Typed Actors

Ryan Bautista

CSCI 49380, Fall 2020

Brief Review of Actors

- Lightweight, event-driven processes

Brief Review of Actors

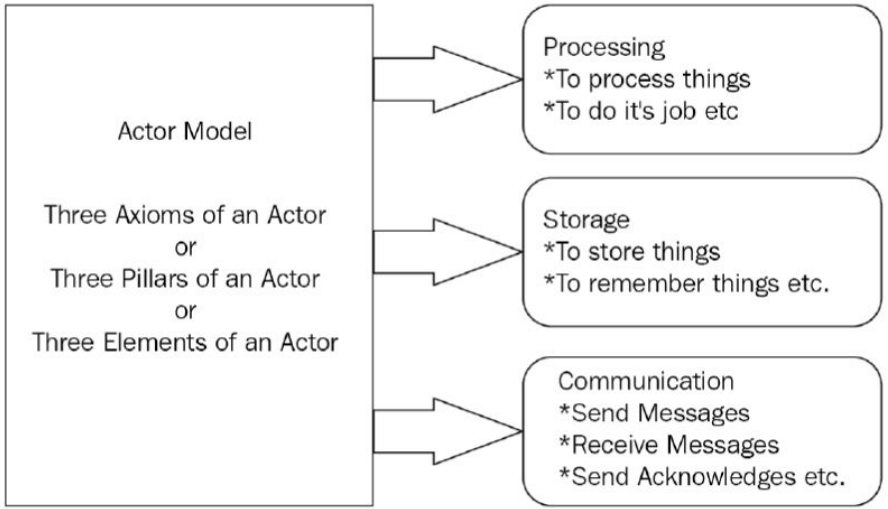
- Lightweight, event-driven processes
- Defined by having both a “state” and a “behavior”.

Brief Review of Actors

- Lightweight, event-driven processes
- Defined by having both a “state” and a “behavior”.
- Self-modifiable only

Brief Review of Actors

- Lightweight, event-driven processes
- Defined by having both a “state” and a “behavior”.
- Self-modifiable only
- The three axioms of an actor:
    1. Processing
    2. Storage
    3. Communication



Three Axioms of Actor

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the message in its message queue

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?
- Typed Actors can send and receive typed messages

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?
- Typed Actors can send and receive typed messages
- Geared towards interacting with non-Actors

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?
- Typed Actors can send and receive typed messages
- Geared towards interacting with non-Actors
- Not meant for use over Untyped Actors due to its limitations and its position as the gate between Actors and non-Actors

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?
- Typed Actors can send and receive typed messages
- Geared towards interacting with non-Actors
- Not meant for use over Untyped Actors due to its limitations and its position as the gate between Actors and non-Actors
  - Cannot use certain methods like *become/unbecome*

Typed Actors

- **Typed Actors:** Actors that interact differently, depending on the protocol in its message queue
  - Example: What is today's date? What time is it right now?
- Typed Actors can send and receive typed messages
- Geared towards interacting with non-Actors
- Not meant for use over Untyped Actors due to its limitations and its position as the gate between Actors and non-Actors
  - Cannot use certain methods like *become/unbecome*
  - Untyped Actors should not interact with non-Actors

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
    - ActorRef must have a typed parameter

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
  - ActorRef must have a typed parameter
  - The message must come with /tell of the given type, and ONLY that

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
  - ActorRef must have a typed parameter
  - The message must come with /tell of the given type, and ONLY that
  - There must be no context.sender

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
  - ActorRef must have a typed parameter
  - The message must come with /tell of the given type, and ONLY that
  - There must be no context.sender
- For the receiving Typed Actor:

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
  - ActorRef must have a typed parameter
  - The message must come with /tell of the given type, and ONLY that
  - There must be no context.sender
- For the receiving Typed Actor:
  - The receiving Actor must have a trait that understands the message being sent

ActorRef and Adding Types to Actors

- **ActorRef:** The reference of an Actor
- ActorRef must be restricted in order to send a proper message
- For the sending Typed Actor:
  - ActorRef must have a typed parameter
  - The message must come with /tell of the given type, and ONLY that
  - There must be no context.sender
- For the receiving Typed Actor:
  - The receiving Actor must have a trait that understands the message being sent
  - The receiving Actor must have the proper ActorContext, or have the appropriate *self* reference

Ping-Pong System with Untyped Actors

```
class Pinger extends Actor {
  var pongsLeft = 4

  def receive = {
    case Pong ⇒
      println(s"Pinger received Pong. $pongsLeft Pongs left to receive")

      if (pongsLeft > 0) {
        pongsLeft -= 1
        sender() ! Ping
      } else {
        sender() ! PoisonPill
        self ! PoisonPill
      }
  }
}

class Ponger(pinger: ActorRef) extends Actor {
  def receive = {
    case Ping ⇒
      println(s"Ponger received ping")
      pinger ! Pong
  }
}
```

Squaring System with Typed Actors

Interface

```
trait SquaringInterface {
  def squareFuture(i: Int): Future[Int]

  def squareOption(i: Int): Option[Int]

  def squareNormal(i: Int): Int

  def squareWords(i: Int): String

  @throws(classOf[Exception])
  def squareTry(i: Int): Int
}
```

Implementation

```
class SquaringImplement extends SquaringInterface {

  def squareFuture(i: Int): Future[Int] = Future.successful(i * i)

  def squareOption(i: Int): Option[Int] = Some(i * i)

  def squareNormal(i: Int): Int = i * i

  def squareWords(i: Int): String = i.toString + " squared"

  def squareTry(i: Int): Int = throw new Exception("SquareException")
}

val SquaringActor: SquaringInterface = TypedActor(system).typedActorOf(TypedProps[SquaringImplement]())
val optionSquare = SquaringActor.squareOption(10)
val futureSquare = SquaringActor.squareFuture(10)
```

Testing Typed Actors

- Testing individual actors in general is difficult
  - Asynchronization introduces timeouts and prevents normal synchronous behavior

Testing Typed Actors

- Testing individual actors in general is difficult
  - Asynchronization introduces timeouts and prevents normal synchronous behavior
  - Results may not be non-deterministic

Testing Typed Actors

- Testing individual actors in general is difficult
  - Asynchronization introduces timeouts and prevents normal synchronous behavior
  - Results may not be non-deterministic
  - Typed messages add to the confusion
- Testing the *behaviors* of actors is far easier

Testing Typed Actors

- Testing individual actors in general is difficult
  - Asynchronization introduces timeouts and prevents normal synchronous behavior
  - Results may not be non-deterministic
  - Typed messages add to the confusion
- Testing the *behaviors* of actors is far easier
  - Testing individual functions, effects of the actors, and how it receives messages

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor
  - Setting/Receiving Timeouts

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor
  - Setting/Receiving Timeouts
  - Message Scheduling

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor
  - Setting/Receiving Timeouts
  - Message Scheduling

```
TestKit.ref ! <messageToSend>
TestKit.runOne()
```

- Message injection is exactly the same as normal messaging, but without ActorContext

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor
  - Setting/Receiving Timeouts
  - Message Scheduling

```
TestKit.ref ! <messageToSend>
TestKit.runOne()
```

- Message injection is exactly the same as normal messaging, but without ActorContext
- *RetrieveAllEffects()*

```
TestKit.retrieveAllEffects()
```

BehaviorTestKit

```
val TestKit = BehaviorTestKit(<actor>)
```

- *BehaviorTestKit*: allows for testing behaviors via message injection and provocation
  - Child Actors (creation and deletion)
  - Termination or Death of the Actor
  - Setting/Receiving Timeouts
  - Message Scheduling

```
TestKit.ref ! <messageToSend>
TestKit.runOne()
```

- Message injection is exactly the same as normal messaging, but without ActorContext
- *RetrieveAllEffects()*
- *BehaviorTestKit* is observation only

```
TestKit.retrieveAllEffects()
```

Testing Mailboxes

- *BehaviorTestKit* is centralized around the selected Typed Actor only, not its messages

Testing Mailboxes

- *BehaviorTestKit* is centralized around the selected Typed Actor only, not its messages
- *TestInbox* provides a place for messages accrued through *BehaviorTestKit*

```
val Inbox = TestInbox[ActorRef[<Actor>]]()
```

Testing Mailboxes

- *BehaviorTestKit* is centralized around the selected Typed Actor only, not its messages
- *TestInbox* provides a place for messages accrued through *BehaviorTestKit*
- *TestInbox.receiveMessage()* outputs the message in the Inbox

```
val Inbox = TestInbox[ActorRef[<Actor>]]()
```

```
TestKit.ref ! NewGreeter(Inbox.ref)
TestKit.runOne()
val greeterRef = Inbox.receiveMessage()
```

Testing Mailboxes

- *BehaviorTestKit* is centralized around the selected Typed Actor only, not its messages
- *TestInbox* provides a place for messages accrued through *BehaviorTestKit*
- *TestInbox.receiveMessage()* outputs the message in the Inbox
- Observable only

```
val Inbox = TestInbox[ActorRef[<Actor>]]()
```

```
TestKit.ref ! NewGreeter(Inbox.ref)
TestKit.runOne()
val greeterRef = Inbox.receiveMessage()
```

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination
- Certain commands may yield events, which are stored in the journal

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination
- Certain commands may yield events, which are stored in the journal
    - These events make the Typed Actor go through state changes

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination
- Certain commands may yield events, which are stored in the journal
  - These events make the Typed Actor go through state changes
- This leads the way to two type-safe expressions

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination
- Certain commands may yield events, which are stored in the journal
  - These events make the Typed Actor go through state changes
- This leads the way to two type-safe expressions
  1. One type-safe function changes the given commands into events

Typed Persistence

- Persistent Typed Actors can take note of changes that occur, even through a reboot or other forms of termination
- Certain commands may yield events, which are stored in the journal
  - These events make the Typed Actor go through state changes
- This leads the way to two type-safe expressions
  1. One type-safe function changes the given commands into events
  2. One type-safe function interprets the events into how the state changes

Supervision

- Sometimes, actors will simply fail for whatever reason

Supervision

- Sometimes, actors will simply fail for whatever reason
- A Supervisor may come into to deal with the failed Actor

Supervision

- Sometimes, actors will simply fail for whatever reason
- A Supervisor may come into to deal with the failed Actor
- However, a Typed Actor is far easier to deal with than a failed Untyped actor

Supervision

- Sometimes, actors will simply fail for whatever reason
- A Supervisor may come into to deal with the failed Actor
- However, a Typed Actor is far easier to deal with than a failed Untyped actor
    - Failure pauses the Actor process, which is more problematic for an Untyped actor, and removes predictability

Supervision

- Sometimes, actors will simply fail for whatever reason
- A Supervising Actor may come into to deal with the failed Actor
- However, a Typed Actor is far easier to deal with than a failed Untyped actor
    - Failure pauses the Actor process, which is more problematic for an Untyped actor, and removes predictability
    - Failure notices contain a lot of information for an Untyped Actor, slowing things down further

A Supervisor Typed Actor

- The context function used to declare supervision on child actors

```
ctx.spawnAnonymous(Behaviors.supervise(actor).onFailure[ArithmeticException](SupervisorStrategy.restart))
```

A Supervisor Typed Actor

- The context function used to declare supervision on child actors
- The supervised Actor is given the ability to supervise its child actors

```
ctx.spawnAnonymous(Behaviors.supervise(actor).onFailure[ArithmeticException](SupervisorStrategy.restart))
```

A Supervisor Typed Actor

- The context function used to declare supervision on child actors
- The supervised Actor is given the ability to supervise its child actors
- Failure states are confined to the failed actor, preventing the failure from spreading (known as **Akka Typed shields**)

```
ctx.spawnAnonymous(Behaviors.supervise(actor).onFailure[ArithmeticException](SupervisorStrategy.restart))
```

Declaring a Supervisor Actor

```
def supervise[T](behavior: Behavior[T]): Behavior[T] = new Restarter(behavior, behavior)

class Restarter[T](initial: Behavior[T], behavior: Behavior[T]) extends ExtensibleBehavior[T] {
  def receive(ctx: ActorContext[T], msg: T): Behavior[T] = {
    try {
      val started = validateAsInitial(start(behavior, ctx))
      val next = interpretMessage(started, ctx, msg)
      new Restarter(initial, canonicalize(next, started, ctx))
    }
    catch {
      case _: ArithmeticException ⇒ new Restarter(initial, validateAsInitial(start(initial, ctx)))
    }
  }
}
```

Thank you!

Any questions?