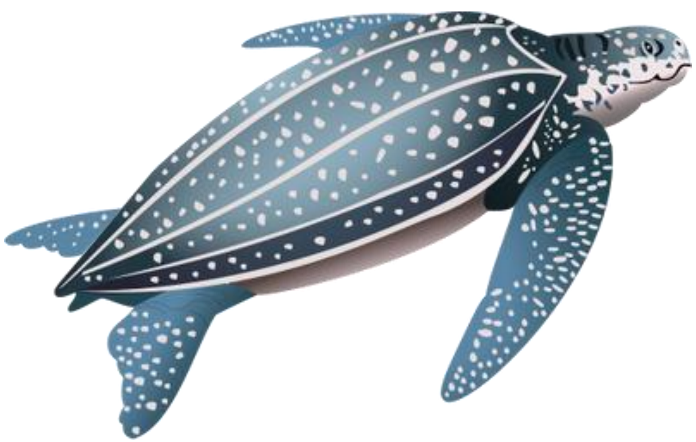
Thirteen ways of looking at a turtle
(NDC London 2025)



@ScottWlaschin

A taste of many different approaches

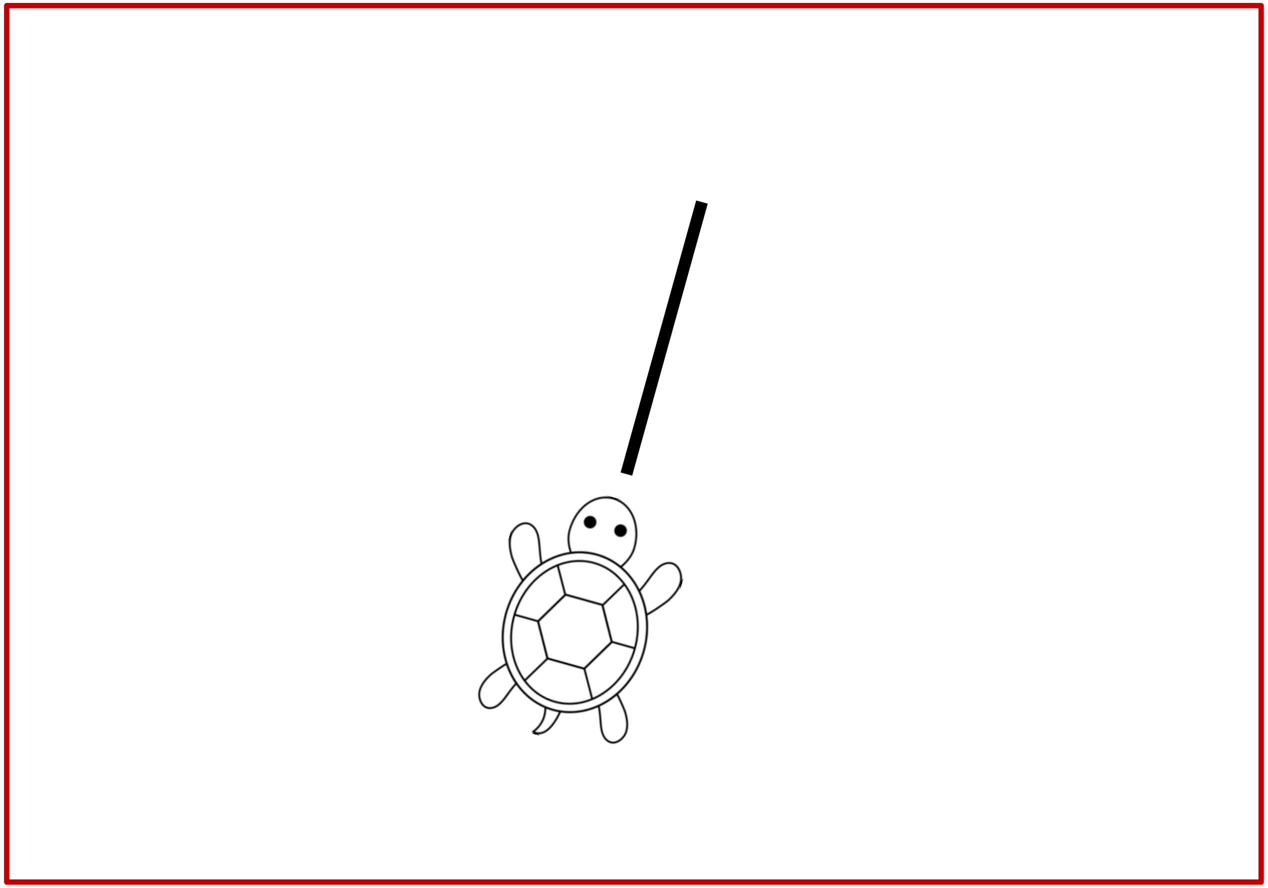
A taste of many different approaches

This is a crazy experiment:
~4 mins per topic!

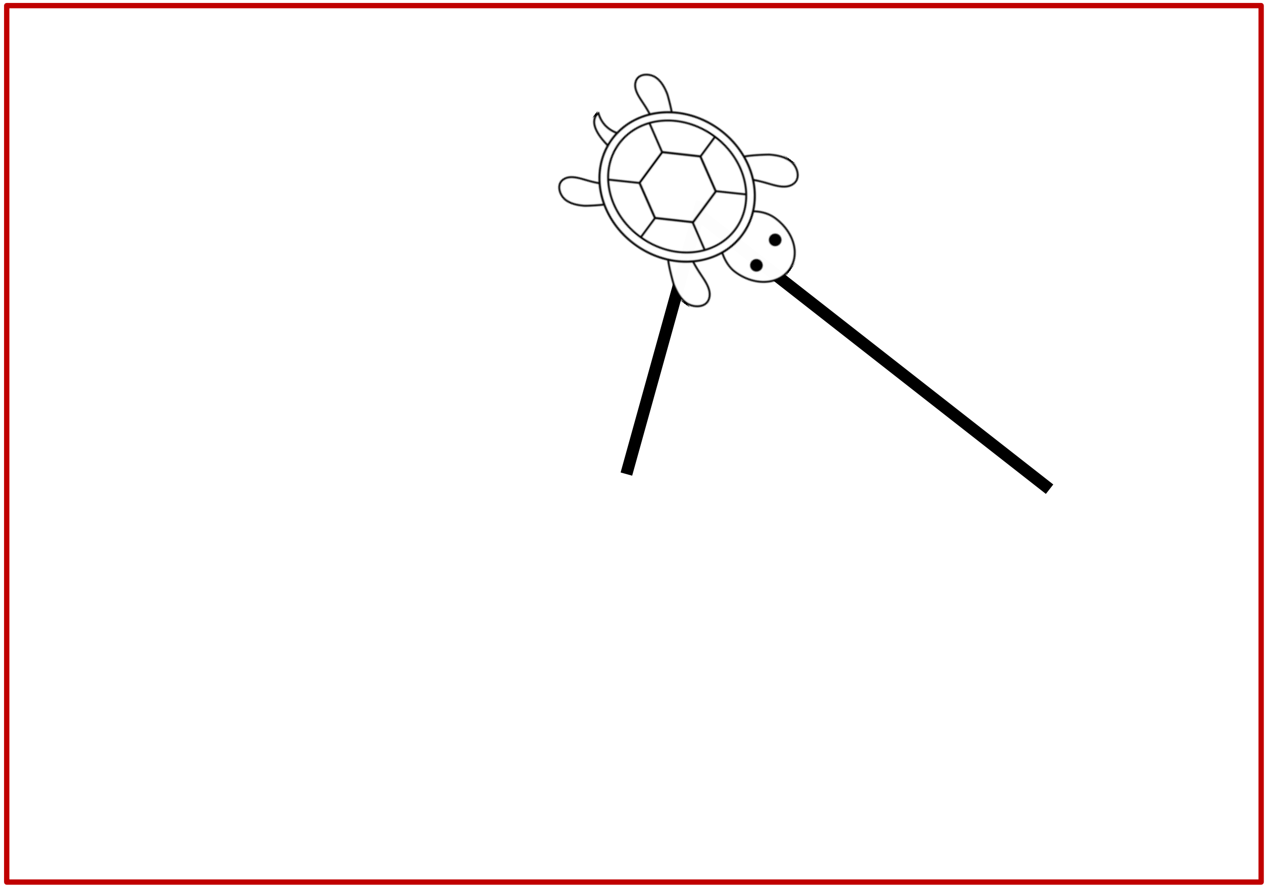
See also fsharpforfunandprofit.com/fppatterns

I'll be using F# code examples, but the concepts will work in most programming languages.

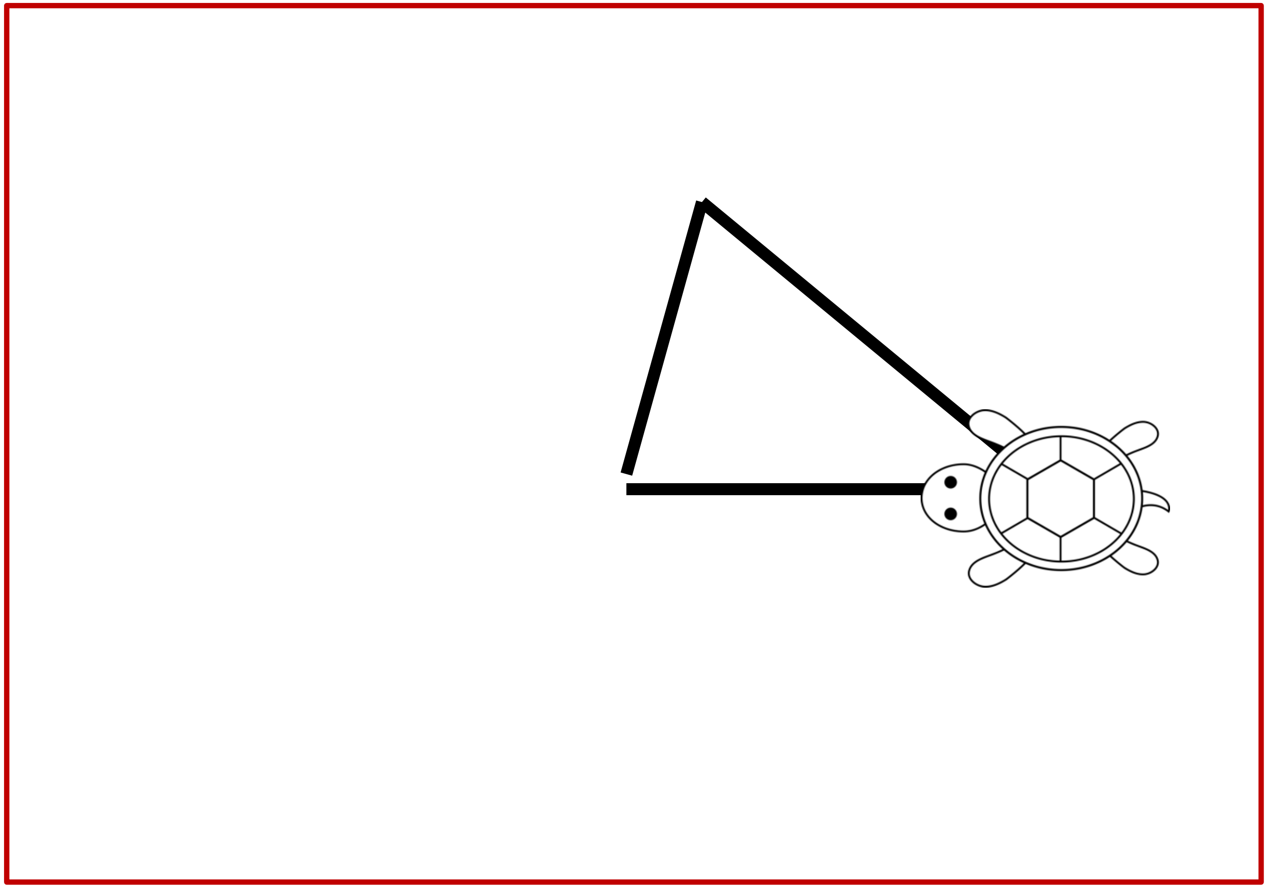
Turtle graphics in action



Turtle graphics in action



Turtle graphics in action



Turtle API

API	Description
`Move aDistance`	Move a distance in the current direction.
`Turn anAngle`	Turn N degrees clockwise
`PenUp` `PenDown`	Put the pen down or up. Moving the turtle draws a line only when pen is down

All of the following implementations will be based on this interface or some variant of it.

Three fundamental approaches

1. Object-Oriented
2. Abstract Data Type
3. Functional

1. Object Oriented Turtle

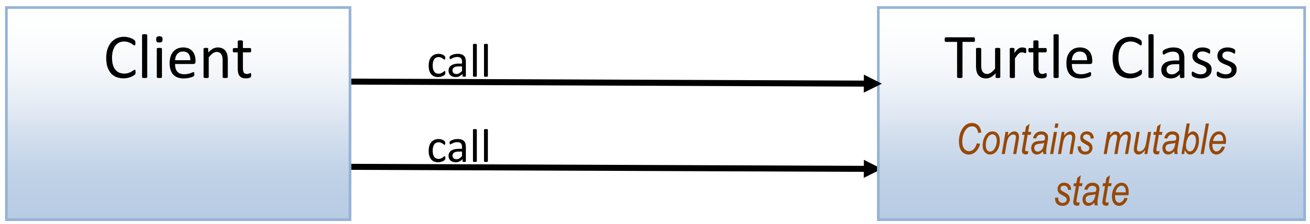
A "Tootle"

Data and behavior are combined

Object Oriented

- Focus on behavior over data
- Encapsulation of state
  - No setters and getters!

Overview



Turtle Implementation

```
// class definition
type Turtle() =

  // internal state
  let mutable position = {x=0.0; y=0.0}
  let mutable angle = 0.0
  let mutable penState = Down

  // methods
  member this.Move(distance) = ...
  member this.Turn(angleToTurn) = ...
  member this.PenUp() = ...
  member this.PenDown() = ...
```

"mutable" keyword needed in F#

No setters or getters

Turtle Implementation

```
// method definition
member this.Move(distance) =
  Logger.info (sprintf "Move %0.1f" distance)

  // do calculation
  let newPos = calcNewPosition(distance,angle,position)

  // draw line if needed
  if penState = Down then
     Canvas.drawLine(position,newPos)

  // update the state
  position <- newPos
```

Assignment

Turtle Implementation

```
// method definition
member this.Move(distance) =
  Logger.info (sprintf "Move %0.1f" distance)

  // do calculation
  let newPos = calcNewPosition(distance,angle,position)

  // draw line if needed
  if penState = Down then
    Canvas.drawLine(position,newPos)

  // update the state
  position <- newPos
```

Hard-coded dependencies!
(we'll fix this later)

Turtle Implementation

```
// method definition
member this.Turn(angleToTurn) =
  Logger.info (sprintf "Turn %0.1f" angleToTurn)

  // do calculation
  let newAngle = calcNewAngle(angleToTurn,angle)

  // update the state
  angle <- newAngle
```

Turtle Implementation

```
// method definition
member this.PenUp() =
  Logger.info "Pen up“
  // update the state
  penState <- Up

// method definition
member this.PenDown() =
  Logger.info "Pen down"
  // update the state
  penState <- Down
```

OO-style usage example

```
let drawTriangle() =
  let distance = 50.0

  let turtle = Turtle()
  turtle.Move(distance)
  turtle.Turn(120.0)
  turtle.Move(distance)
  turtle.Turn(120.0)
  turtle.Move(distance)
  turtle.Turn(120.0)
  // back home at (0,0) with angle 0
```

OO Turtle demo

Advantages and disadvantages

- Advantages
  - Familiar
- Disadvantages
  - Stateful (black box), hard to test
    - Add backdoors? ☹
  - Can't easily compose
    - How to move two turtles at once?
  - Hard to add user-defined behavior (like “triangle”)
    - Need extension methods
  - Hard-coded dependencies (for now)

2. Abstract Data Turtle

Data is separated from behavior

Abstract Data Types

- As with OO
  - Encapsulation of state
  - Focus on behavior over data
- But...
  - Data structure and behavior are separate
  - Client uses an opaque data structure (handle, ptr)

Opaque Data Structure

```
type TurtleHandle = private {
    mutable position : Position
    mutable angle : Angle
    mutable penState : PenState
}
```

Only turtle functions can access it

Behavior

```
module Turtle =
    let move(handle,distance) =  ...
    let turn(handle,angleToTurn) = ...
    let penUp(handle) = ...
    let penDown(handle) =
```

Handle passed in explicitly to every function

ADT usage example

```
let drawTriangle(handle) =
    let distance = 50.0
    Turtle.move(handle,distance)
    Turtle.turn(handle,120.0)
    Turtle.move(handle,distance)
    Turtle.turn(handle,120.0)
    Turtle.move(handle,distance)
    Turtle.turn(handle,120.0)
```

Handle (state) passed in explicitly

Advantages and disadvantages

- Advantages
  - Simple
  - Forces composition over inheritance!
  - Functions free to be moved around project
  - Easy to add user-defined behavior
    - no need for extension methods
- Disadvantages
  - As with OO: stateful, etc

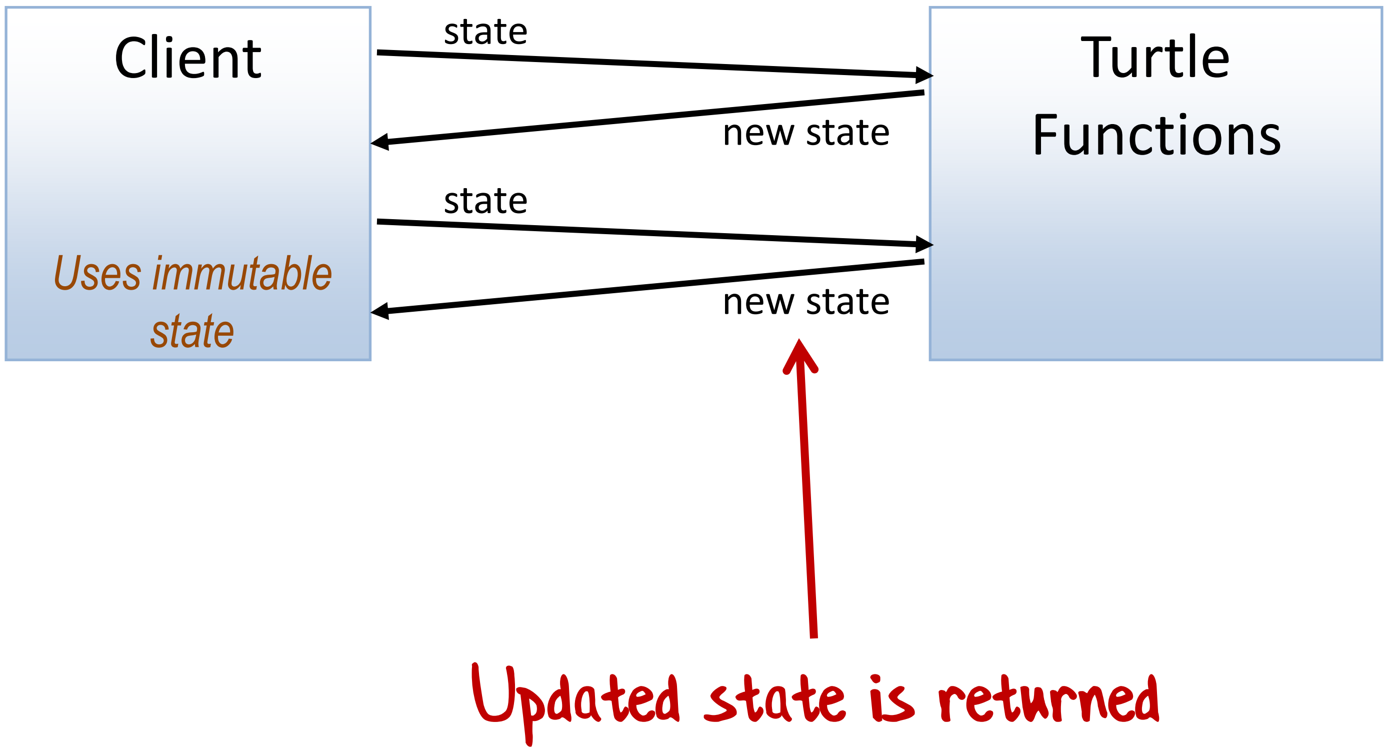
3. Functional Turtle

Data is immutable

Functional style

- Use functions, not methods
- All data is immutable
- No internal state – state is tracked by client

Overview



Client
Uses immutable state
state
new state
state
new state
Turtle Functions
Updated state is returned

Data

```
type TurtleState = {
  position : Position
  angle : Angle
  penState : PenState
}
```

Public, immutable

Behavior

State passed in explicitly

```
module Turtle =
  let move distance state =  ... // return new state
  let turn angleToTurn state = ... // return new state
  let penUp state = ... // return new state
  let penDown state = // return new state
```

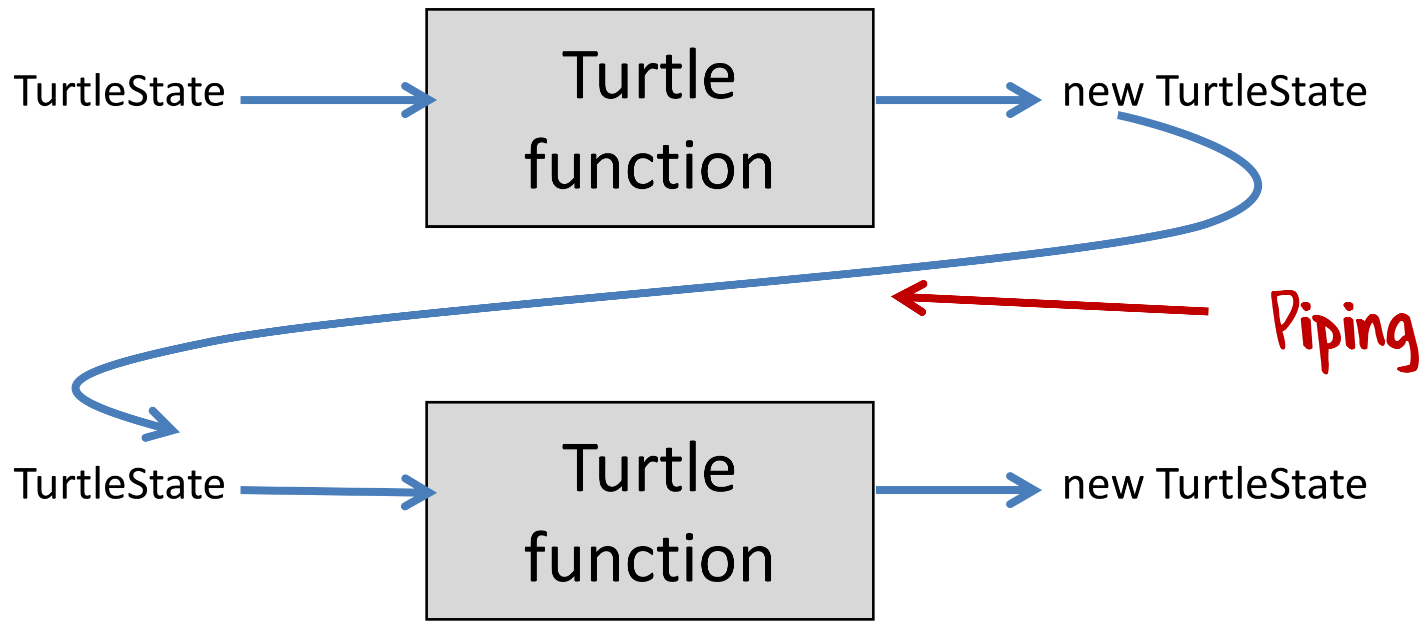
State returned

FP usage example

```
let drawTriangle() =
  let state0 = Turtle.initialTurtleState
  let state1 = Turtle.move 50.0 state0
  let state2 = Turtle.turn 120.0 state1
  let state3 = Turtle.move 50.0 state2
  ...
```

Great for testing!

But passing state around is annoying and ugly!


TurtleState
Turtle function
new TurtleState
Piping
TurtleState
Turtle function
new TurtleState

FP usage example with piping

```
let drawTriangle() =
  Turtle.initialTurtleState
    |> Turtle.move 50.0
    |> Turtle.turn 120.0
    |> Turtle.move 50.0
     ...
```

Nicer to read ☺

|> is pipe operator

Advantages and disadvantages

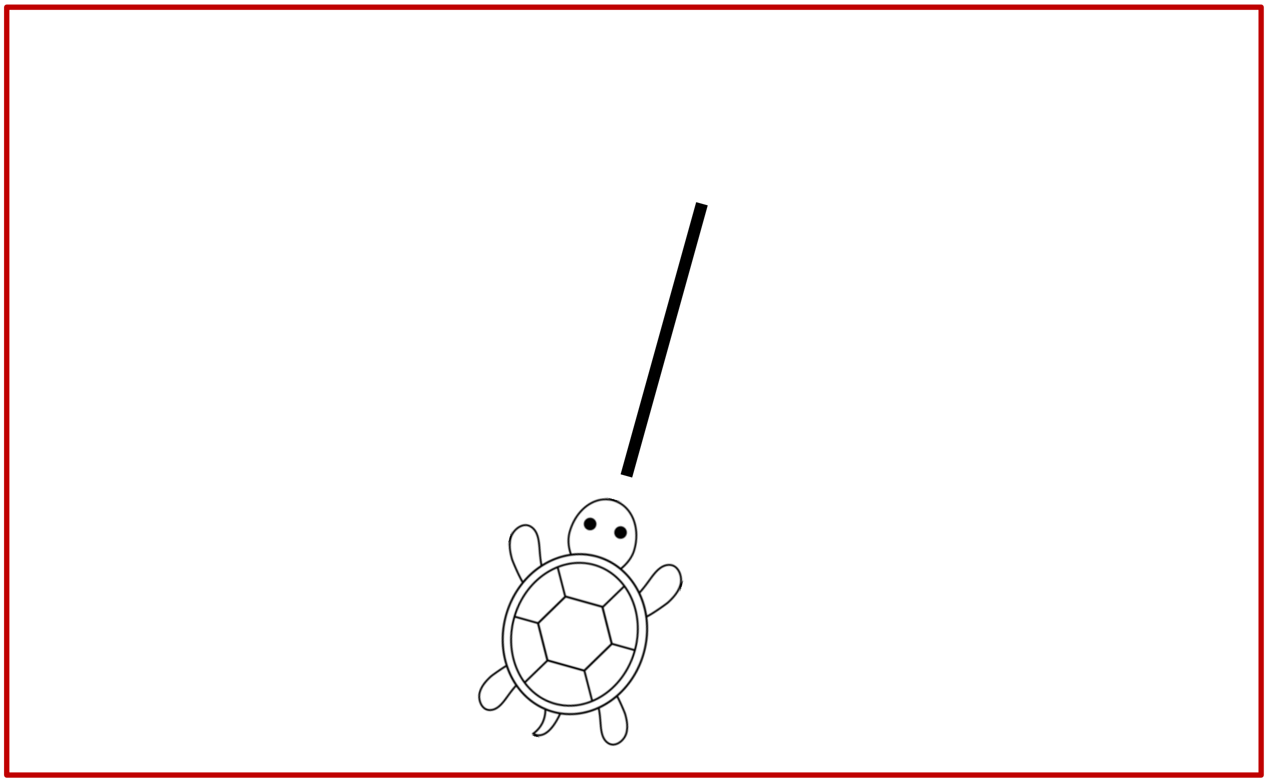
- Advantages
  - Immutability: Easy to reason about
  - Stateless: Easy to test
  - Functions are composable
- Disadvantages
  - Client has to keep track of the state ☹
  - Hard-coded dependencies (for now)

More complex turtles

4. Working with state
5. Working with errors
5½. Working with async

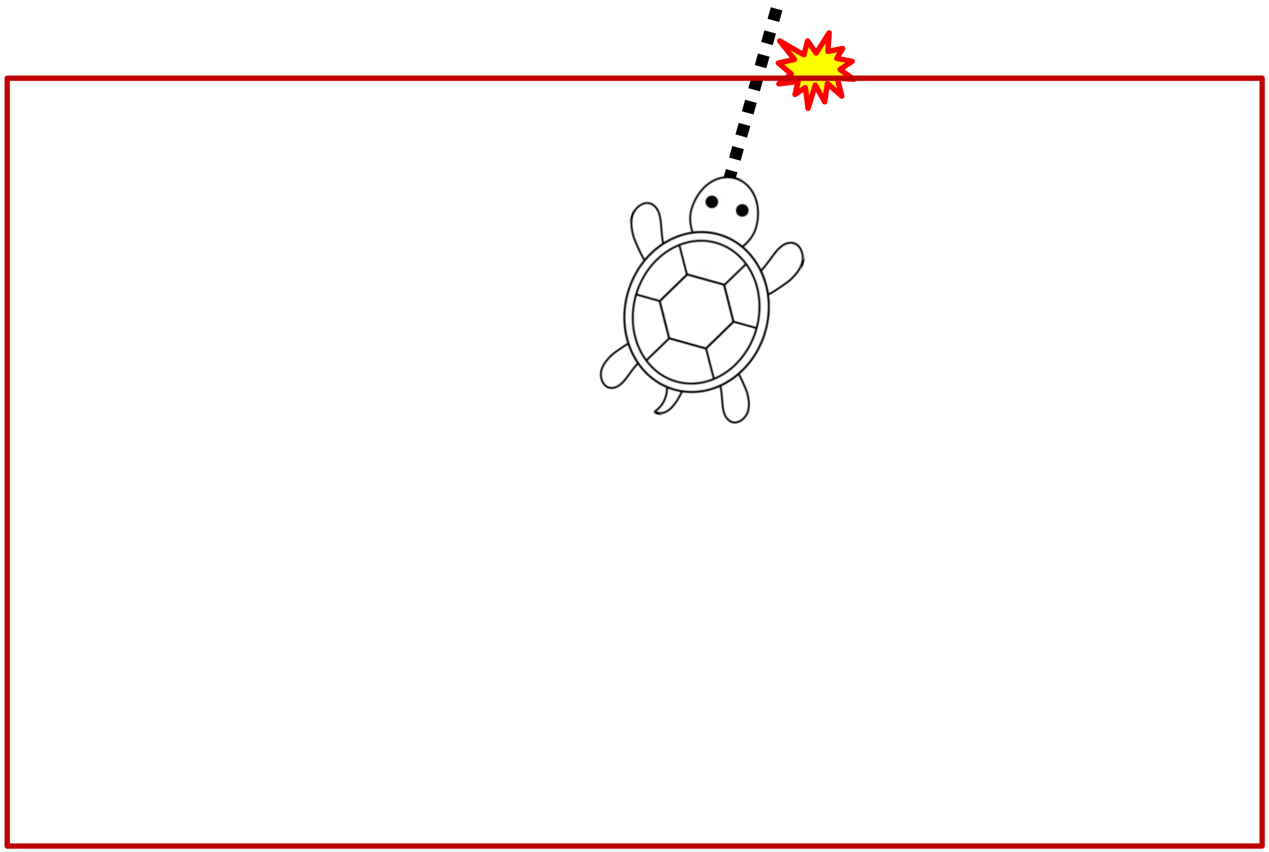
4. State monad

Threading state behind the scenes



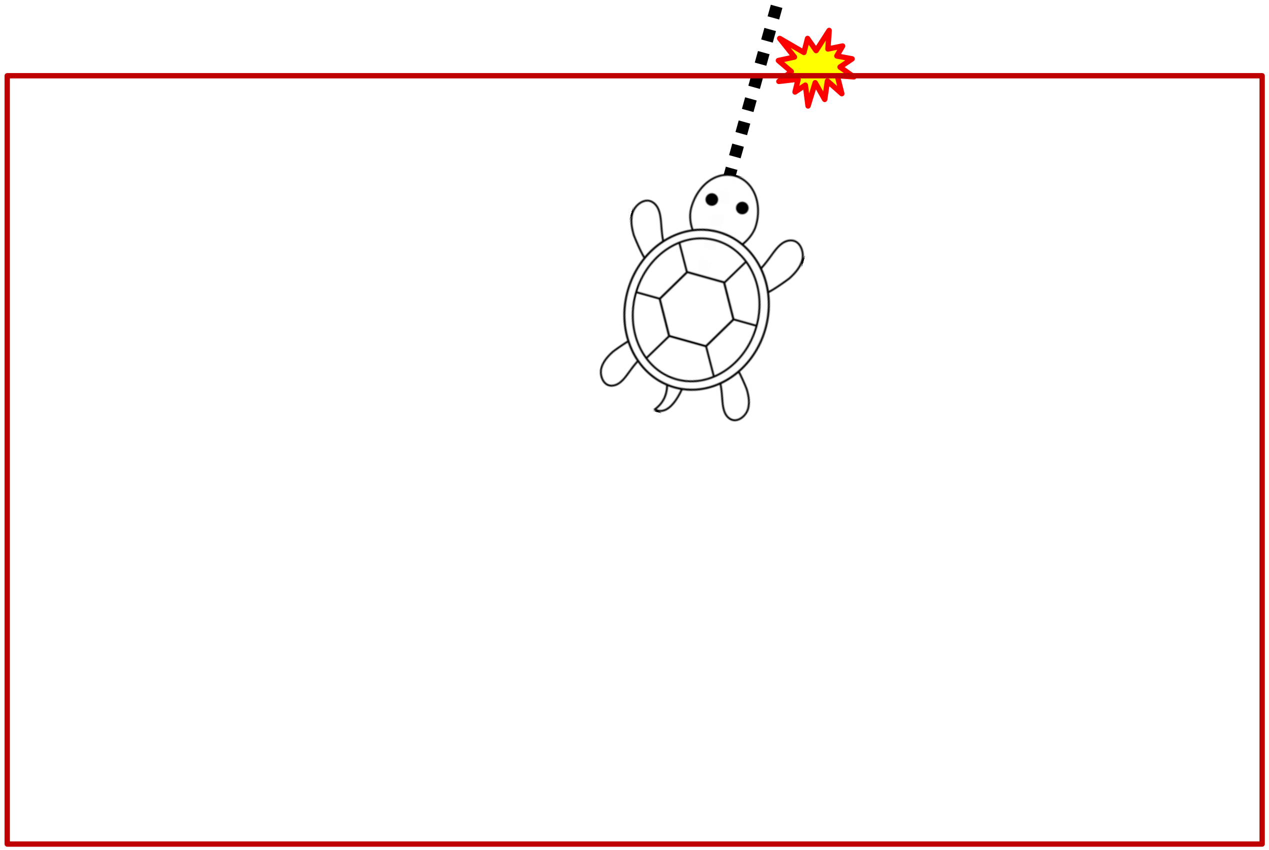
New requirement: there is a boundary that you can bump into.

You need to check the actual distance moved and change your behavior based on that.



New requirement: there is a boundary that you can bump into.

You need to check the actual distance moved and change your behavior based on that.



Fix the implementation to enable the requirement:

The `Move` function now returns a pair: New state AND actual distance moved

State usage example

The returned pair

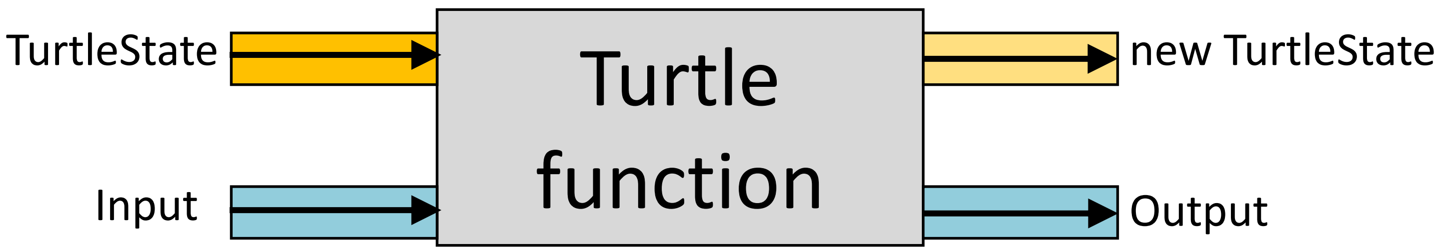
```
let s0 = Turtle.initialTurtleState
let (actualDistA,s1) = Turtle.move 80.0 s0
if actualDistA < 80.0 then
   log "first move failed -- turning"
   let s2 = Turtle.turn 120.0 s1
   let (actualDistB,s3) = Turtle.move 80.0 s2
   ...
else
   log "first move succeeded"
   let (actualDistC,s2) = Turtle.move 80.0 s1
   ...
```

Yuck!

How can we keep track of the state?

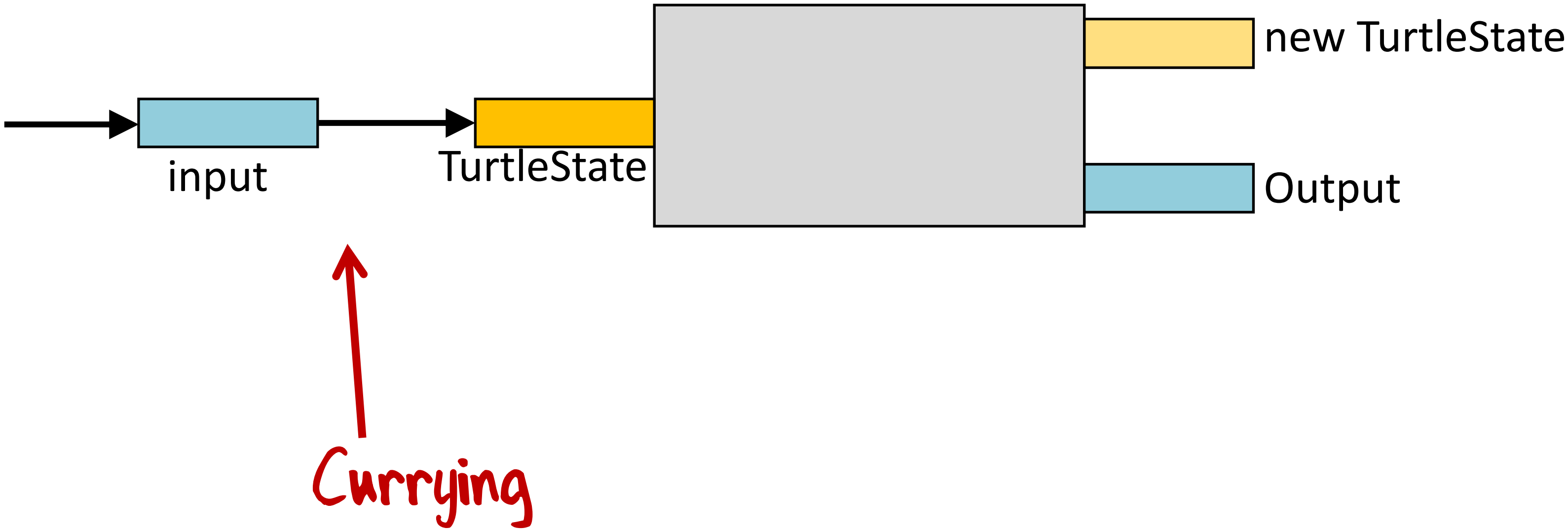
Piping doesn't work now ☹

Transforming the function #1

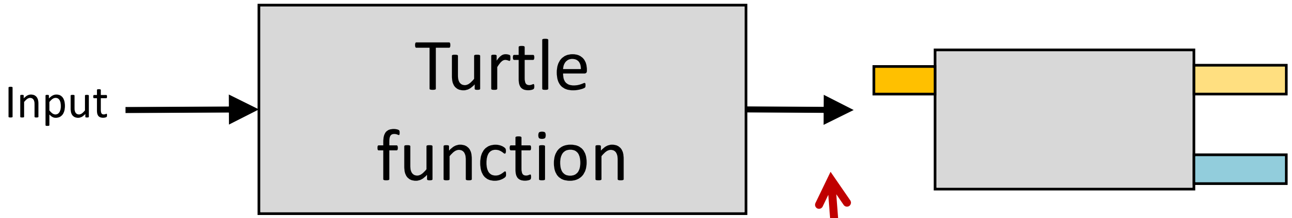


We can transform this into something nicer to work with

Transforming the function #2

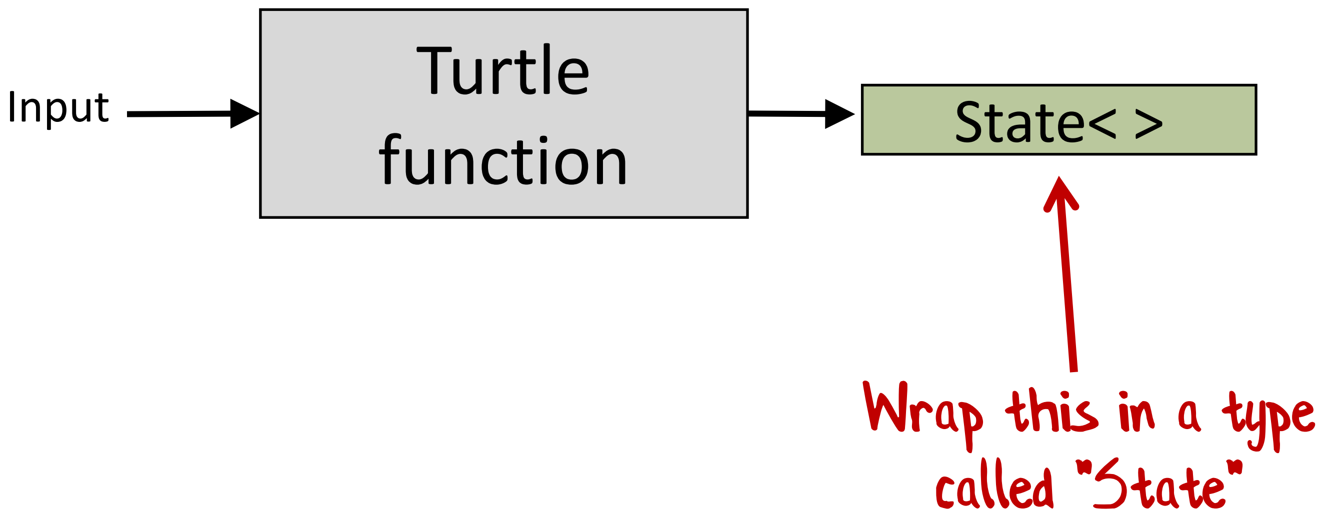


Transforming the function #3



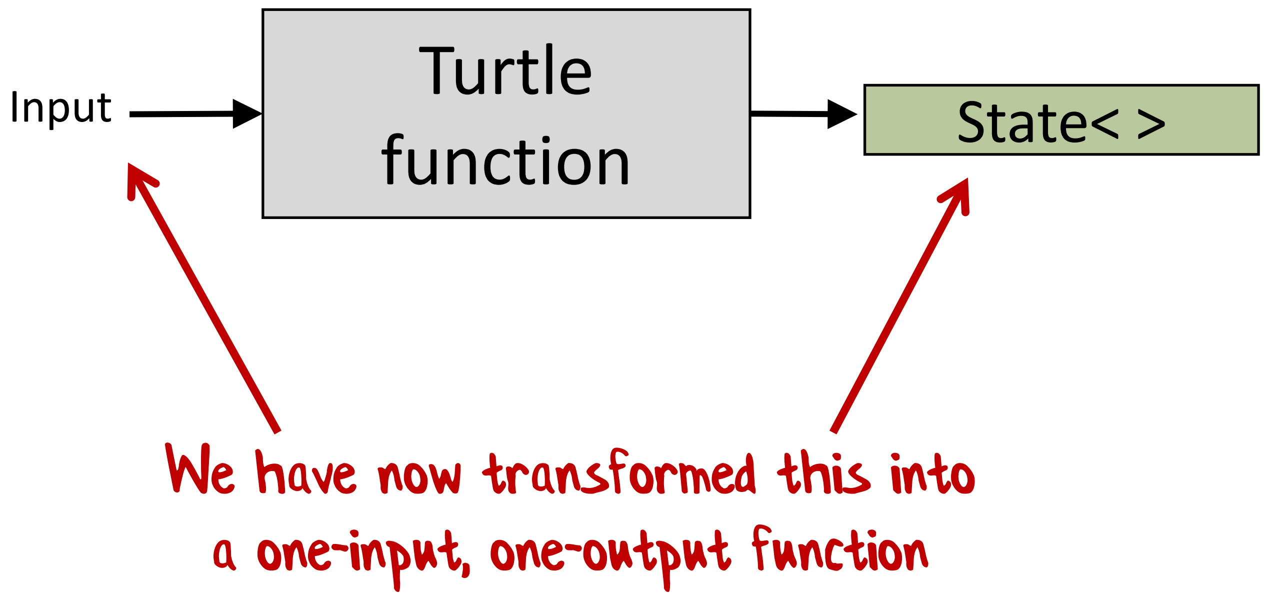
A function returns another function (common in FP!)

Transforming the function #4



For more on how this works, see fsharpforfunandprofit.com/monadster

Transforming the function #4



For more on how this works, see fsharpforfunandprofit.com/monadster

Using the State expression

```
let stateExpression = state {        // "state" expression
  let! distA = move 80.0
  if distA < 80.0 then
     log "first move failed -- turning"
     do! turn 120.0
     let! distB = move 80.0
     ...
  else
     log "first move succeeded"
     let! distB = move 80.0
     ...
}
```

State is threaded through behind the scenes

Haskell has "do" notation. Scala has "for" comprehensions

OO/imperative version

```
let distA = turtle.move 80.0
if distA < 80.0 then
   log "first move failed -- turning"
   turtle.turn 120.0
   let distB = turtle.move 80.0
```

State-expression version

```
state {
  let! distA = Turtle.move 80.0
  if distA < 80.0 then
     log "first move failed -- turning"
     do! Turtle.turn 120.0
     let! distB = Turtle.move 80.0
}
```

Looks similar to the imperative version!

OO/imperative version

```
let distA = turtle.move 80.0
if distA < 80.0 then
   log "first move failed -- turning"
   turtle.turn 120.0
   let distB = turtle.move 80.0
```

This is a function that works with mutable Turtle class

Not easily testable ☹

State-expression version

```
state {
  let! distA = Turtle.move 80.0
  if distA < 80.0 then
     log "first move failed -- turning"
     do! Turtle.turn 120.0
     let! distB = Turtle.move 80.0
}
```

This is a function that works with immutable TurtleState

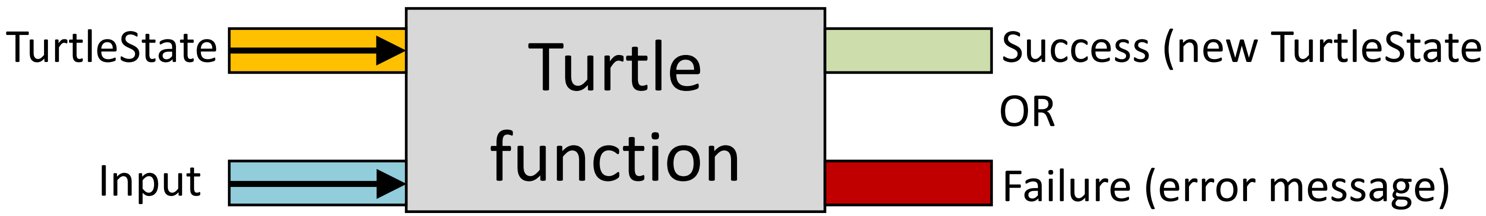
Easily testable ☺

Advantages and disadvantages

- Advantages
  - Looks imperative, but preserves immutability
  - Functions are still composable
  - Functions are easy to test
- Disadvantages
  - Harder to implement and use

5. Error handling

How to return errors?




```
type Result =
    | Ok of TurtleState
    | Error of ErrorInfo
```

Choice type (aka Sum, aka Discriminated Union)

Implementation using Result

```
let move distanceRequested state =
  // calculate new position...
  // draw line if needed...

  // return success or failure
  if actualDistanceMoved <> distanceRequested then
    Error "Moved out of bounds"
  else
    Ok {state with position = endPosition}
```

Two different choices for return value
(not easy in OO)

Using Result directly

```
let state0 = Turtle.initialTurtleState
let result1 = Turtle.move 80.0 state0
match result1 with
| Error msg ->
    log $"first move failed – {msg}"
| Ok state1 ->
    let result2 = Turtle.move 80.0 state1
    match result2 with
    | Error msg ->
        log $"second move failed: {msg}"
    | Ok state2 ->
        log "second move succeeded"
        ...
```

Again Yuck!

We will fix this using the same approach as State

Using the Result expression

```
let finalResult = result {        // result expression
    let state0 = Turtle.initialTurtleState
    let! state1 = Turtle.move 80.0 state0
    log "first move succeeded"
    let! state2 = Turtle.move 30.0 state1
    log "second move succeeded"
    let! state3 = Turtle.turn 120.0 state2
    let! state4 = Turtle.move 80.0 state3
    log "third move succeeded"
    return ()
    }
```

Errors are managed behind the scenes

You can stay focused on the happy path!

For more on how this works, see "Railway Oriented Programming"

Using the Result expression

```
let finalResult = result {       // result expression
    let state0 = Turtle.initialTurtleState
    let! state1 = Turtle.move 80.0 state0
    log "first move succeeded"
    let! state2 = Turtle.move 30.0 state1
    log "second move succeeded"
    let! state3 = Turtle.turn 120.0 state2
    let! state4 = Turtle.move 80.0 state3
    log "third move succeeded"
    return ()
    }
```

Still ugly with the explicit state though

State and Result expressions combined

```
let finalResult = resultState {
    do! Turtle.move 80.0
    log "first move succeeded"
    do! Turtle.move 30.0
    log "second move succeeded"
    do! Turtle.turn 120.0
    do! Turtle.move 80.0
    log "third move succeeded"
    return ()
    }
```

Both errors and state are now managed behind the scenes

Advantages and disadvantages

- Advantages
  - Errors are explicitly returned (no exceptions)
  - Looks like "happy path" code
    - But errors are being handled properly
- Disadvantages
  - Slightly harder to implement and use

5½. Async turtle

What if the Turtle was a physical object and the calls were async?

What if the Turtle calls were async?

```
let state0 = Turtle.initialTurtleState
state0 |> Turtle.moveAsync 80.0 (fun state1 ->
    state1 |> Turtle.moveAsync 80.0 (fun state2 ->
        state2 |> Turtle.moveAsync 80.0 (fun state3 ->
        ...
        )
    )
)
```

Callbacks

"Pyramid of doom"

Using the Async expression

'async' expression

```
let finalResult = async {
  let state0 = Turtle.initialTurtleState
  let! state1 = Turtle.moveAsync 80.0 state0
  let! state2 = Turtle.moveAsync 80.0 state1
  let! state3 = Turtle.moveAsync 80.0 state2
   ...
  }
```

Callbacks are now managed behind the scenes

'let!' is equivalent to 'async await' in C#

Using the Async expression

```
let finalResult = async {
  let state0 = Turtle.initialTurtleState
  let! state1 = Turtle.moveAsync 80.0 state0
  let! state2 = Turtle.moveAsync 80.0 state1
  let! state3 = Turtle.moveAsync 80.0 state2
   ...
  }
```

Callbacks are now managed behind the scenes

Managing state is still explicit though… ☹

…but we know how to fix that!

Using the combined Async/State expression

```
let finalResult = asyncState {
  do! Turtle.initialTurtleState
  do! Turtle.moveAsync 80.0
  do! Turtle.moveAsync 80.0
  do! Turtle.moveAsync 80.0
   ...
  }
```

Callbacks AND state are managed behind the scenes

Do you see a pattern?
(the m-word)

Common FP patterns

- FP likes composition
  - Always an output => you can pipe data
- FP likes explicitness
  - Explicit state management (no mutation)
  - Explicit errors (no exceptions)
- FP has techniques to hide ugliness
  - Can track state/errors/callbacks behind the scenes
  - The m-word!

Decoupled turtles

6. Batch Processing
6½. Actor model
7. Event Sourcing
8. Stream Processing

Options for working with state?

A: Turtle hides mutable state from caller ✘

– OO style

B: Caller keeps track of state everywhere ✓

– FP style

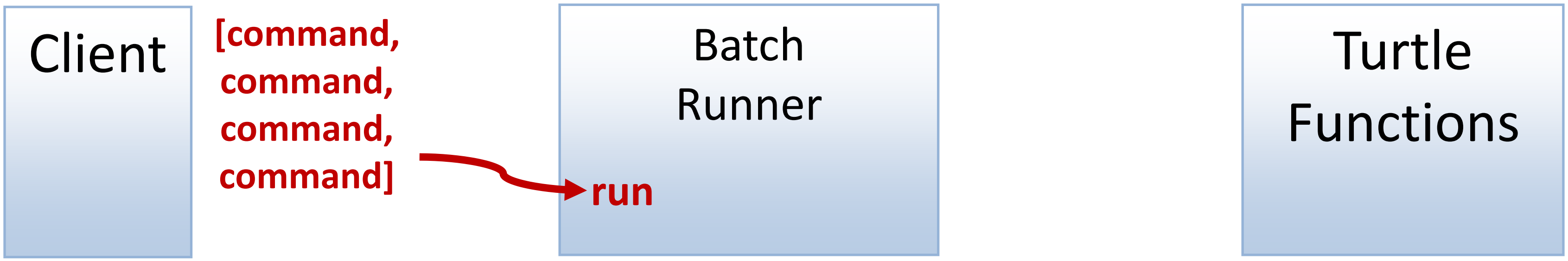
C: Someone else handles the state for you ✓

– Batching, agents, etc

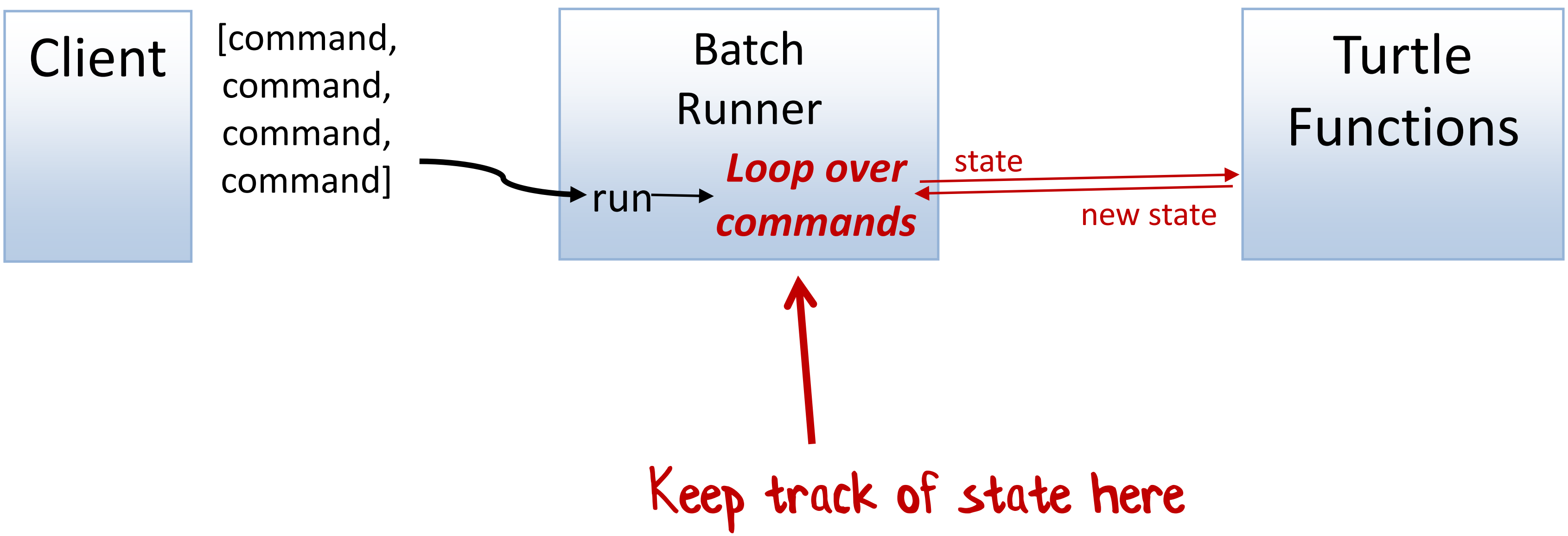
6. Batch commands

Helps the caller avoid managing state!

Overview


Client
[command,
command,
command,
command]
Batch
Runner
run
Turtle
Functions

Overview


Client
[command,
command,
command,
command]
Batch
Runner
run
Loop over
commands
state
new state
Turtle
Functions
Keep track of state here

How to convert a function into data?

```
// Turtle functions
let move distance = ...
let turn angle = ...
let penUp () = ...
let penDown () = ...
```

```
type TurtleCommand =
| Move of Distance
| Turn of Angle
| PenUp
| PenDown
```

Choice type

Usage example

```
// create the list of commands
let commands = [
    Move 100.0
    Turn 120.0
    Move 100.0
    Turn 120.0
    Move 100.0
    Turn 120.0
    ]

// run them
TurtleBatch.run commands
```

This is *data* not function calls

"*execute*" *implementation*

```
/// Apply a command to the turtle state
let executeCommand state command =
    match command with
    | Move distance ->
        Turtle.move distance state
    | Turn angleToTurn ->
        Turtle.turn angleToTurn state
    | PenUp ->
        Turtle.penUp state
    | PenDown ->
        Turtle.penDown state
```

"*execute*" *implementation*

```
/// Apply a command to the turtle state
let executeCommand state command =
    match command with
    | Move distance ->
      Turtle.move distance state
    | Turn angleToTurn ->
      Turtle.turn angleToTurn state
    | PenUp ->
      Turtle.penUp state
    | PenDown ->
      Turtle.penDown state
```

One-to-one correspondence
between data and action

"execute" implementation

```
/// Apply a command to the turtle state
let executeCommand state command =
    match command with
    | Move distance ->
        Turtle.move distance state
    | Turn angleToTurn ->
        Turtle.turn angleToTurn state
    | PenUp ->
        Turtle.penUp state
    | PenDown ->
        Turtle.penDown state
```

One-to-one correspondence
between data and action

"*run*" *implementation*

```
/// Run list of commands in one go
let run aListOfCommands =
  let mutable state = Turtle.initialTurtleState
  for command in aListOfCommands do
     state <- executeCommand state command
  // return final state
  state
```

Could also use "fold" here

"*run*" *implementation*

```
/// Run list of commands in one go
let run aListOfCommands =
  let initialState = Turtle.initialTurtleState
  aListOfCommands
  |> List.fold executeCommand initialState
```

Use built-in collection functions where possible

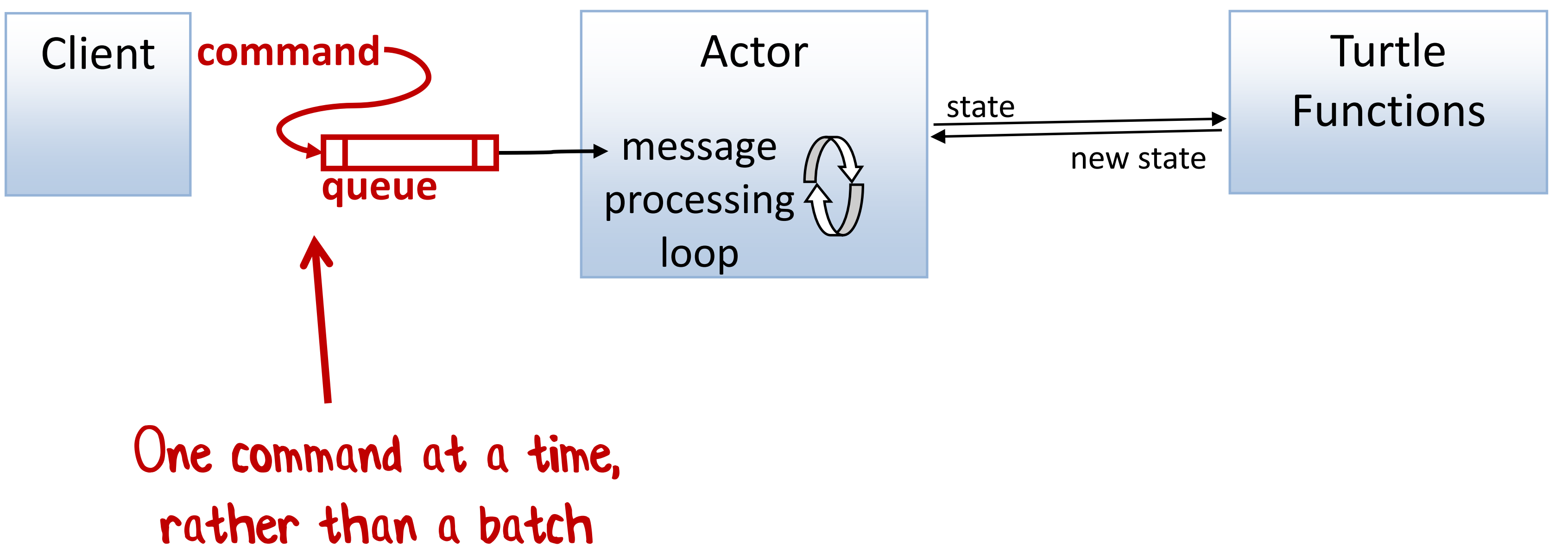
Advantages and disadvantages

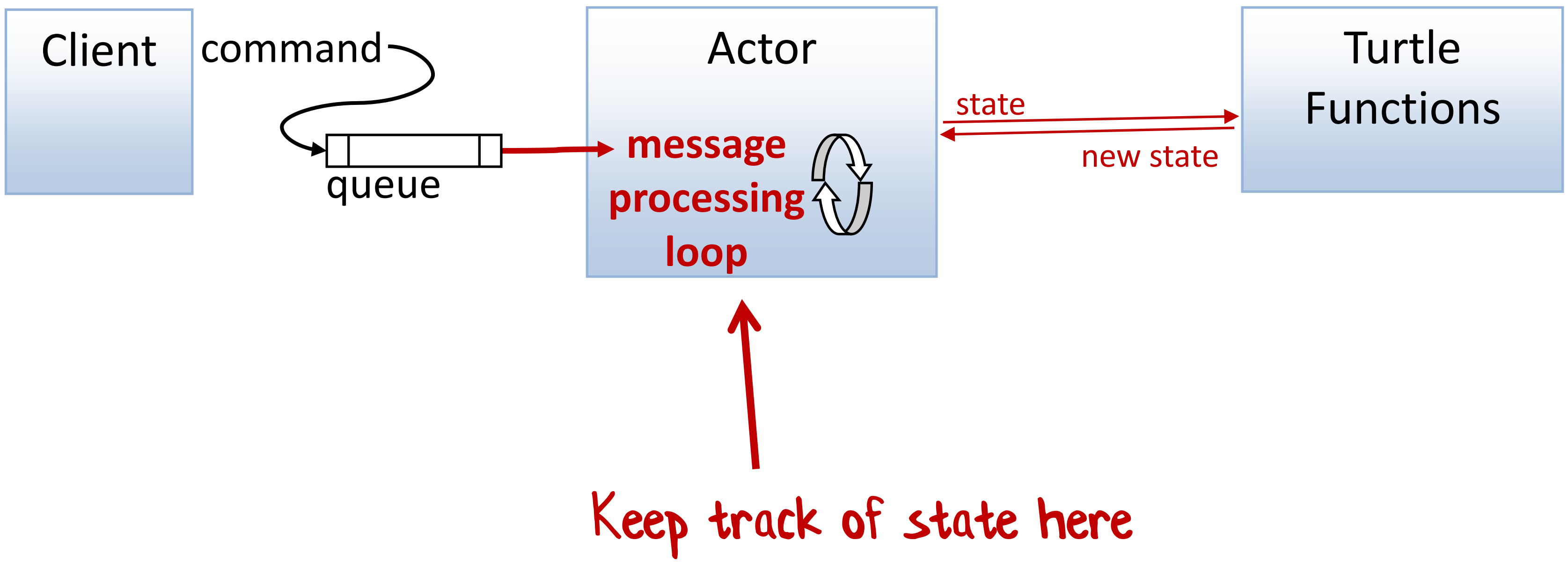
- Advantages
  - Decoupled – I don't care how the turtle works
  - Simpler than monads!
- Disadvantages
  - Batch oriented only
  - No control flow inside batch (handle errors, etc)

This can be fixed with the "interpreter" approach.
Stay tuned!

6½. Actor model

Real-time rather than batch


Client
command
queue
Actor
message
processing
loop
state
new state
Turtle
Functions
One command at a time,
rather than a batch


Client
command
queue
Actor
message processing loop
state
new state
Turtle Functions
Keep track of state here

Pulling commands off a queue

```
let rec loop turtleState =
    let command = readFromQueue() // block if empty
    let newState =
        match command with
        | Move distance ->
            Turtle.move distance turtleState
        // etc
    loop newState
```

Handle commands just like the "batch" implementation

Pulling commands off a queue

```
let rec loop turtleState =
    let command = readFromQueue() // block if empty
    let newState =
        match command with
        | Move distance ->
            Turtle.move distance turtleState
        // etc
    loop newState
```

Handle the next waiting command,
using the new state

Usage example

```
// post a list of commands
let turtleActor = new TurtleActor()
turtleActor.Post (Move 100.0)
turtleActor.Post (Turn 120.0)
turtleActor.Post (Move 100.0)
turtleActor.Post (Turn 120.0)
```

Again, this is *data*

Advantages and disadvantages

- Advantages
  - Decoupled in space (remote)
  - Decoupled in time (buffered, async)
  - Simpler than state monad
- Disadvantages
  - Extra boilerplate needed

7. Event Sourcing

What if we crash?
How should we persist state?

Store the journey not the destination

	White		Black
01.	e2-e4	01.	e7-e5
02.	Kt g1-f3	02.	Kt b8-c6
03.	B f1-b5	03.	a7-a6
04.	B b5-a4	04.	Kt g8-f6
05.	o-o	05.	b7-b5
06.	B a4-b3	06.	B f8-e7
07.	d2-d4	07.	d7-d6
08.	c2-c3	08.	B c8-g4
09.	B c1-e3	09.	o-o
10.	Kt b1-d2	10.	d6-d5
11.	e4 x d5	11.	Kt f6 x d5
12.	Q d1-c2	12.	e5 x d4
13.	B e3 x d4	13.	Kt c6 x d4
14.	Kt f3 x d4	14.	Q d8-d7
15.	Kt d2-f3	15.	B e7-f6
16.	Q c2-e4	16.	B f6 x d4
17.	B b3 x d5	17.	B g4 x f3
18.	Q e4 x f3	18.	Resigns

Keep track of events, not just final state

Store the journey not the destination

JOHN JONES
1643 DUNDAS ST W APT 27
TORONTO ON M6K 1V2

Statement period	Account No.
2003-10-09 to 2003-11-08	00005-123-456-7

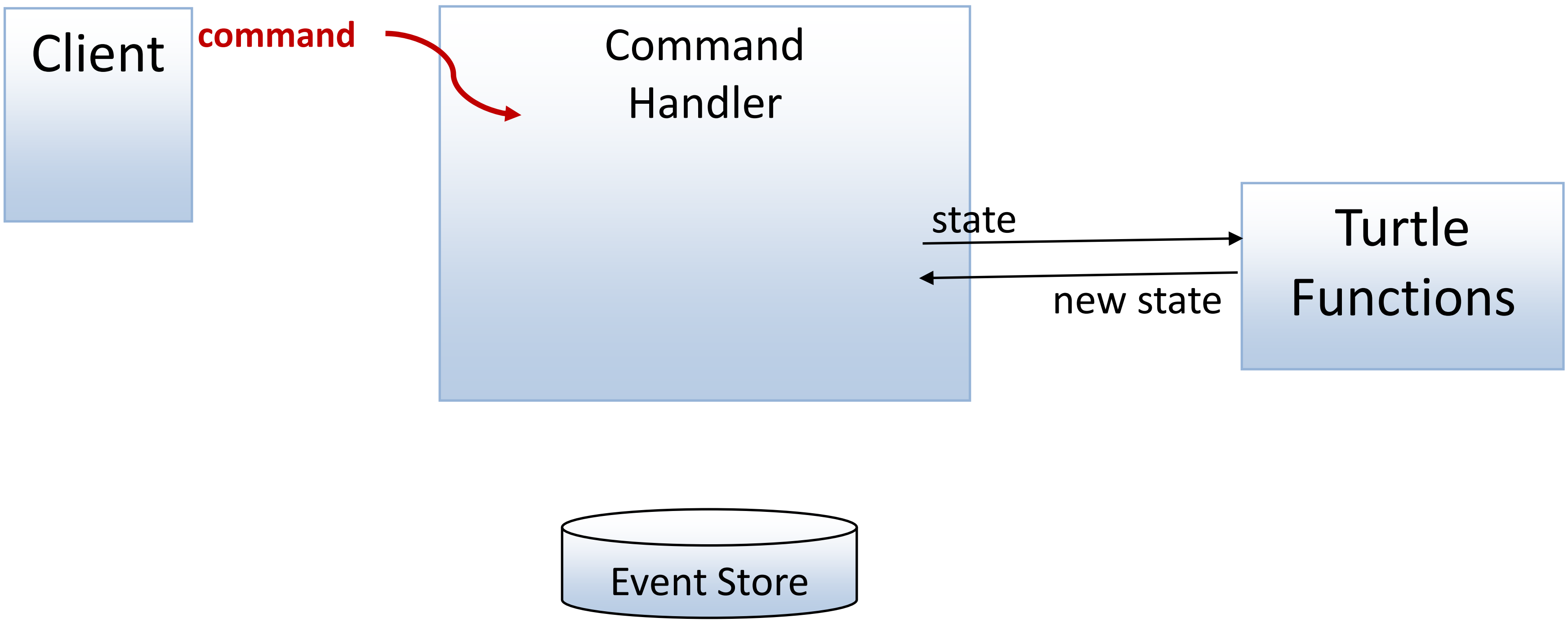
Date	Description	Ref.	Withdrawals	Deposits	Balance
2003-10-08	Previous balance				0.55
2003-10-14	Payroll Deposit - HOTEL			694.81	695.36
2003-10-14	Web Bill Payment - MASTERCARD	9685	200.00		495.36
2003-10-16	ATM Withdrawal - INTERAC	3990	21.25		474.11
2003-10-16	Fees - Interac		1.50		472.61
2003-10-20	Interac Purchase - ELECTRONICS	1975	2.99		469.62
2003-10-21	Web Bill Payment - AMEX	3314	300.00		169.62
2003-10-22	ATM Withdrawal - FIRST BANK	0064	100.00		69.62
2003-10-23	Interac Purchase - SUPERMARKET	1559	29.08		40.54
2003-10-24	Interac Refund - ELECTRONICS	1975		2.99	43.53
2003-10-27	Telephone Bill Payment - VISA	2475	6.77		36.76
2003-10-28	Payroll Deposit - HOTEL			694.81	731.57
2003-10-30	Web Funds Transfer - From SAVINGS	2620		50.00	781.57
2003-11-03	Pre-Auth. Payment - INSURANCE		33.55		748.02
2003-11-03	Cheque No. - 409		100.00		648.02
2003-11-06	Mortgage Payment		710.49		-62.47
2003-11-07	Fees - Overdraft		5.00		-67.47
2003-11-08	Fees - Monthly		5.00		-72.47
	*** Totals ***		1,515.63	1,442.61	

Keep track of events,
not just final state

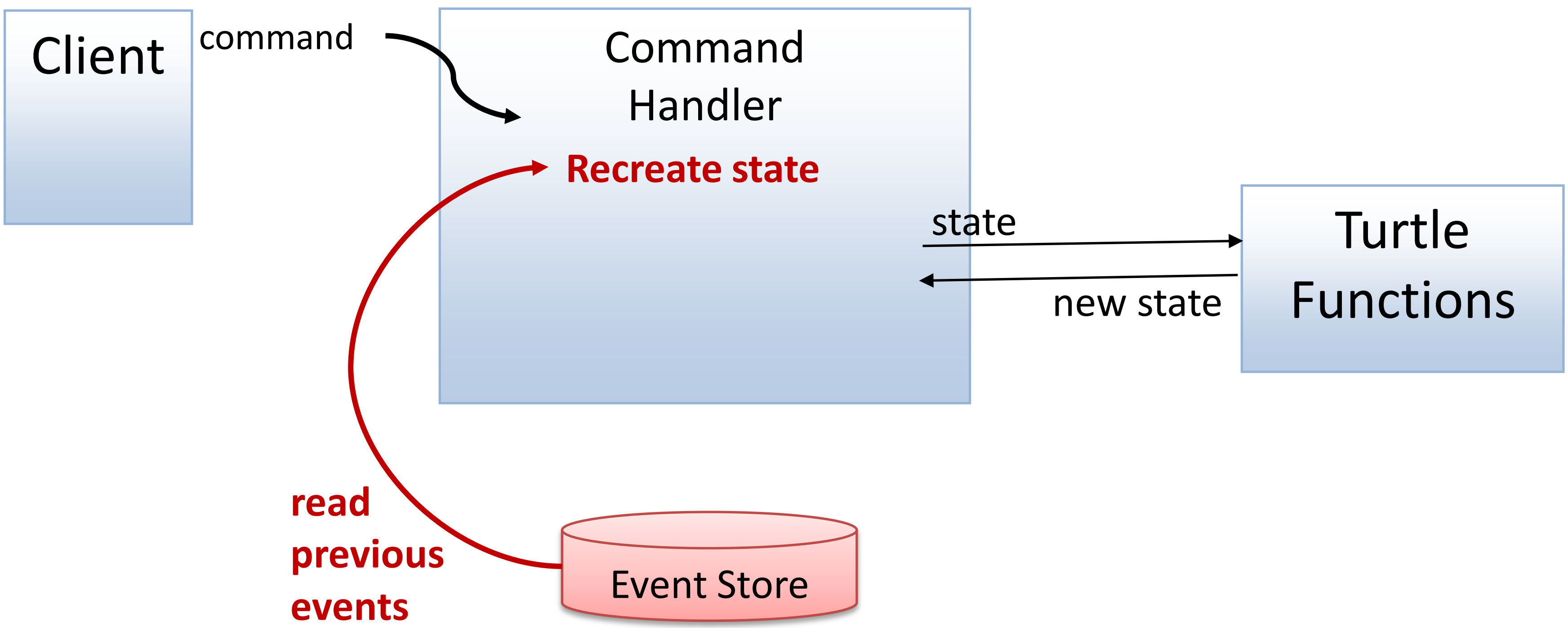
Event sourcing

- Store events not just final state
- Rebuild state from stored events
- After processing a command, store a new event

Overview

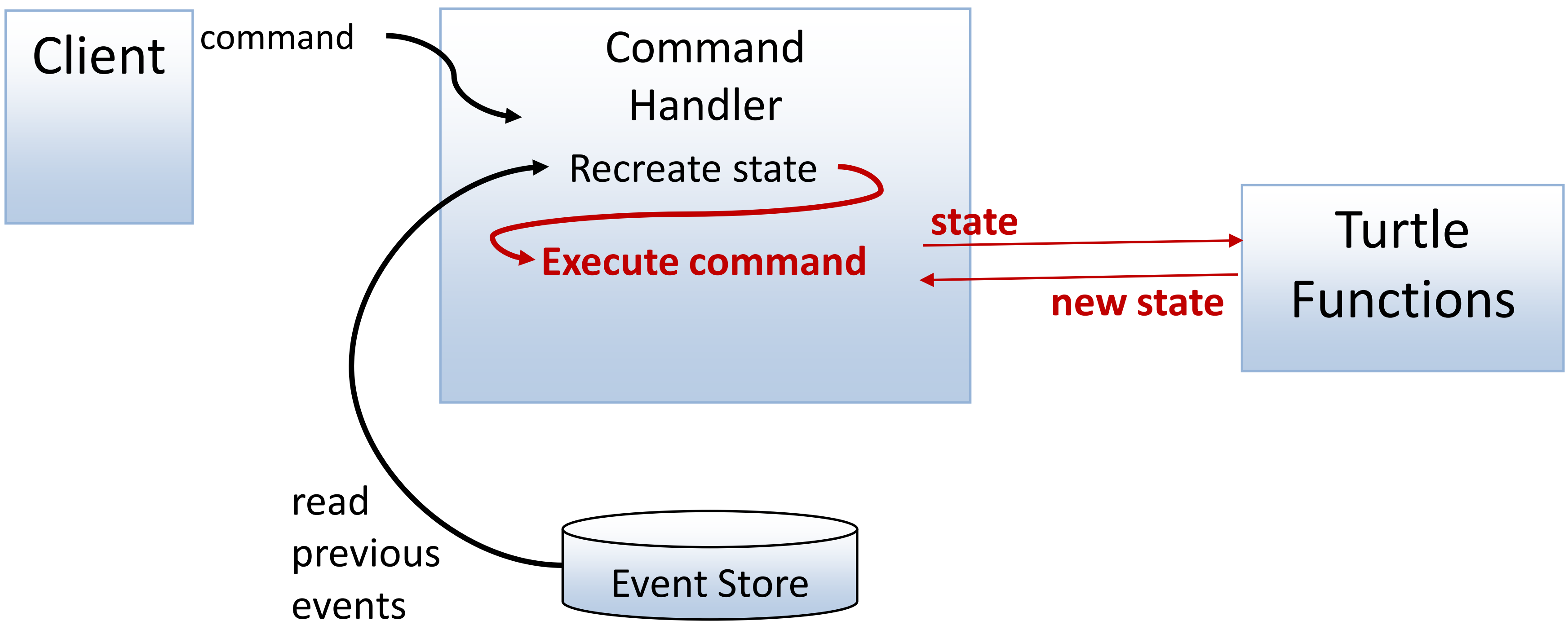

Client
command
Command Handler
state
new state
Turtle Functions
Event Store

Overview

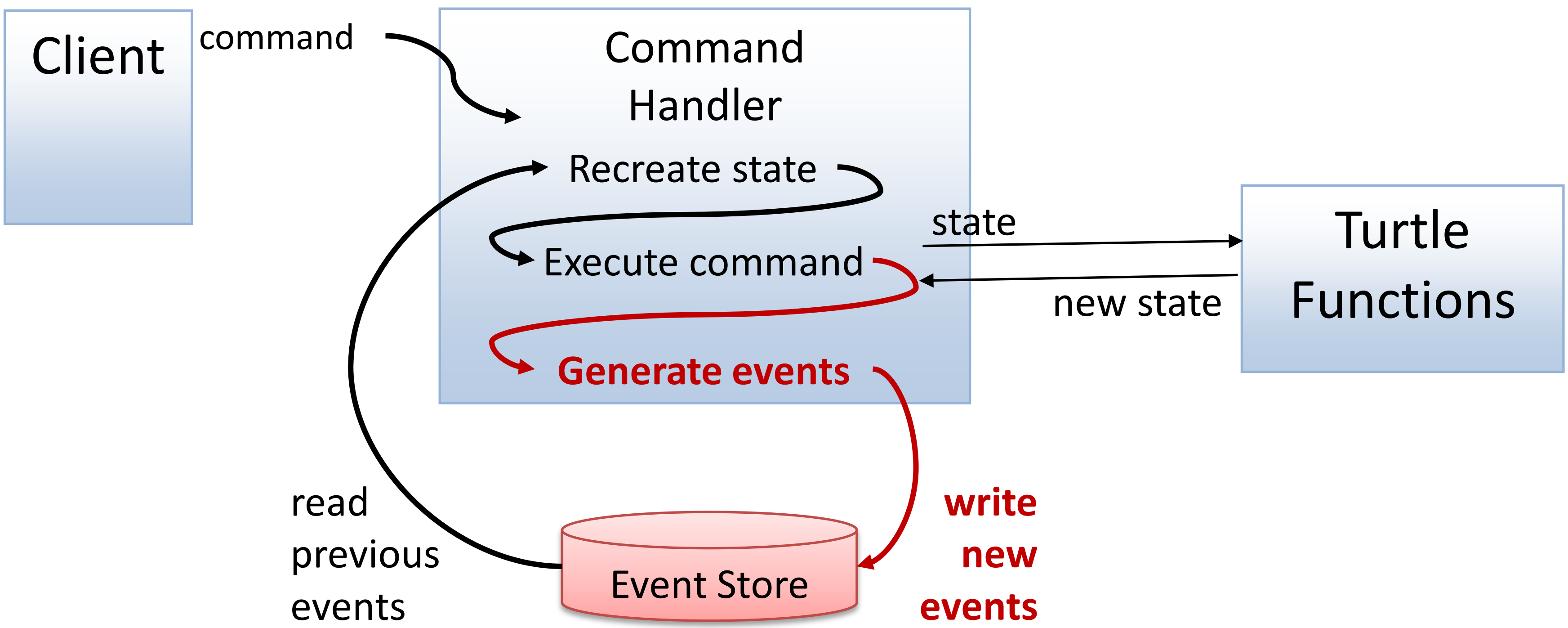


Client
command
Command
Handler
Recreate state
state
new state
Turtle
Functions
read
previous
events
Event Store

Overview


Client
command
Command Handler
Recreate state
Execute command
state
new state
Turtle Functions
read previous events
Event Store

Overview



Client
command
Command
Handler
Recreate state
Execute command
Generate events
state
new state
Turtle
Functions
read
previous
events
Event Store
write
new
events

Command vs. Event

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
```

What you *want* to have happen

```
type TurtleEvent =
    | Moved of Distance * StartPosition * EndPosition
    | Turned of AngleTurned * FinalAngle
    | PenStateChanged of PenState
```

What *actually* happened (past tense)
This is what gets stored in the event-store

Advantages and disadvantages

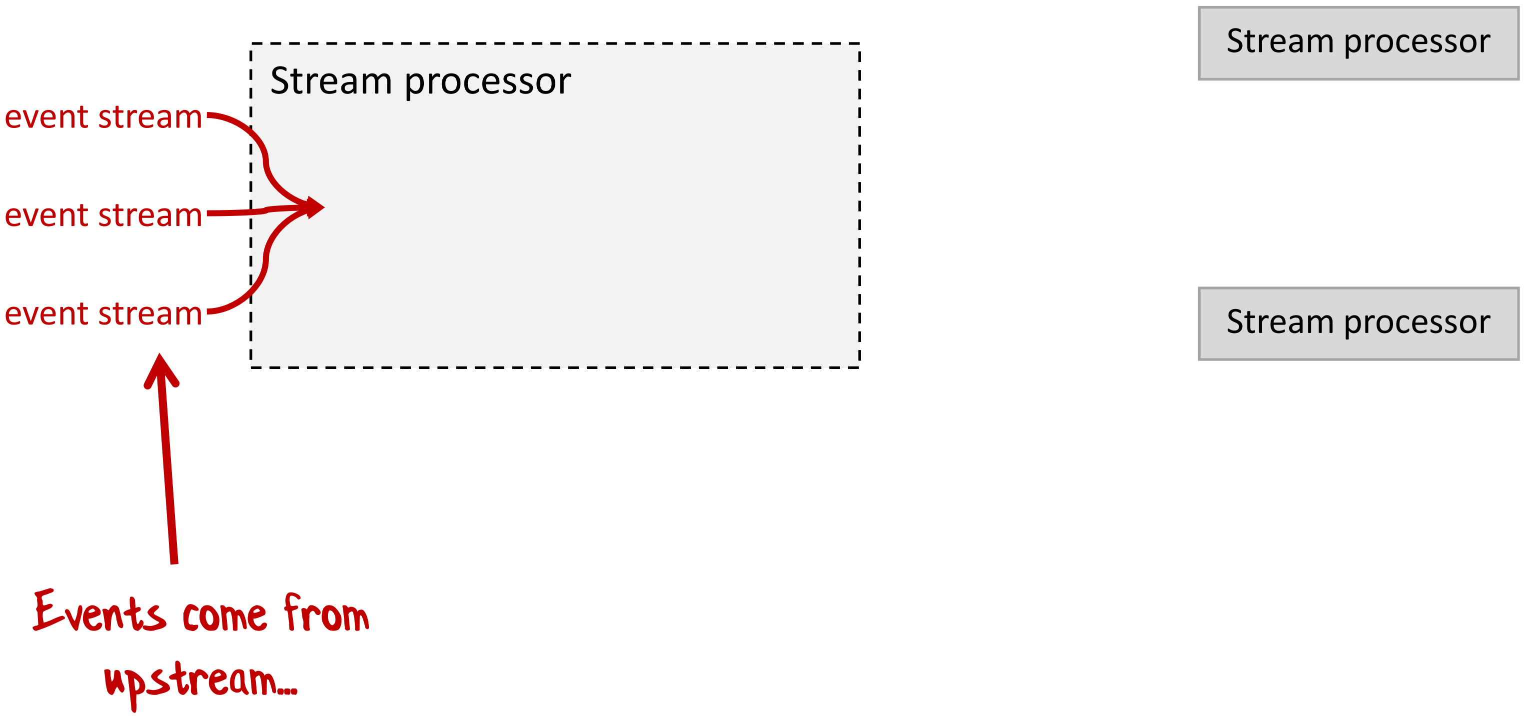
- Advantages
  - Decoupled
  - Stateless in memory (can crash and recover)
  - Supports replay of events
    - Good for audit trails, compliance, traceability
- Disadvantages
  - More complex
  - Versioning events can be tricky

8. Stream Processing

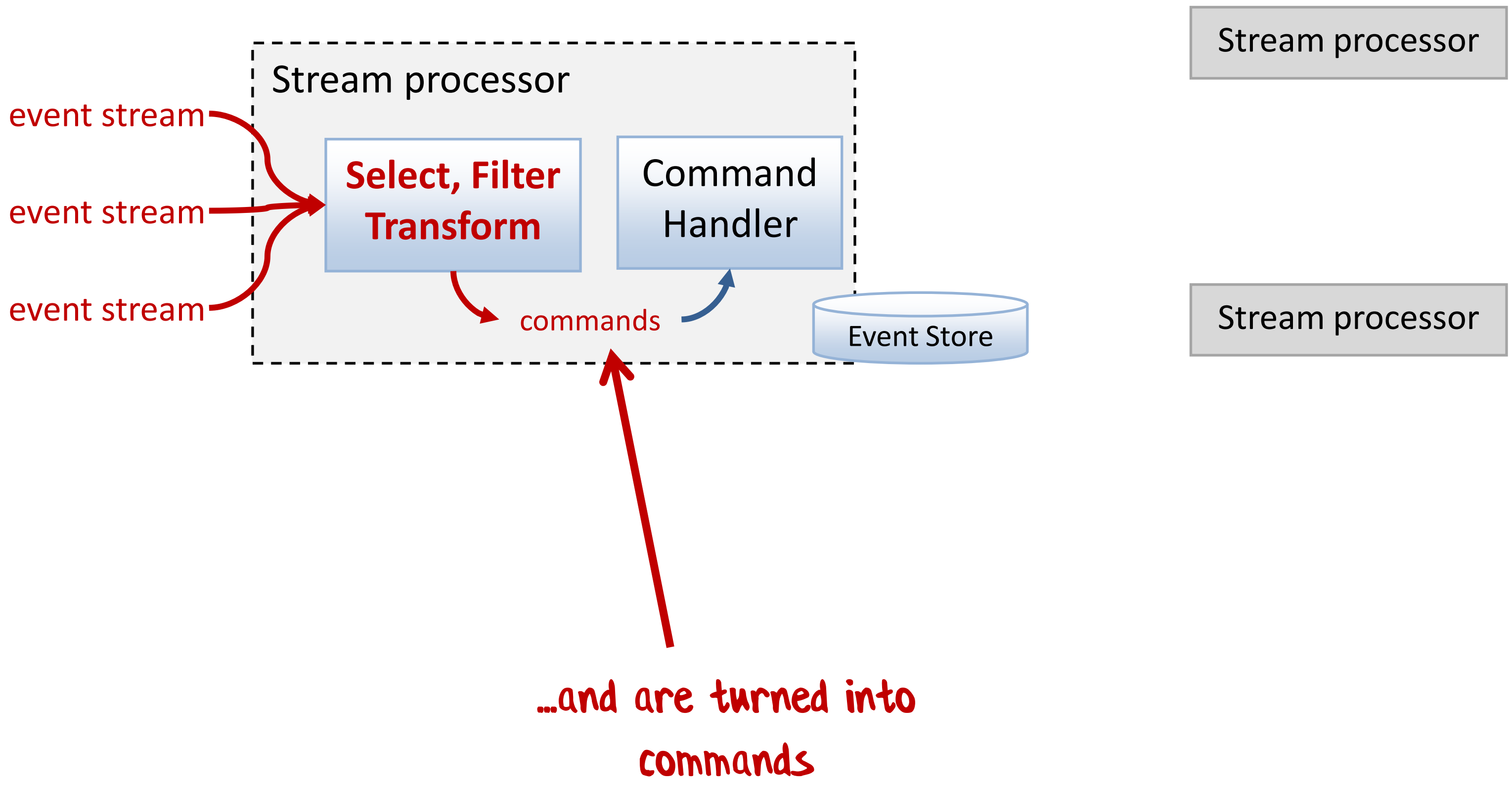
Event sourcing

- Events are published on a stream
- Separate decision-making from actions
  - Event generators create events
  - Event consumers do actions

Overview

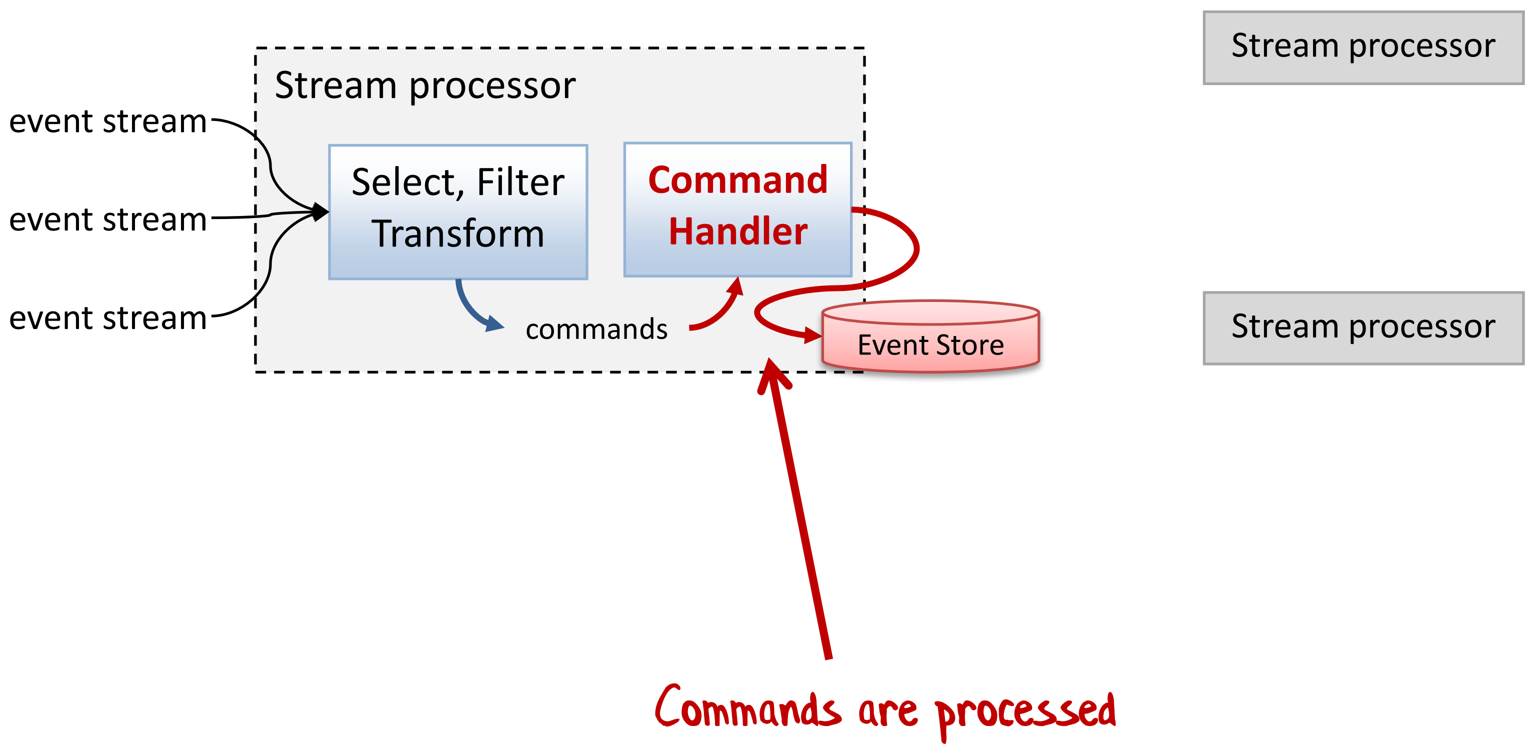

Stream processor
event stream
event stream
event stream
Events come from upstream...
Stream processor
Stream processor

Overview

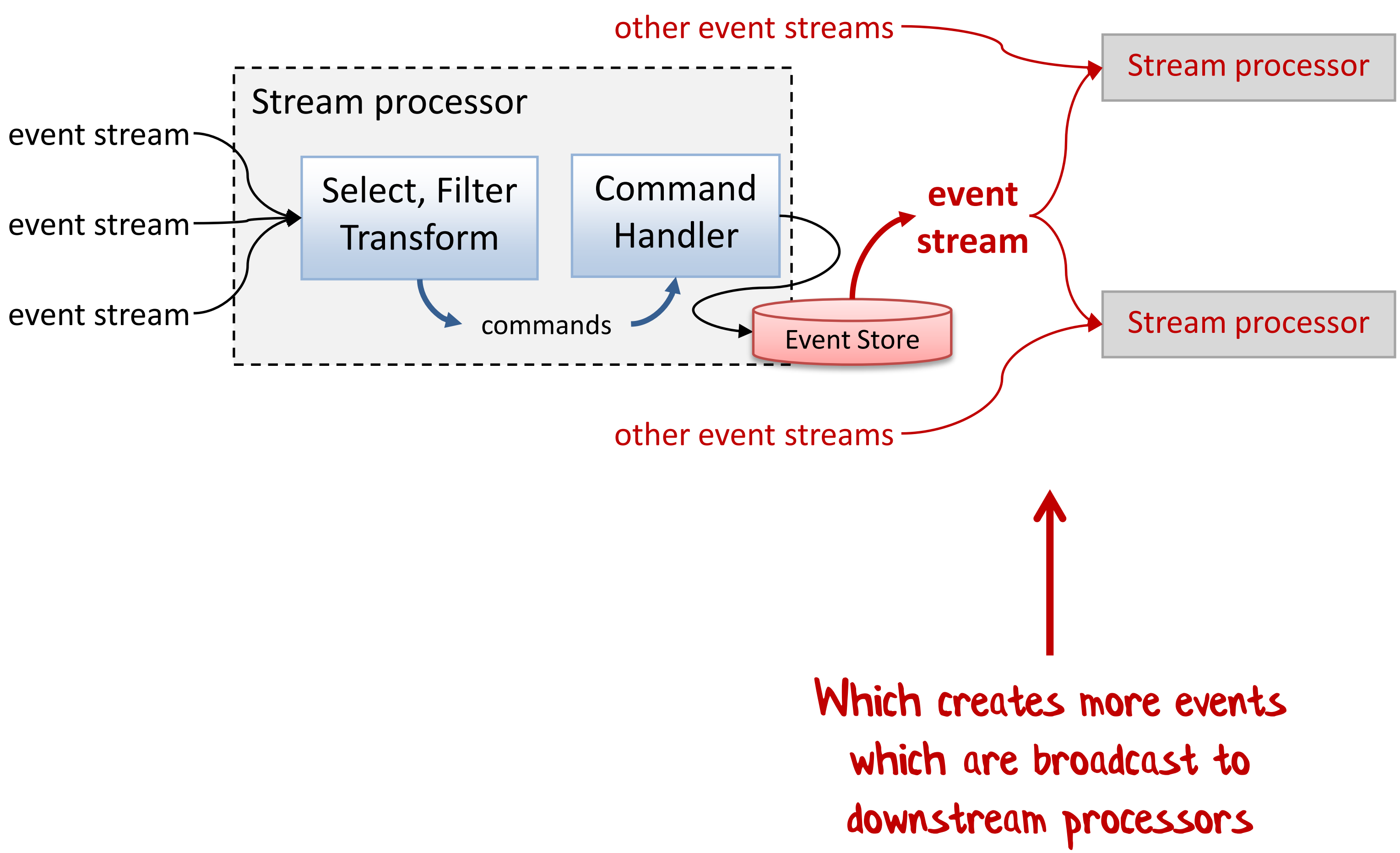


Stream processor
event stream
event stream
event stream
Select, Filter
Transform
Command
Handler
commands
Event Store
Stream processor
Stream processor
...and are turned into
commands

Overview

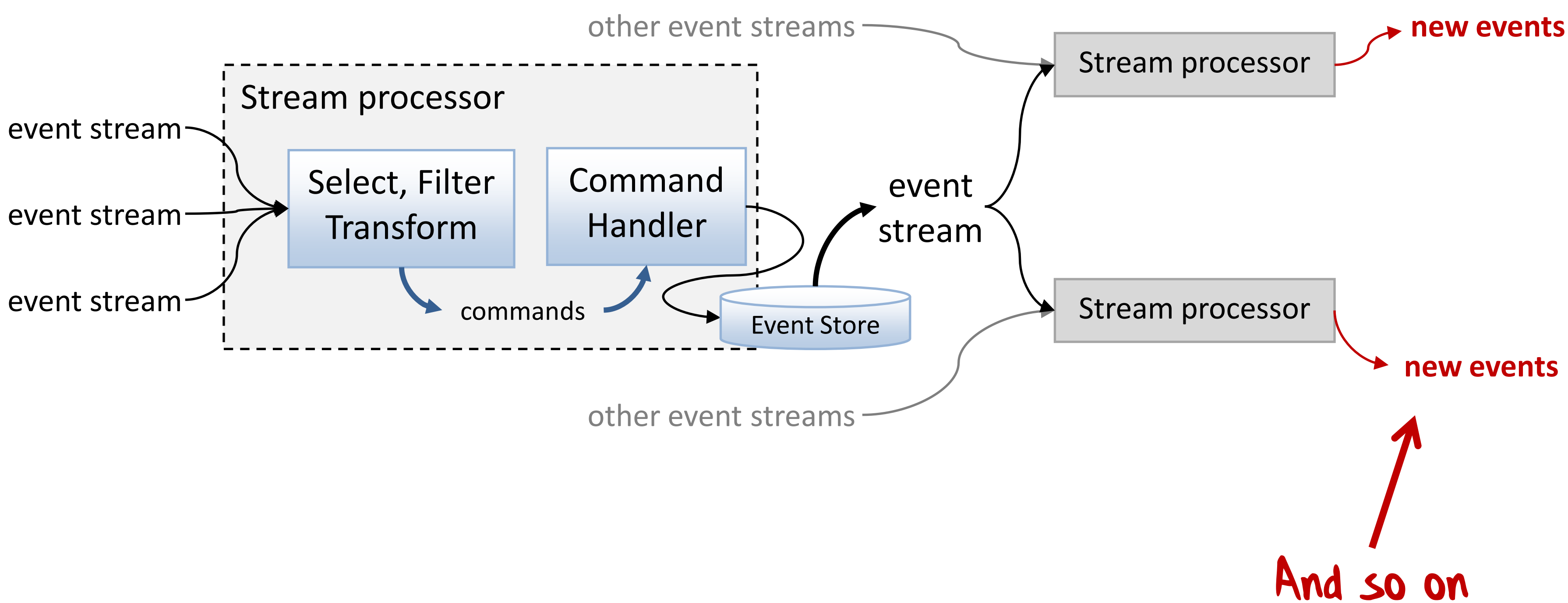

Stream processor
event stream
event stream
event stream
Select, Filter
Transform
Command
Handler
commands
Event Store
Stream processor
Stream processor
Commands are processed

Overview

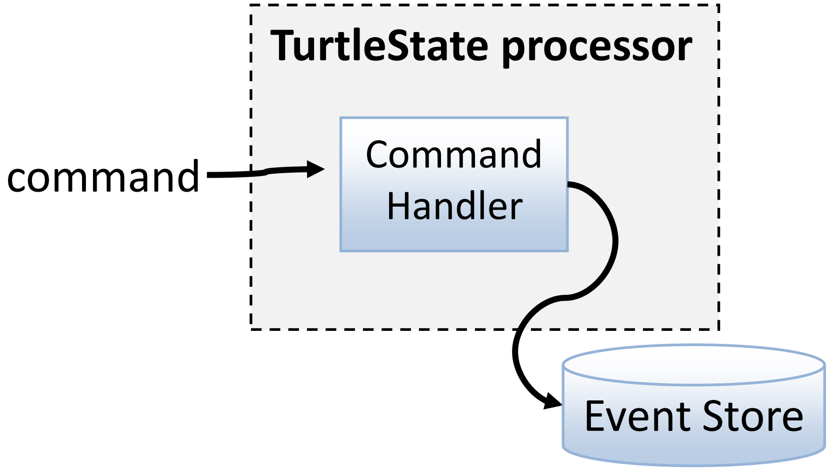


other event streams
Stream processor
event stream
event stream
event stream
Select, Filter
Transform
Command
Handler
commands
Event Store
event
stream
Stream processor
Stream processor
other event streams
Which creates more events
which are broadcast to
downstream processors

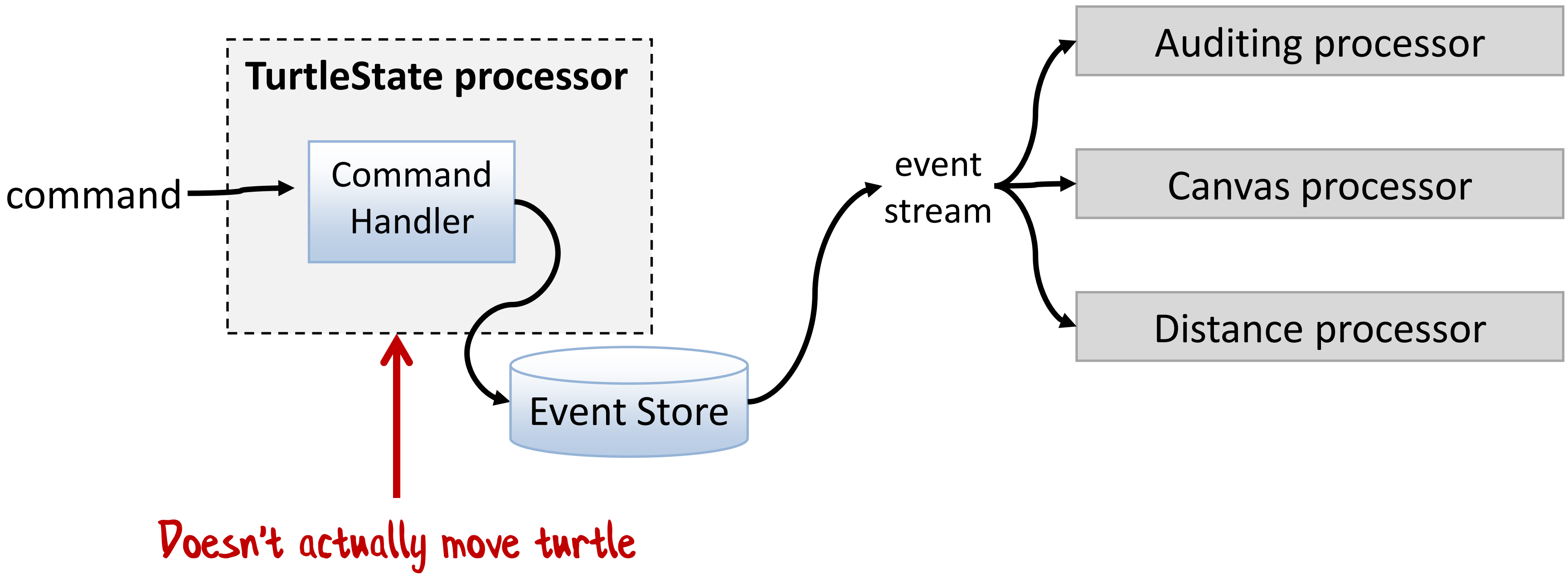
Overview


other event streams
new events
Stream processor
Stream processor
event stream
event stream
event stream
Select, Filter Transform
Command Handler
commands
Event Store
event stream
Stream processor
other event streams
new events
And so on

Turtle stream processing example



Turtle stream processing example



Stream processing demo

Auditing Processor

Canvas Processor

Distance Travelled Processor

Advantages and disadvantages

- Advantages
  - Same as Event Sourcing, plus
  - Separates state management from actions
  - Microservice friendly!
- Disadvantages
  - Even more complex!

We're not done making things complex yet! ☺

Review of turtles 6-8

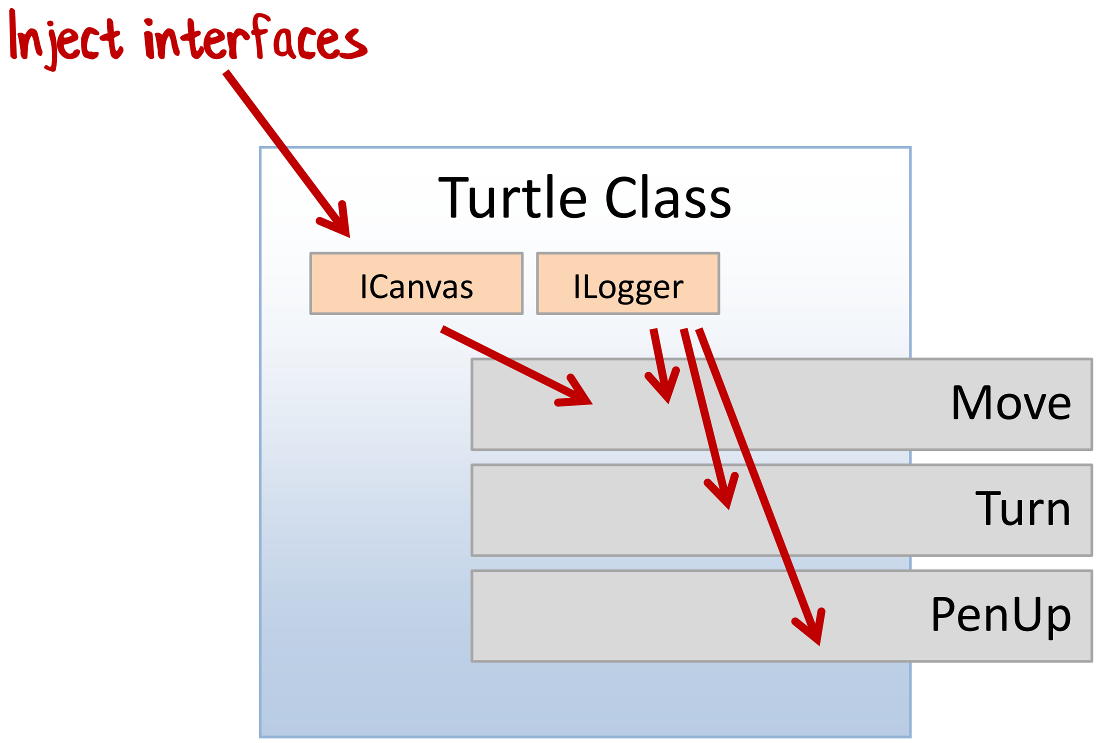
- "Conscious decoupling"
  – Passing data instead of calling functions
- Immutable data stores
  – *Append* to event history rather than mutating a database record

Dependent turtles

9. Dependency injection
10. Dependency parameterization
11. Dependency rejection

9. Dependency injection

Overview

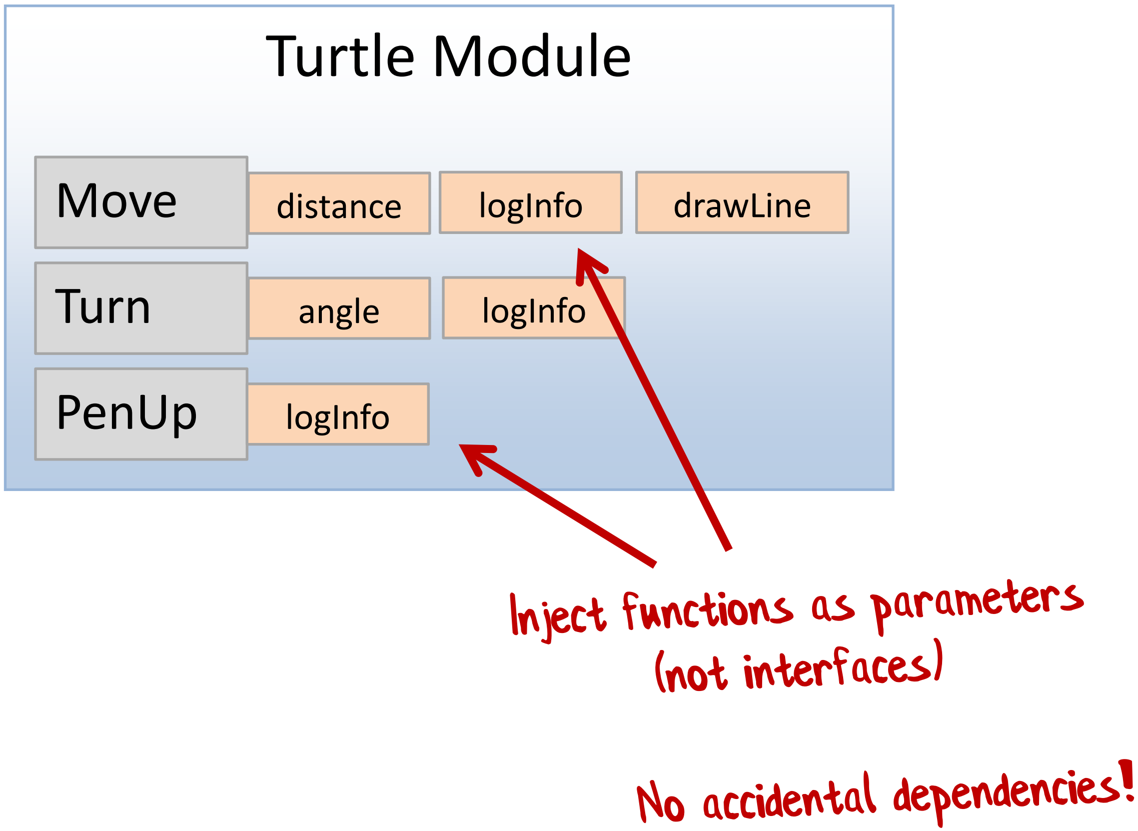

Inject interfaces
Turtle Class
ICanvas
ILogger
Move
Turn
PenUp

Advantages and disadvantages

- Advantages
  - Well understood
  - Constructor injection used by many frameworks
- Disadvantages
  - Interfaces often not fine grained (hence ISP, SRP)
  - Unintentional dependencies & coupling
  - Often requires IoC container or similar

10. Dependency parameterization

Overview



Turtle Module
Move
distance
logInfo
drawLine
Turn
angle
logInfo
PenUp
logInfo
Inject functions as parameters
(not interfaces)
No accidental dependencies!

Parameterization in practice

```
let move = Turtle.move Logger.info Canvas.drawLine
let turn = Turtle.turn Logger.info
```

partial application

```
Turtle.initialTurtleState
|> move 50.0
|> turn 120.0
|> move 50.0
|> turn 120.0
|> move 50.0
|> turn 120.0
```

Use new functions here

Demo:
Dependency parameterization

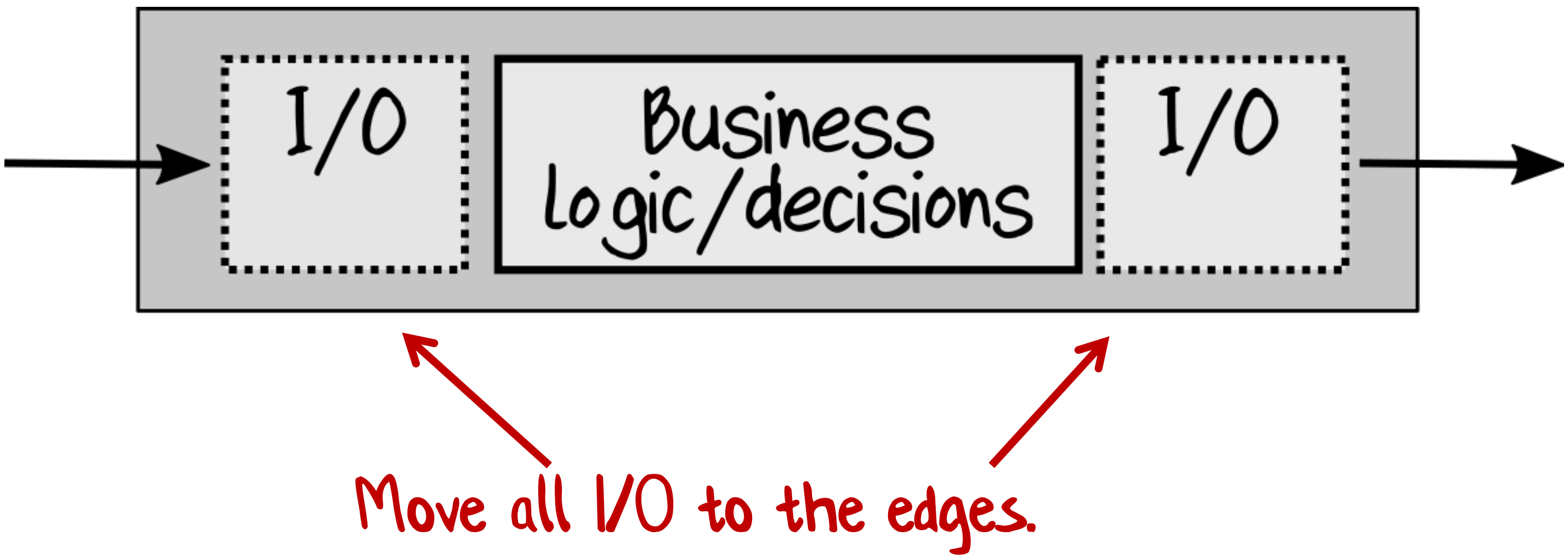
Advantages and disadvantages

- Advantages
  - Dependencies are explicit
  - Functions, not interfaces
  - Counterforce to having too many dependencies (ISP for free!)
  - Built in! No special libraries needed
- Disadvantages
  - Hard to work with deep nesting
  - Use in conjunction with "dependency rejection"

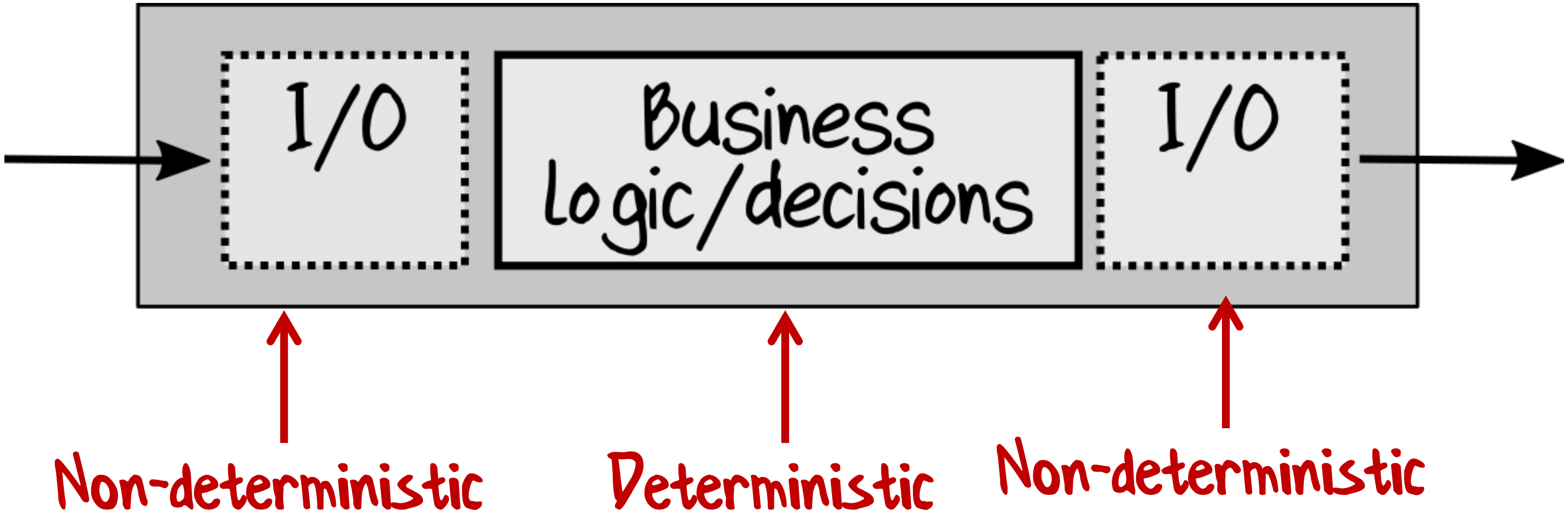
11. Dependency rejection

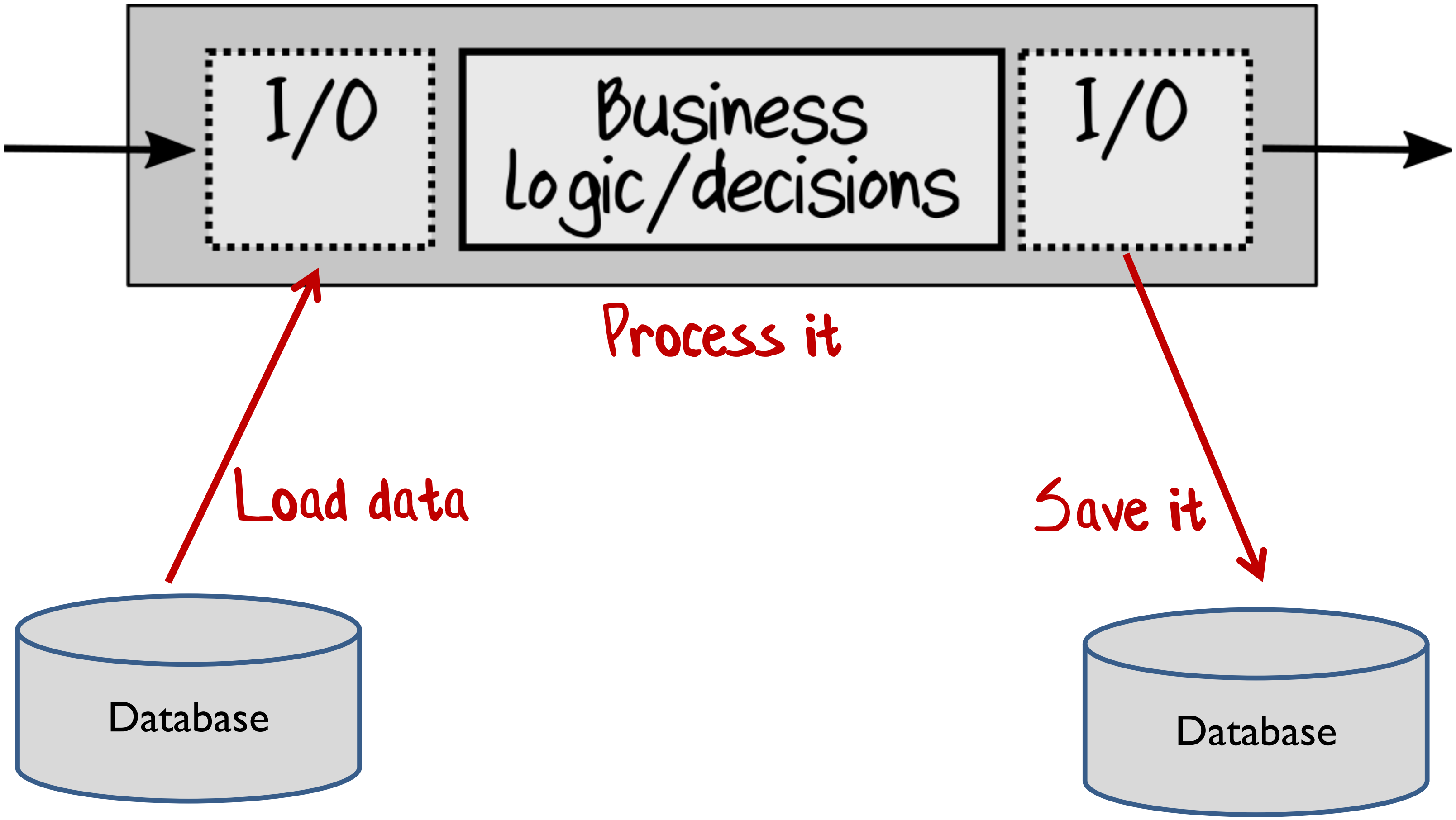
What does "rejection" mean?

Keep dependencies away from business logic

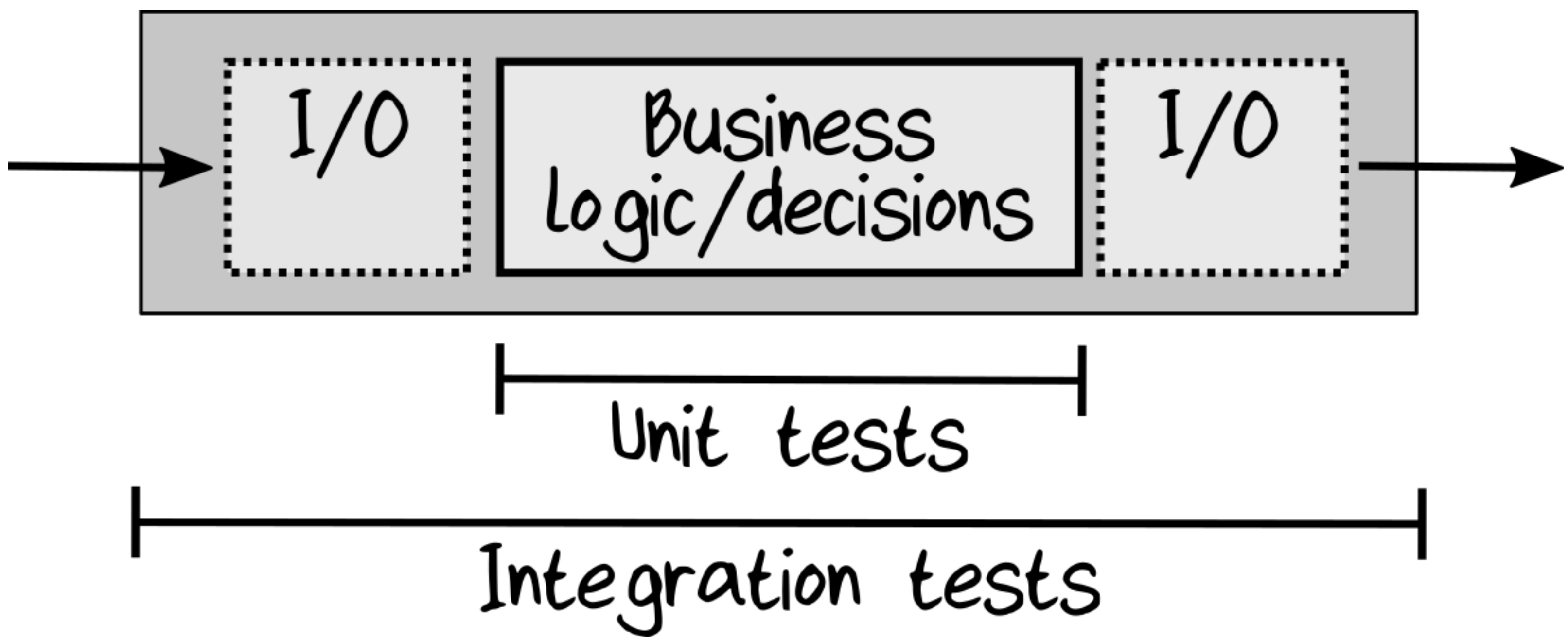


Keep dependencies away from business logic





I/O
Business logic/decisions
I/O
Process it
Load data
Save it
Database
Database


I/O
Business logic/decisions
I/O
Unit tests
Integration tests

Original "move" implementation

```
member this.Move(distance) =
    Logger.info (sprintf "Move %0.1f" distance)

    let newPos = calcNewPosition(distance,angle,position)

    if penState = Down then
        Canvas.drawLine(position,newPos)

    // update the state
    position <- newPos
```

I/O

Deterministic code

Decisions and I/O are interleaved ×

The pure "decision"

```
member this.MoveDecision(distance) =
   let logMessage = sprintf "Move %0.1f" distance

   let newPos = calcNewPosition(distance,angle,position)

   let drawDecision =
      if penState = Down then
          Some(position,newPos)
      else
          None

   position <- newPos
   {logMessage=logMessage; draw=drawDecision}
```

I/O replaced with decisions

Return the decisions

Completely deterministic. No dependencies ✓

The impure wrapper with I/O

```
member this.Move(distance) =
    // decision
    let decision = this.MoveDecision(distance)

    // I/O stuff
    Logger.info decision.logMessage
    match decision.draw with
    | Some (position,newPos) ->
        Canvas.drawLine(position,newPos)
    | None ->
        () // do nothing
```

I/O

Decision

Demo:
Dependency rejection

12. Interpreter

APIs create coupling

```
module TurtleAPI =
  move : Distance * (current)State -> Distance * (new)State
  turn : Angle * (current)State -> (new)State
  penUp : (current)State -> (new)State
  penDown : (current)State -> (new)State
```

Fine, but what if the API changes to return Result?

APIs create coupling

```
module TurtleAPI =
  move : Distance * State -> Result<Distance * State>
  turn : Angle -> State -> Result<State>
  penUp : State -> Result<State>
  penDown : State -> Result<State>
```

Fine, but what if it needs to be Async as well?

APIs create coupling

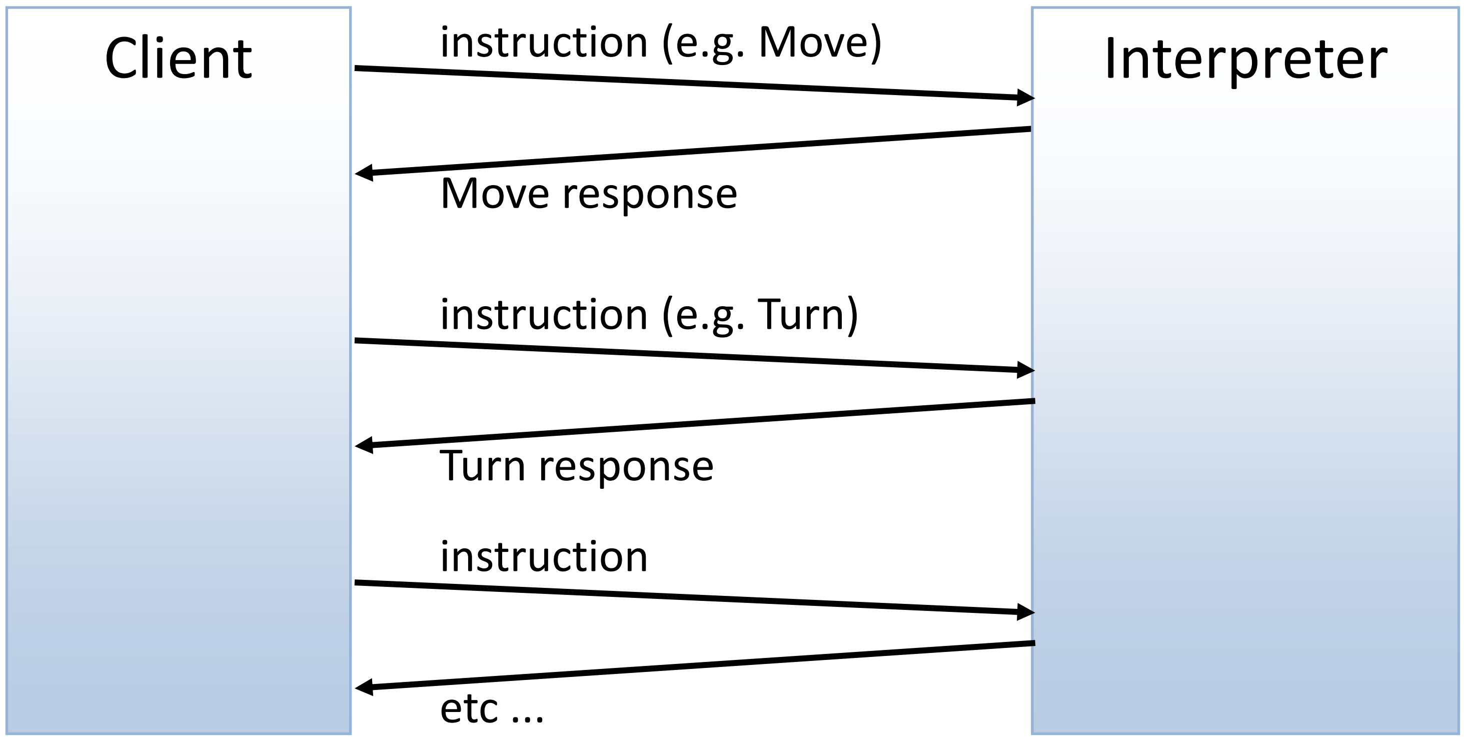
```
module TurtleAPI =
  move : Distance * State -> AsyncResult<Distance * State>
  turn : Angle -> State -> AsyncResult<State>
  penUp : State -> AsyncResult<State>
  penDown : State -> AsyncResult<State>
```

Each change breaks the caller ☹

Solution: decouple using data instead of functions!

We saw how to do this with batch/actors...
But how to manage control flow?

Overview



Client
Interpreter
instruction (e.g. Move)
Move response
instruction (e.g. Turn)
Turn response
instruction
etc ...

Create a set of instructions

```
type TurtleProgram =           // [-------callback----------]
    //          (input params)   (response)    (next step)
    | Move      of Distance    * (Distance  -> TurtleProgram)
    | Turn      of Angle       * (unit      -> TurtleProgram)
    | PenUp     of (* none *)    (unit      -> TurtleProgram)
    | PenDown   of (* none *)    (unit      -> TurtleProgram)
    | Stop
```

Input

Output from interpreter

Next step for the interpreter to run

New case needed!

Usage example

```
let drawTriangle =
  Move (100.0, fun actualDistA ->
               // ^ response from interpreter
  Turn (120.0, fun () ->
               // ^ is response from interpreter
  Move (100.0, fun actualDistB ->
  Turn (120.0, fun () ->
  Move (100.0, fun actualDistC ->
  Turn (120.0, fun () ->
  Stop))))))
```

Ugly!

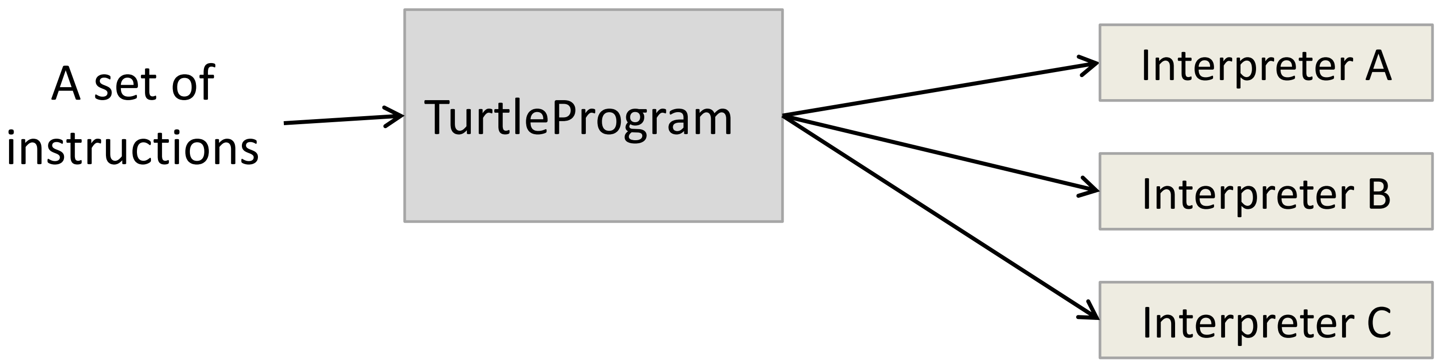
Can we hide the continuations behind the scenes?

Yes, we can!

Usage example

```
let drawTriangle = turtleProgram {// We've seen this before!
    let! actualDistA = move 100.0
    do! turn 120.0
    let! actualDistB = move 100.0
    do! turn 120.0
    let! actualDistC = move 100.0
    do! turn 120.0
    }
```

Overview


A set of instructions
TurtleProgram
Interpreter A
Interpreter B
Interpreter C

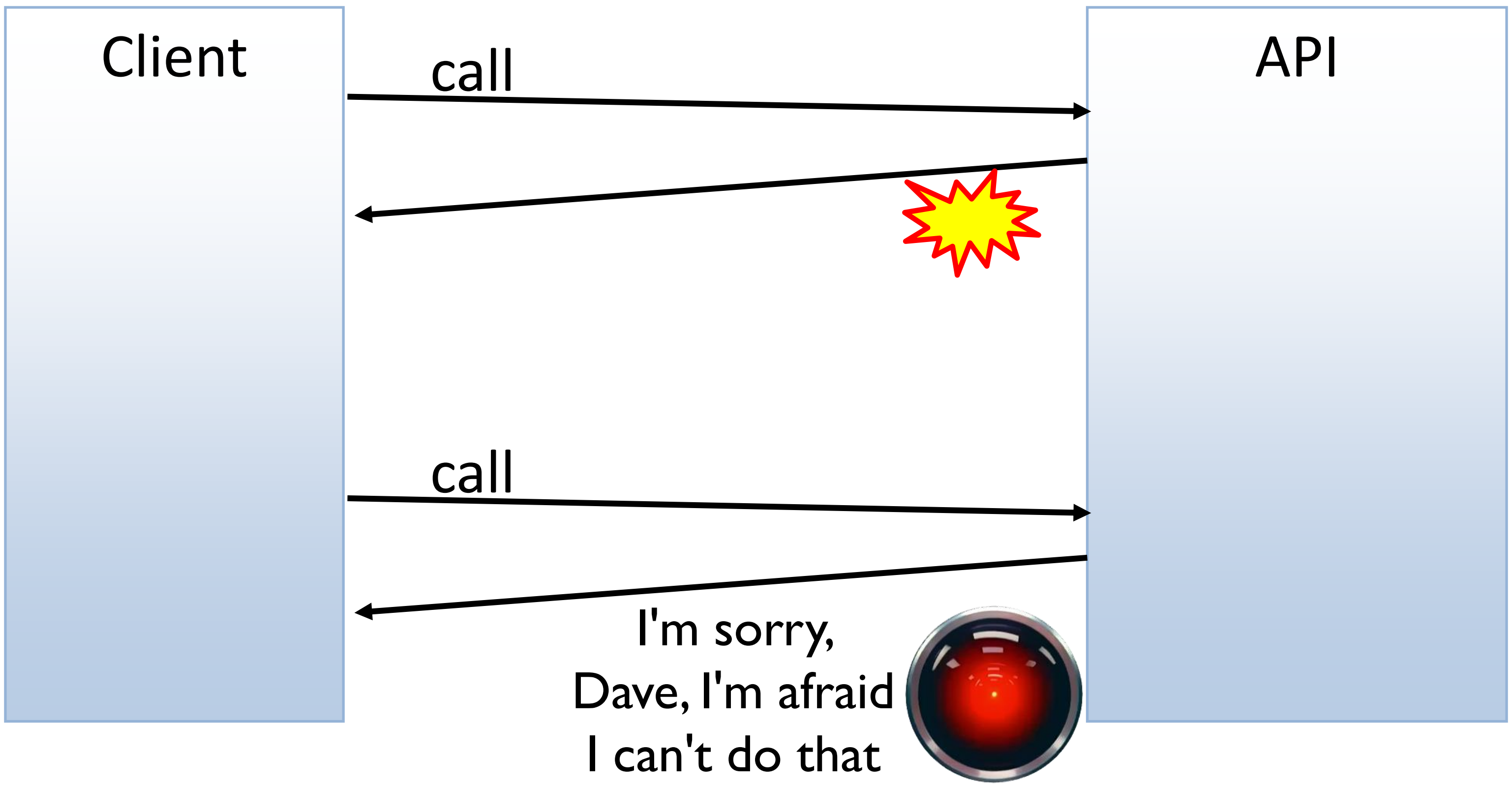
Interpreter demo

Advantages and disadvantages

- Advantages
  - Completely decoupled
  - Pure API
  - Optimization possible
- Disadvantages
  - Complex
  - Best with limited set of operations

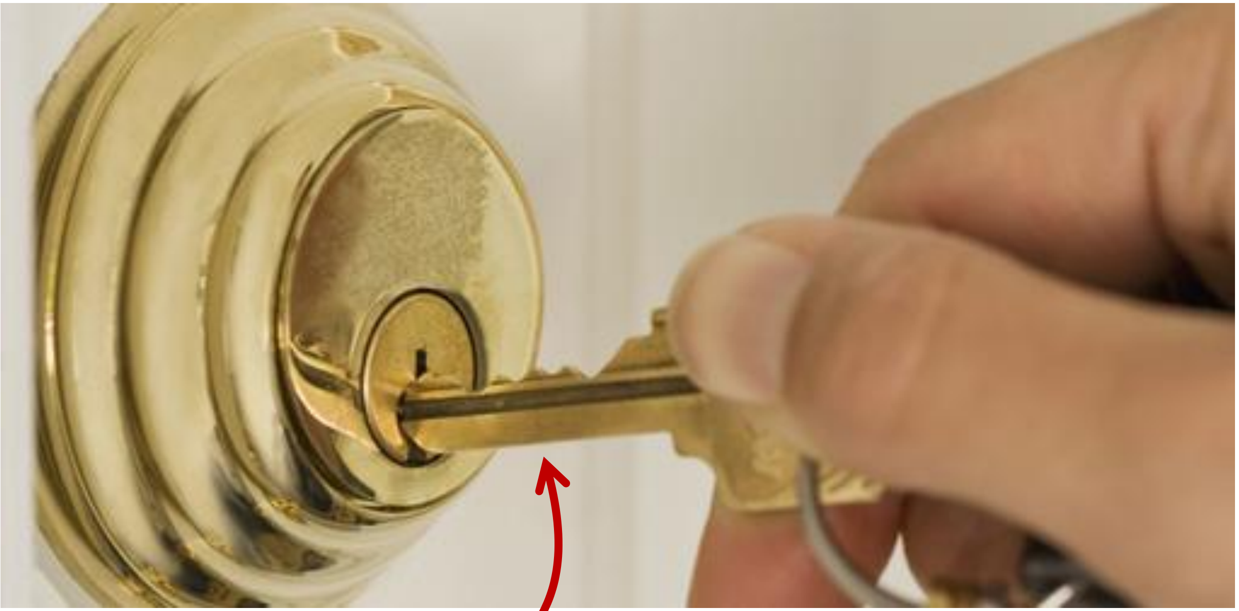
13. Capabilities

Overview



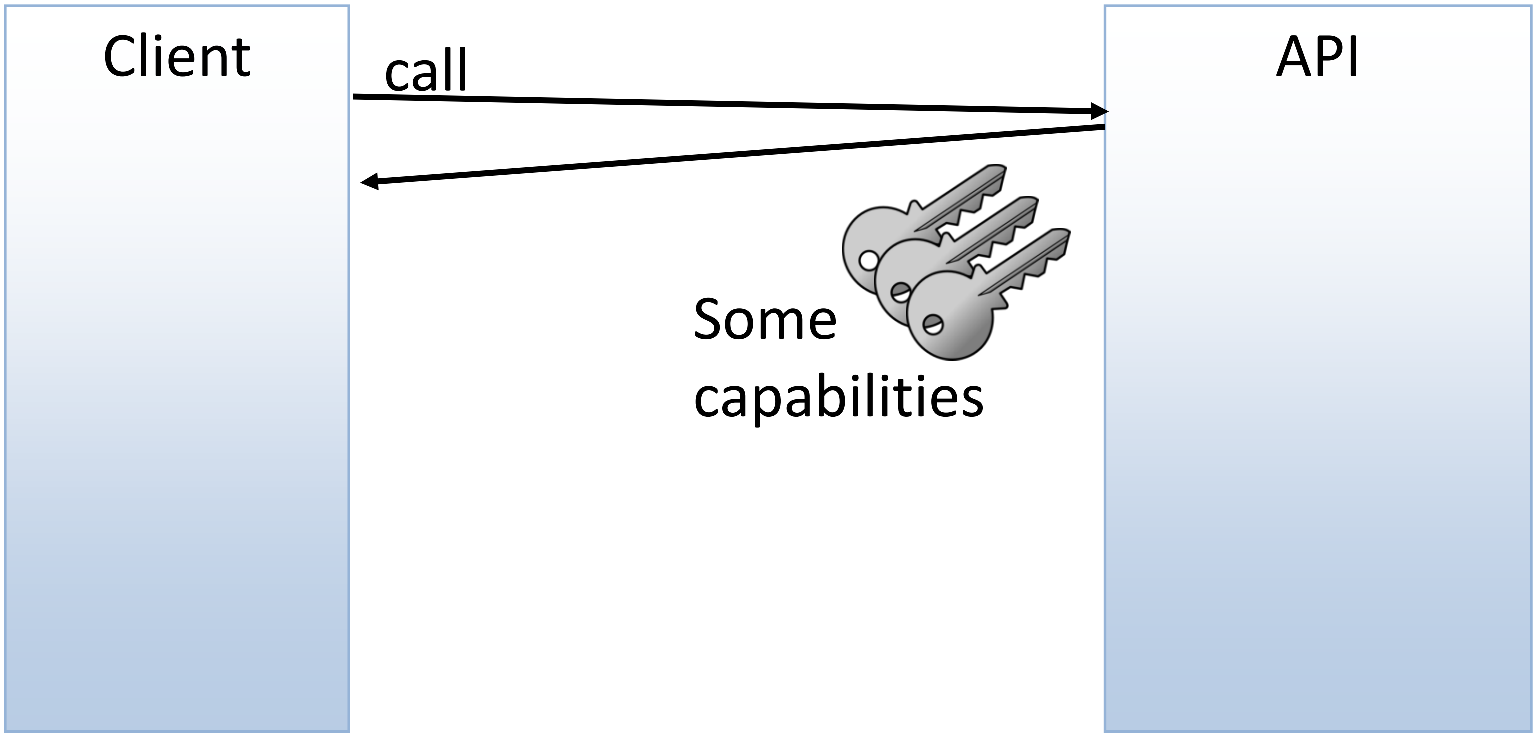
Rather than telling me what I can't do,
why not tell me what I can do?

Capability-based API

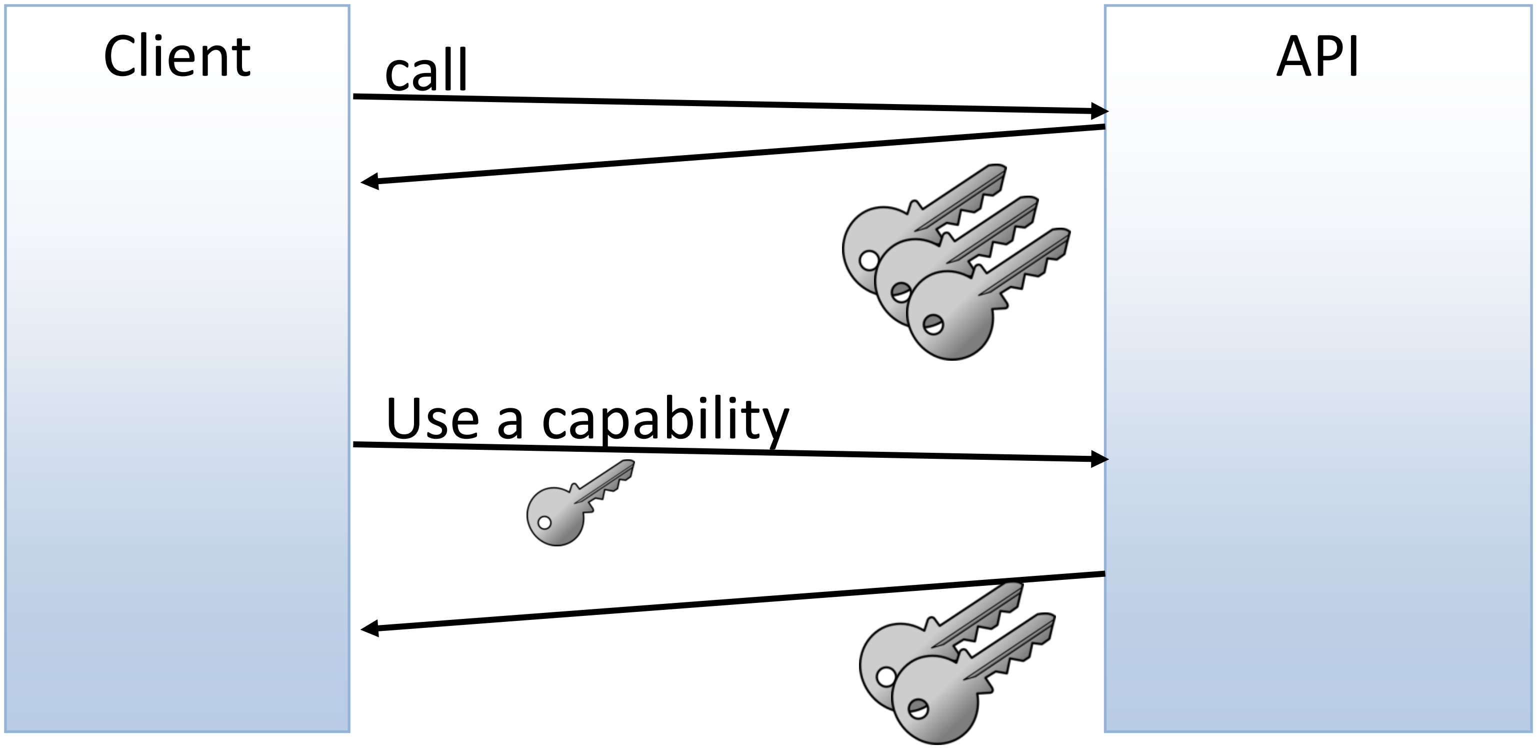


A capability

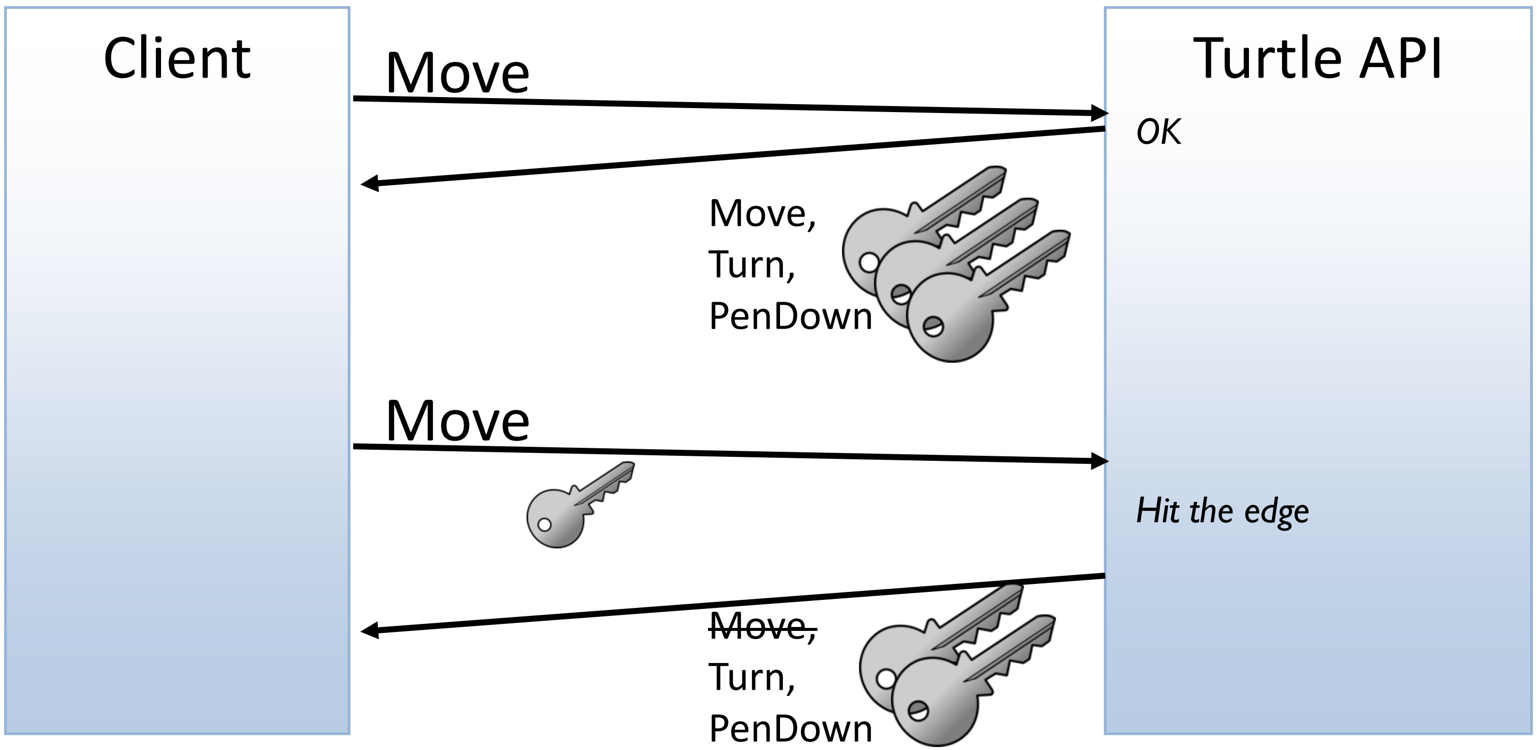
Overview



Overview



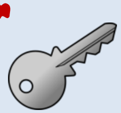
Turtle API capabilities



Turtle Capabilities

```
type TurtleCapabilities = {
    move    : MoveFn option
    turn    : TurnFn
    penUp   : PenUpDownFn
    penDown: PenUpDownFn
    }
and MoveFn =  Distance -> TurtleCapabilities
and TurnFn =     Angle -> TurtleCapabilities
and PenUpDownFn = unit -> TurtleCapabilities
```

All the "Keys"
to the turtle



Capabilities demo

Advantages and disadvantages

- Advantages
  - Client doesn't need to duplicate business logic
  - Better security
  - Capabilities can be transformed for business rules
- Disadvantages
  - Complex to implement
  - Client has to handle unavailable functionality

Example of cap-based design: HATEOAS

Hypermedia as the Engine of Application State

Turtle HATEOAS

```
[
{ "rel":“Move",
  "href": "/turtle/ec03def5-7ea8-4ac3-baf7-b290582cd3f2" },
{ "rel":“Turn",
  "href": "/turtle/d4532ca0-4e61-4fae-bbb1-fc11d4e173df" },
{ "rel":“PenUp",
  "href": "/turtle/fe1bfa98-e77b-4331-b99b-22850d35d39e" }
...
]
```

capabilities

Phew!

github.com/swlaschin/turtle

Slides and code here

Thanks!