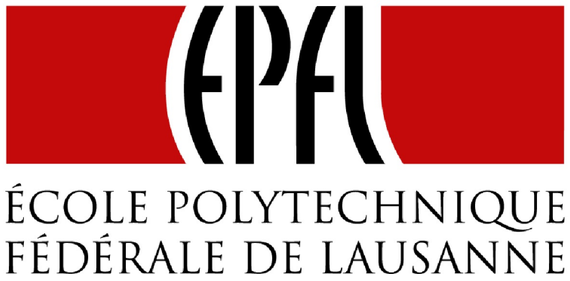


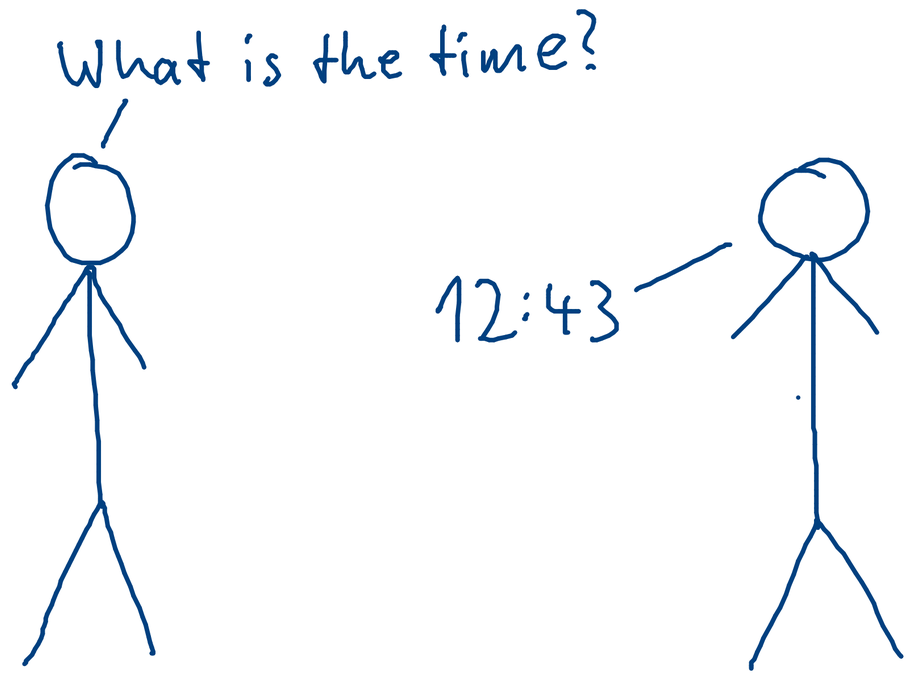
The Actor Model

Principles of Reactive Programming

Roland Kuhn

What is an Actor?

The Actor Model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

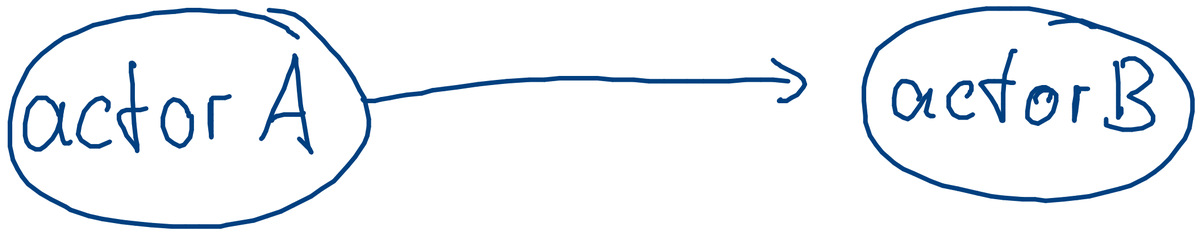


What is an Actor?

The Actor Model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

An Actor[1]

- is an object with identity
- has a behavior
- only interacts using *asynchronous* message passing



[1] Hewitt, Bishop, Steiger: *A Universal Modular Actor Formalism for Artificial Intelligence, IJCAI 1973*

The Actor Trait

```
type Receive = PartialFunction[Any, Unit]

trait Actor {
  def receive: Receive
  ...
}
```

The `Actor` type describes the behavior of an Actor, its response to messages.

A Simple Actor

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
  }
}
```

This object does not exhibit stateful behavior.

Making it Stateful

Actors can send messages to addresses (`ActorRef`) they know:

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
    case ("get", customer: ActorRef) => customer ! count
  }
}
```

How Messages are Sent

```
trait Actor {
  implicit val self: ActorRef
  def sender: ActorRef
  ...
}

abstract class ActorRef {
  def !(msg: Any)(implicit sender: ActorRef = Actor.noSender): Unit
  def tell(msg: Any, sender: ActorRef) = this.!(msg)(sender)
  ...
}
```

Sending a message from one actor to the other picks up the sender's address implicitly.

Using the Sender

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
    case "get"  => sender ! count
  }
}
```

The Actor's Context

The `Actor` type describes the behavior, the execution is done by its `ActorContext`.

```
trait ActorContext {
  def become(behavior: Receive, discardOld: Boolean = true): Unit
  def unbecome(): Unit
  ...
}

trait Actor {
  implicit val context: ActorContext
  ...
}
```

Changing an Actor's Behavior

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case "incr" => context.become(counter(n + 1))
    case "get"  => sender ! n
  }
  def receive = counter(0)
}
```

Changing an Actor's Behavior

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case "incr" => context.become(counter(n + 1))
    case "get"  => sender ! n
  }
  def receive = counter(0)
}
```

Functionally equivalent to previous version, with advantages

- state change is explicit
- state is scoped to current behavior

Similar to "asynchronous tail-recursion".

Creating and Stopping Actors

```
trait ActorContext {
  def actorOf(p: Props, name: String): ActorRef
  def stop(a: ActorRef): Unit
  ...
}
```

Actors are created by actors.

“stop” is often applied to “self”.

An Actor Application

```
class Main extends Actor {
  val counter = context.actorOf(Props[Counter], ”counter”)

  counter ! ”incr”
  counter ! ”incr”
  counter ! ”incr”
  counter ! ”get”

  def receive = {
    case count: Int =>
      println(s”count was $count”)
      context.stop(self)
  }
}
```

The Actor Model of Computation

Upon reception of a message the actor can do any combination of the following:

- send messages
- create actors
- designate the behavior for the next message