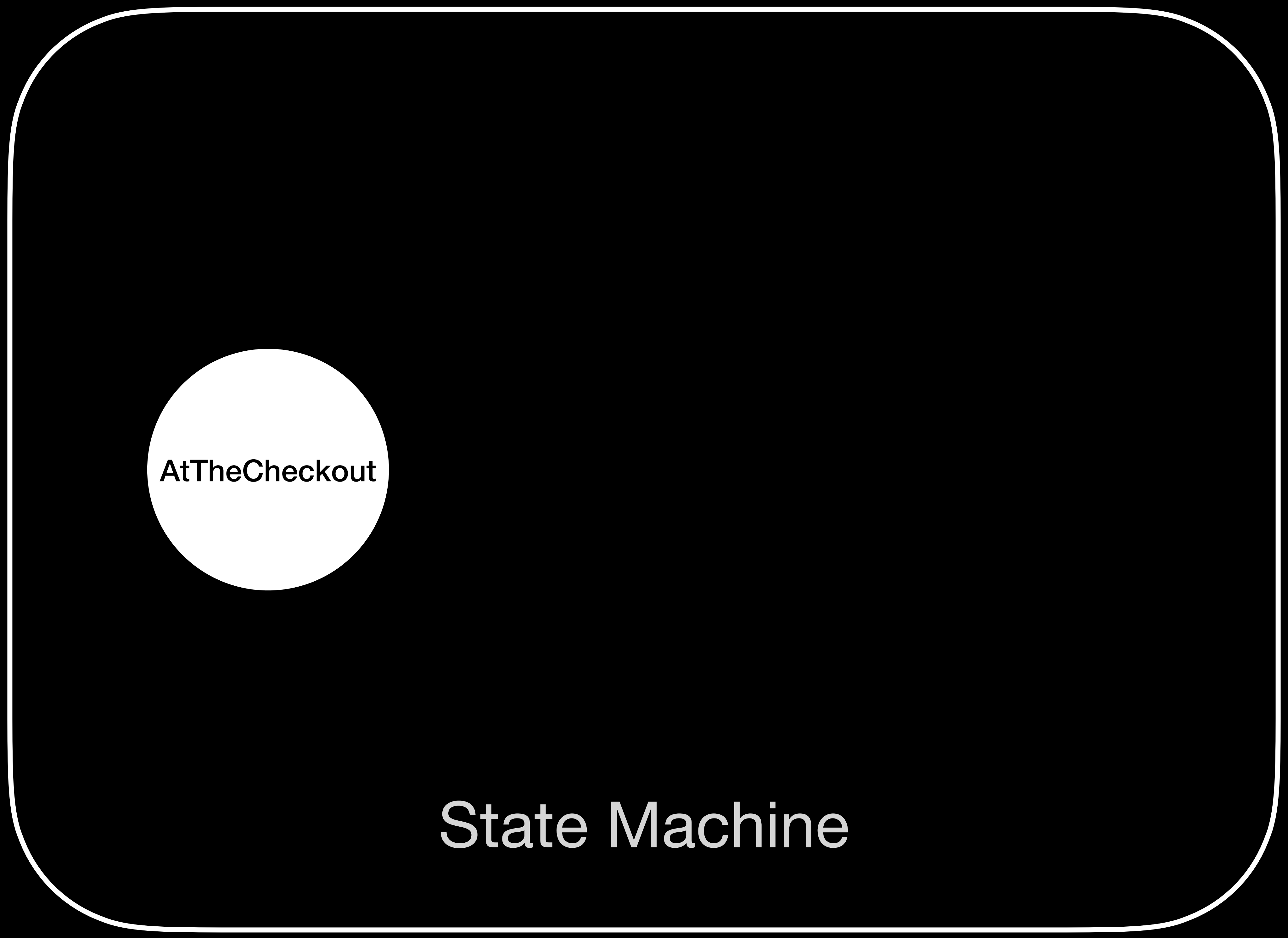
Unlock collaboration across teams around a diagram and eventually a DSL

Document our projects

Help in the paradigm V = f(S) of unidirectional data flow architectures

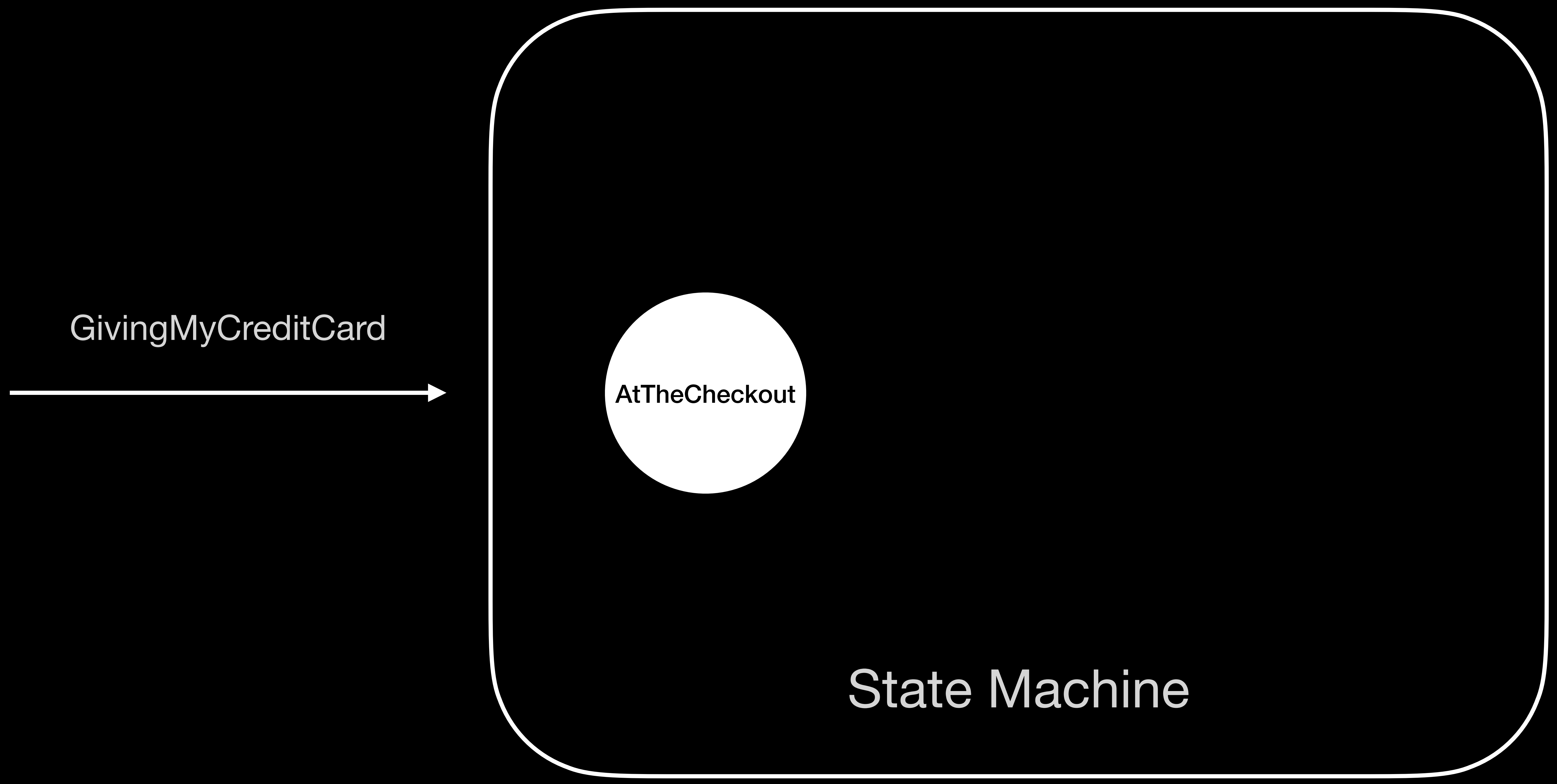

GoingHomeSad
PaymentHasFailed
Initial
FillingInTheCart
WalkingToTheCashier
(UpdateCustomerQueue)
AtTheCheckout
GivingMyCreditCard
(ExecuteTransaction)
Paying
ItemWasAdded
(UpdateStock)
GoingHomeHappy
PaymentHasSucceeded

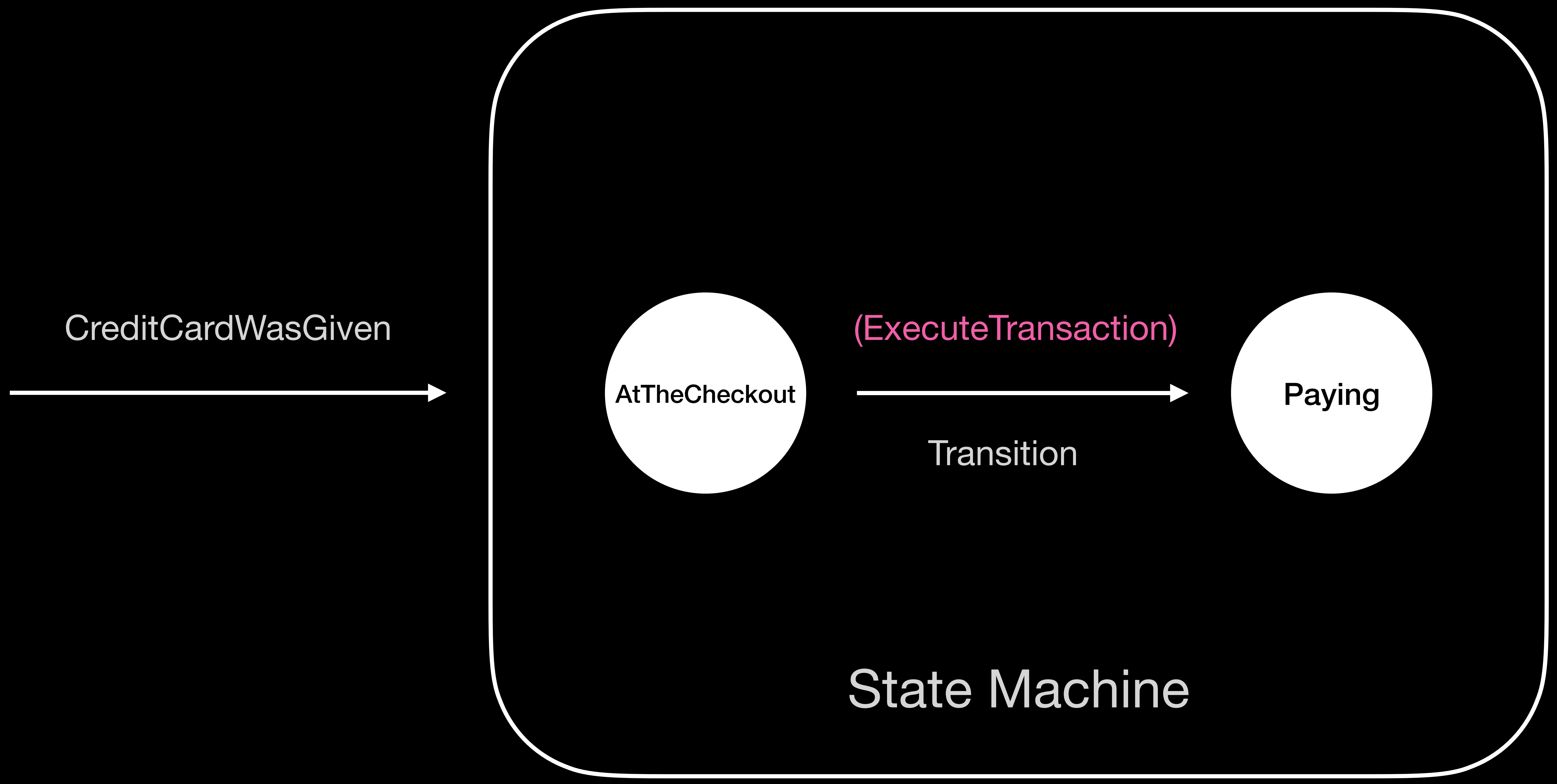
The internal behaviour of our state machine when paying

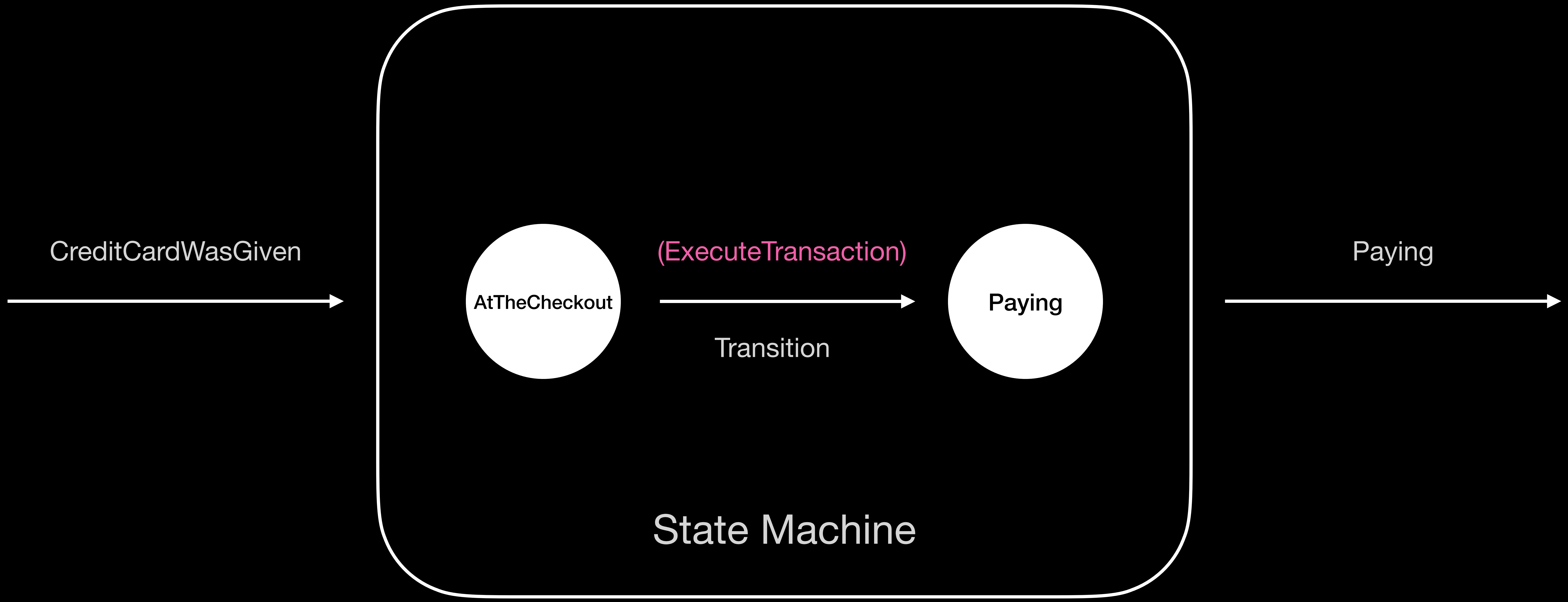


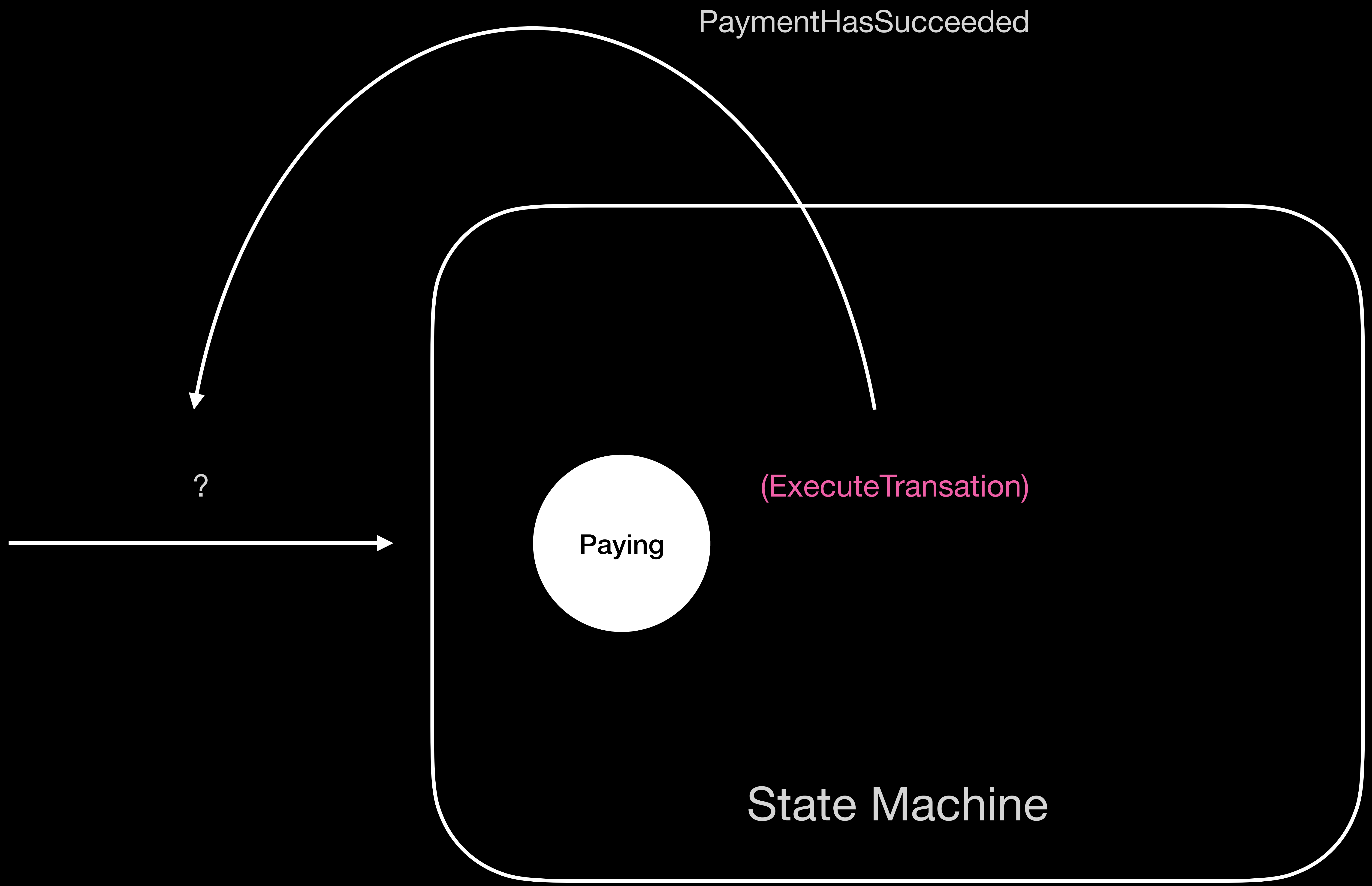
AtTheCheckout

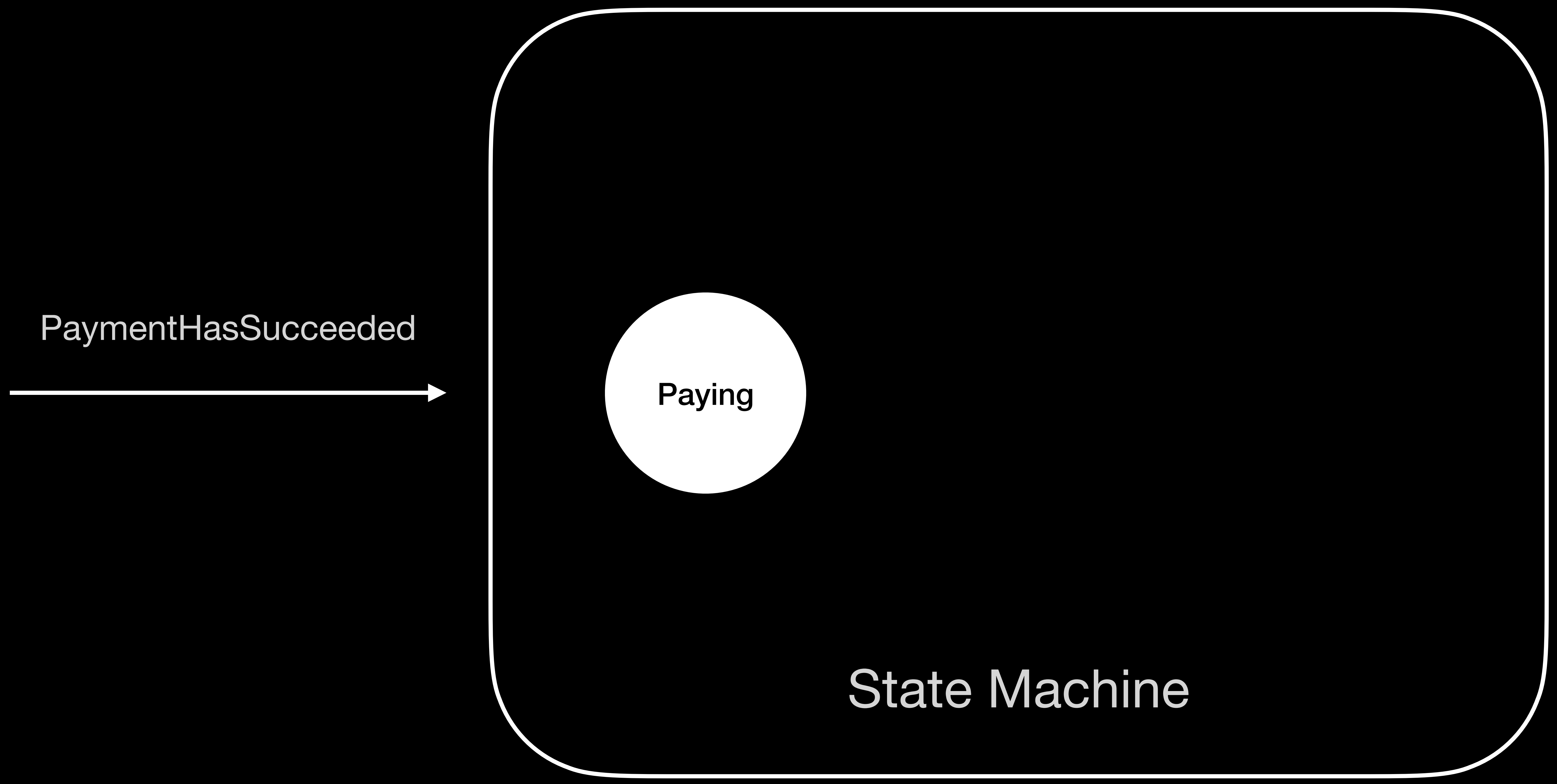
State Machine

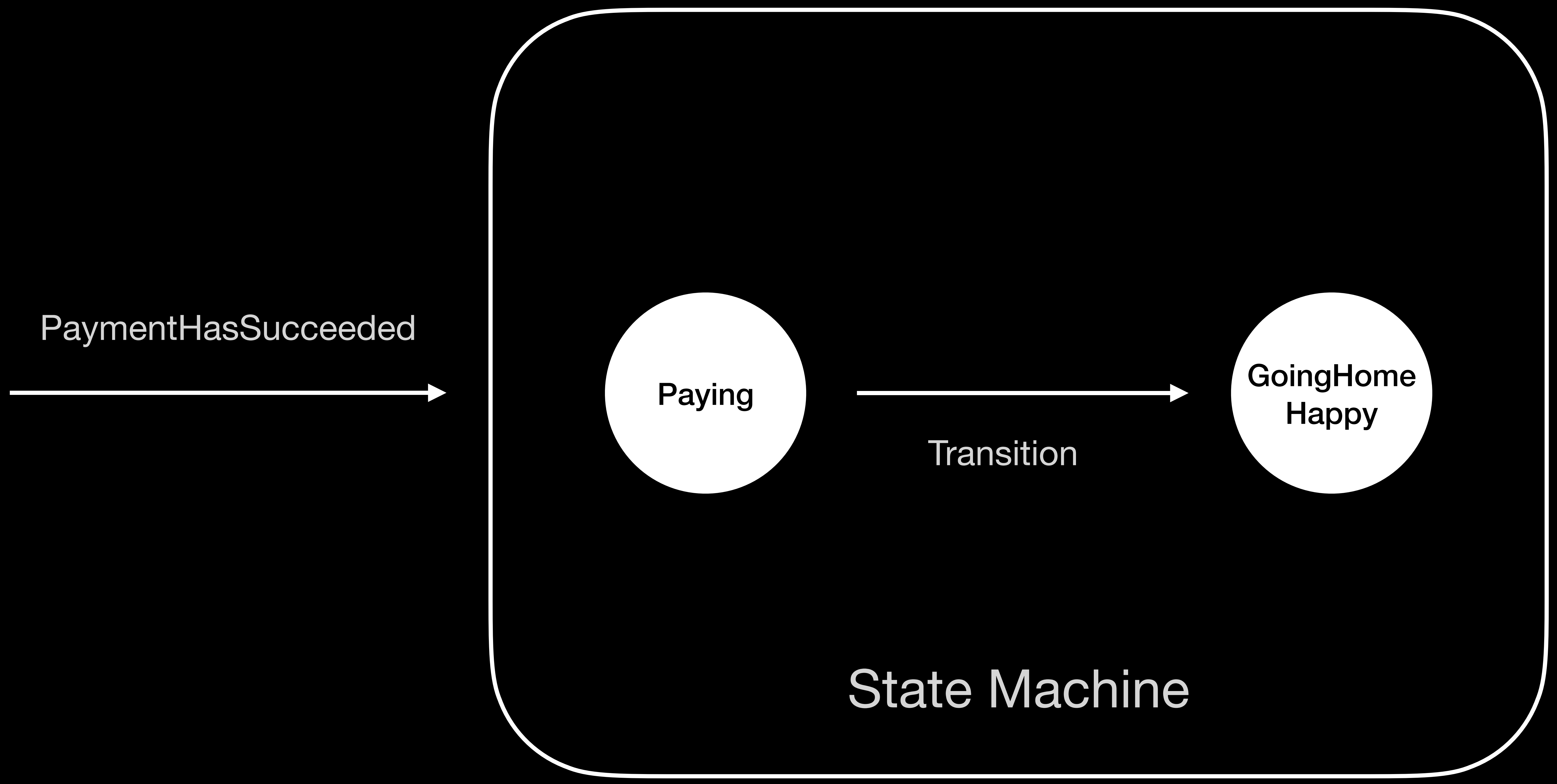

GivingMyCreditCard
AtTheCheckout
State Machine

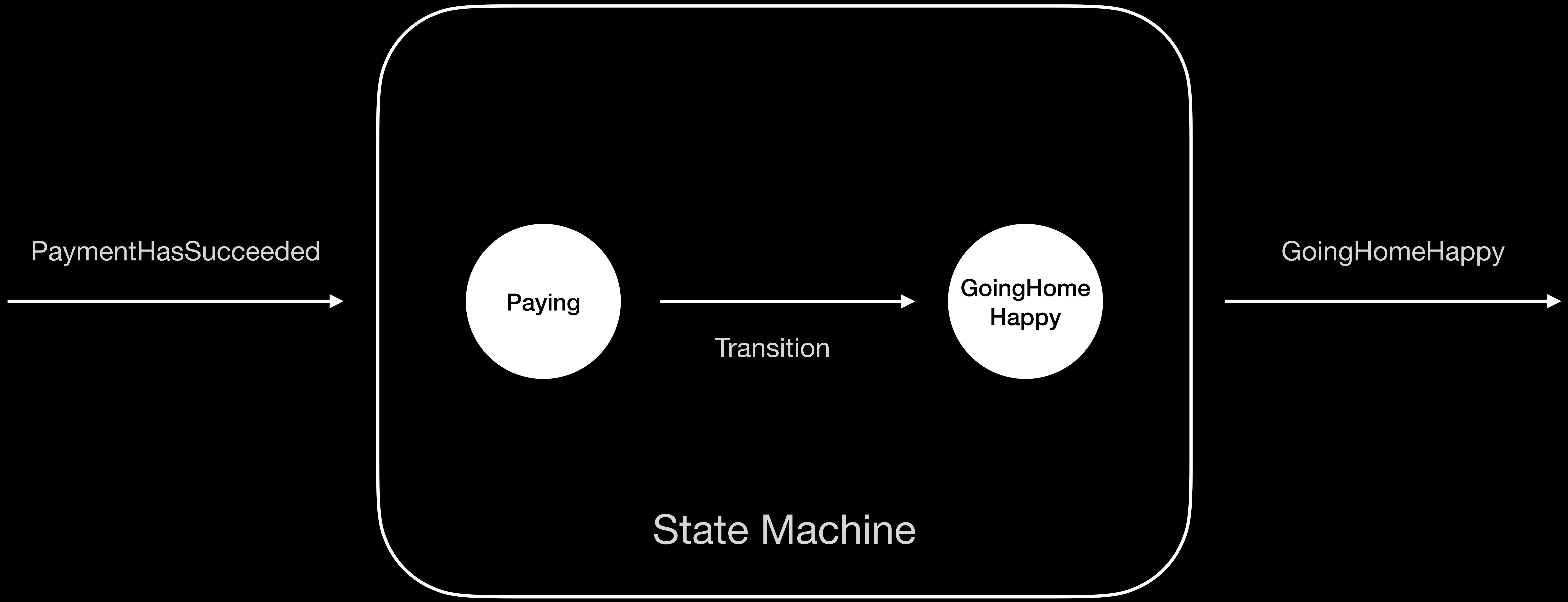

CreditCardWasGiven
AtTheCheckout
(ExecuteTransaction)
Transition
Paying
State Machine

CreditCardWasGiven
AtTheCheckout
(ExecuteTransaction)
Transition
Paying
Paying
State Machine


PaymentHasSucceeded
?
Paying
(ExecuteTransation)
State Machine


PaymentHasSucceeded
Paying
State Machine


PaymentHasSucceeded
Paying
Transition
GoingHome
Happy
State Machine


PaymentHasSucceeded

Paying

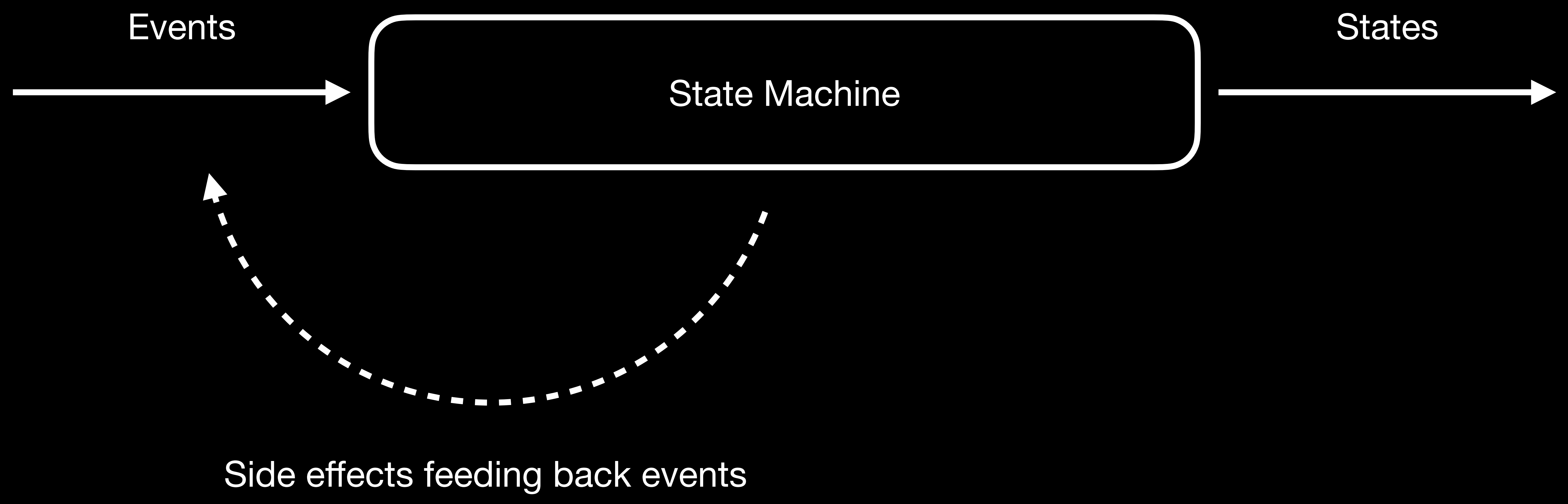
Transition

GoingHome Happy

GoingHomeHappy

State Machine

We can see a state machine as a stream of states



Events

State Machine

States

Side effects feeding back events

Transitions cannot happen concurrently to guarantee the determinism of the state machine

Outputs on the other hand are completely asynchronous

That being said, the state machine as a whole cannot block its callers (could be a UI)

Leveraging Swift concurrency

(Won’t be a deep dive)

Structured

```
let state1 = await transitions(state0, event0)
let state2 = await transitions(state1, event1)
```

A transition might take time to execute (if heavy computations).

The caller thread is free to do something else in the meantime, the result is deferred to a point in future.

Cancellation is collaborative, if the root task is cancelled, so will be the transitions.

(We can use Task.isCancelled to break a for loop for instance)

Unstructured

```
let task = Task {
   let event = await sideEffect()
   stateMachine.send(event)
}

// task.cancel() -> if needed
```

The task execution is scheduled by the system (inherits parent Actor executor).
The collaborative cancellation doesn’t apply, we must handle it by ourselves.

Values over time

```
struct StateMachine: AsyncSequence {
    func next() async -> State? {
        // apply transition
        // return the new state
    }
}
```

```
for await state in stateMachine {
    // publish the state
}
```

A state machine is a sequence of states produced asynchronously.
We will leverage **AsyncSequence** to iterate over these states.

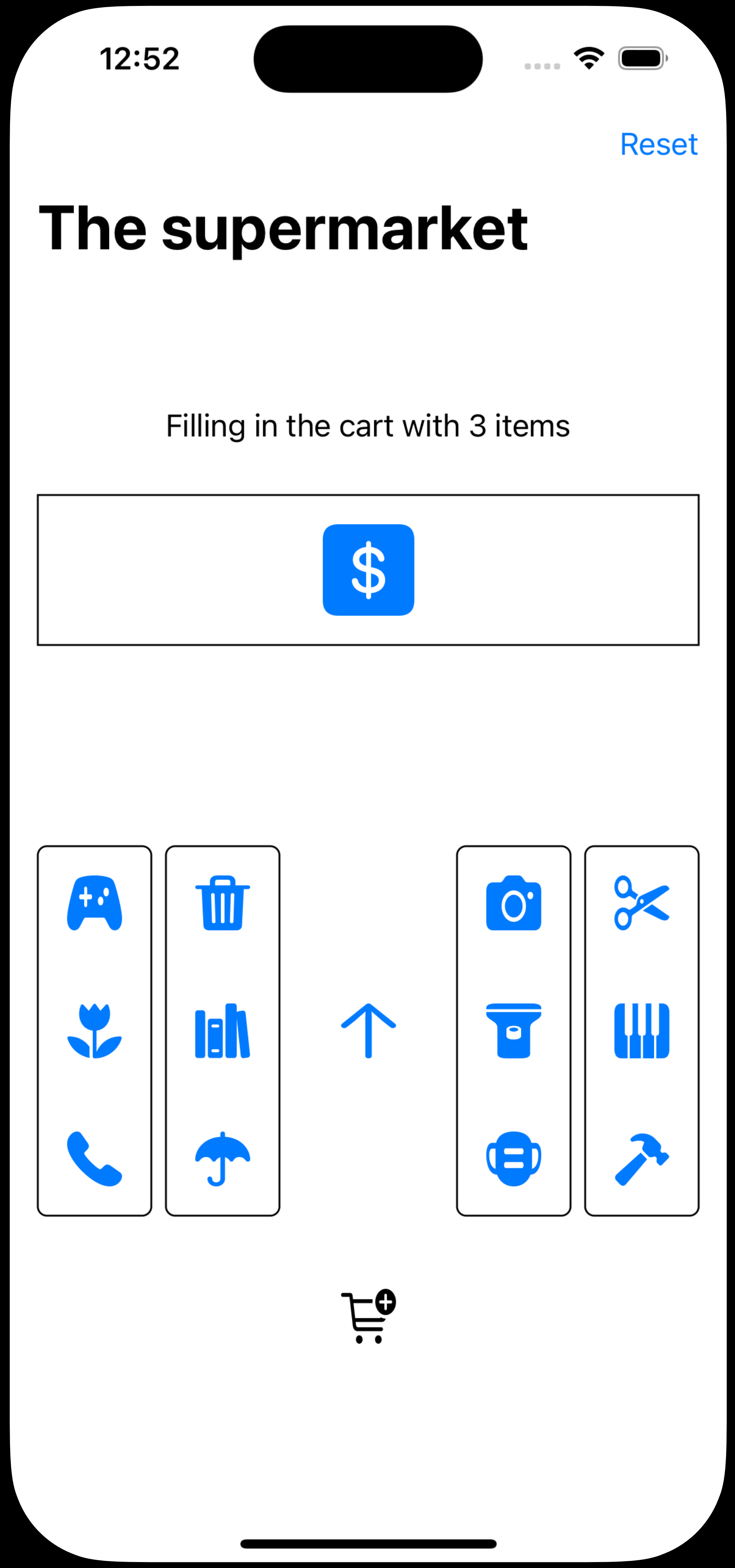
Thibault Wittemberg

FrenchKit 2022

Swift concurrency and state machines

Hands on

The goal: to create a generic state machine engine and use it to model the supermarket use case in a SwiftUI application



The supermarket

Filling in the cart with 3 items

Clone the repo

https://github.com/sideeffect-io/FrenchKit2022_HandsOn https://github.com/sideeffect-io/FrenchKit2022_HandsOn

There’s a README file at the root of the project, just follow the instructions 😄

Credits

Photo by Shashank Verma on Unsplash

Photo by Radek Homola on Unsplash

Photo by Vlad Frolov on Unsplash

Photo by Sven Mieke on Unsplash

Photo by Patrick Tomasso on Unsplash

Photo by Alfred Kenneally on Unsplash

Photo by Studio Blackthorns on Unsplash

Photo by Clay Banks on Unsplash

Photo by Evergreens and Dandelions on Unsplash

Photo by Michael Fousert on Unsplash

Photo by David Veksler on Unsplash

Grokking Simplicity by Eric Normand

Photo by Egor Myznik on Unsplash

Thibault Wittemberg
Freelance @SideEffect

Twitter:	@nakodark
GitHub:	https://github.com/sideeffect-io
	https://github.com/twittemb
LinkedIn:	https://www.linkedin.com/in/twittemb/
Mail:	thibault@sideeffect.io