

Kinga Gałdzińska

Developing applications using Actor model.
Introduction to Akka.NET

Software engineer at  grupa pracuj

kinga.gazdzinska@pracuj.pl

<https://www.linkedin.com/in/gazdzinskak/>

<https://github.com/gazdzinskak/AkkaNET>



Agenda

why even bother?

Actor model – the idea

thinking in actors

introduction to Akka.NET

use case discussion

+

Overall idea of Actor model

Scenarios where using Akka.NET is worth consideration

Where to start learning Akka.NET

+

Overall idea of Actor model

Scenarios where using Akka .NET is worth consideration

Where to start learning Akka.NET

-

Broad understanding of Akka.NET concepts

Advanced Akka.NET features



PLACE ORDER

Name *

First

Last

Email *

Phone

Foo Product

Price: **\$10.00** Quantity:

Bar Product

Price: **\$20.00** Quantity:

Total

\$0.00

Submit Order

1. Log
2. Check product availability
3. Save order to DB
4. Update product quantity
5. Notify




```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

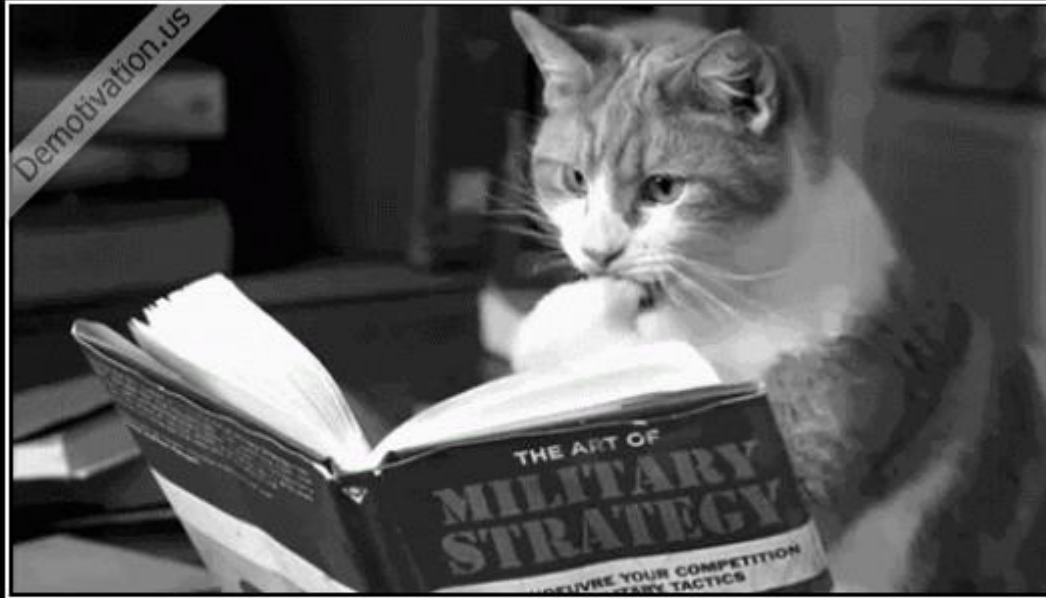
```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

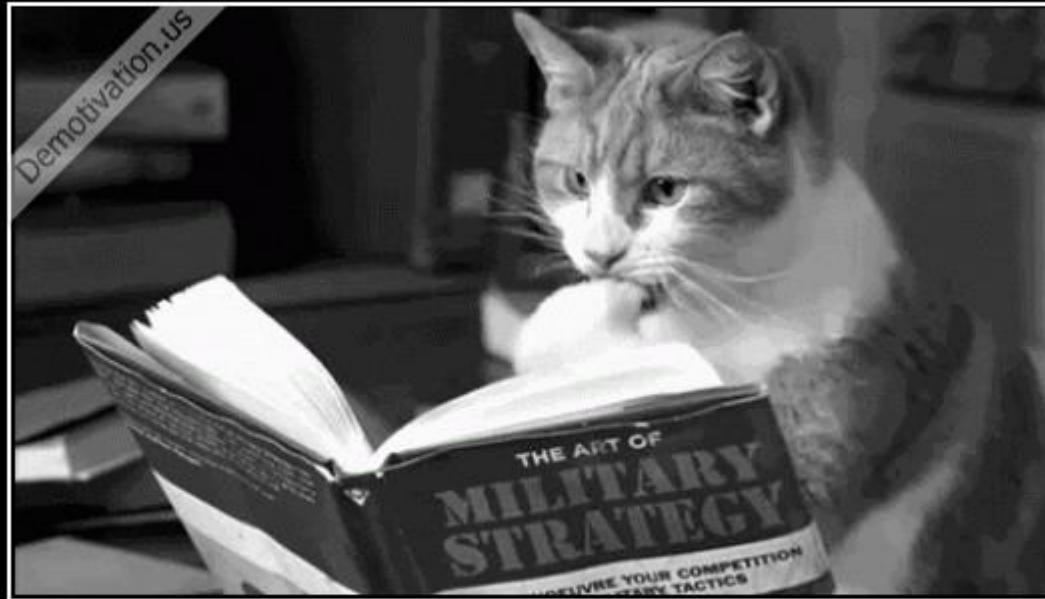
```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```



We are doomed



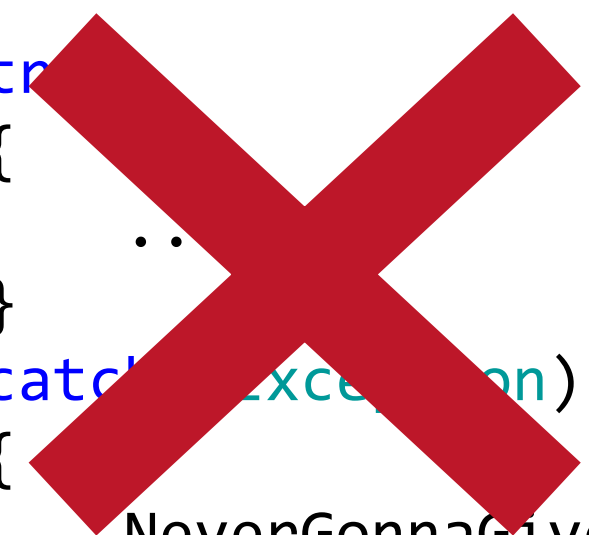
...are we though?

#0 I've got a Great Idea!

```
static void Main(string[] args)
{
    try
    {
        ...
    }
    catch (Exception)
    {
        NeverGonnaGiveYouUp();
    }
}
```

#0 No, you don't.

```
static void Main(string[] args)
{
    try
    {
        ..
    }
    catch (Exception)
    {
        NeverGonnaGiveYouUp();
    }
}
```



- slooow
- it's better dead than corrupting data

#1 Let it burn



#1 Let it burn

-

more time on post mortems than
on new features

+

first feature may be delivered on
time

#2 Handle „expected” errors



#2 Handle „expected” errors

-

code readability
unexpectedable happens ... a lot
lacks adaptability

+

unit testable
relevant decision can be made

#3 Global error handlers



#3 Global error handlers

-

no context
certain air of „automagic”
no self-heal

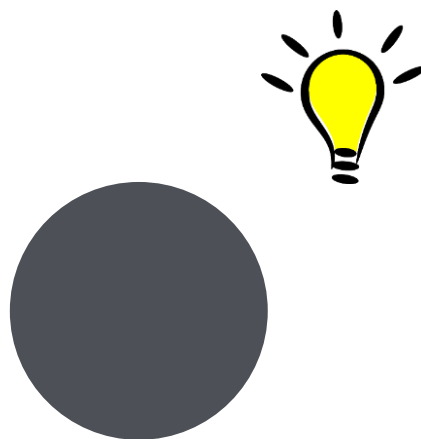
+

testable
readability



Let's step back for a moment...

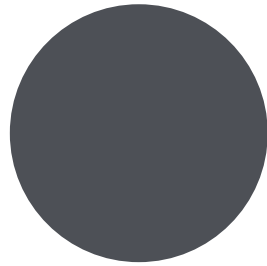
Let's step back for a moment...



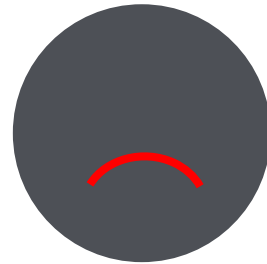
Let's step back for a moment...



Let's step back for a moment...



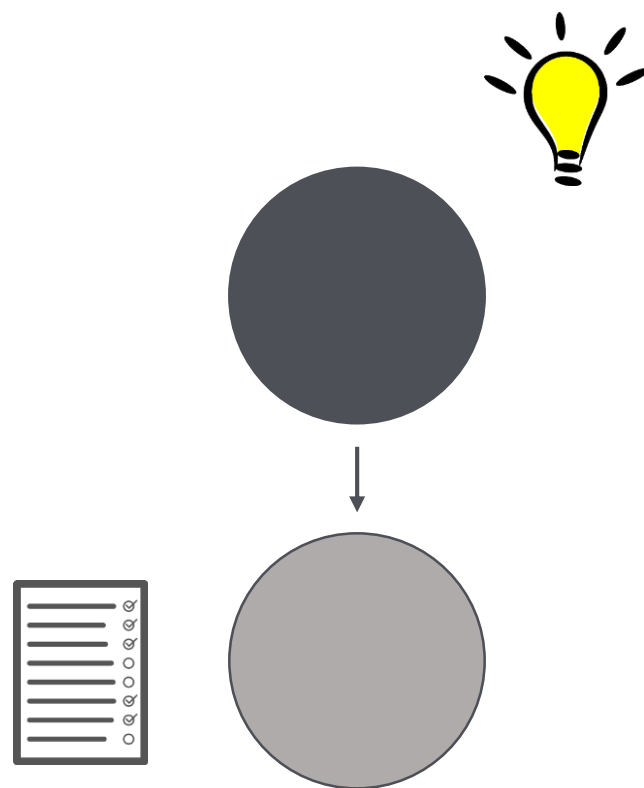
Let's step back for a moment...



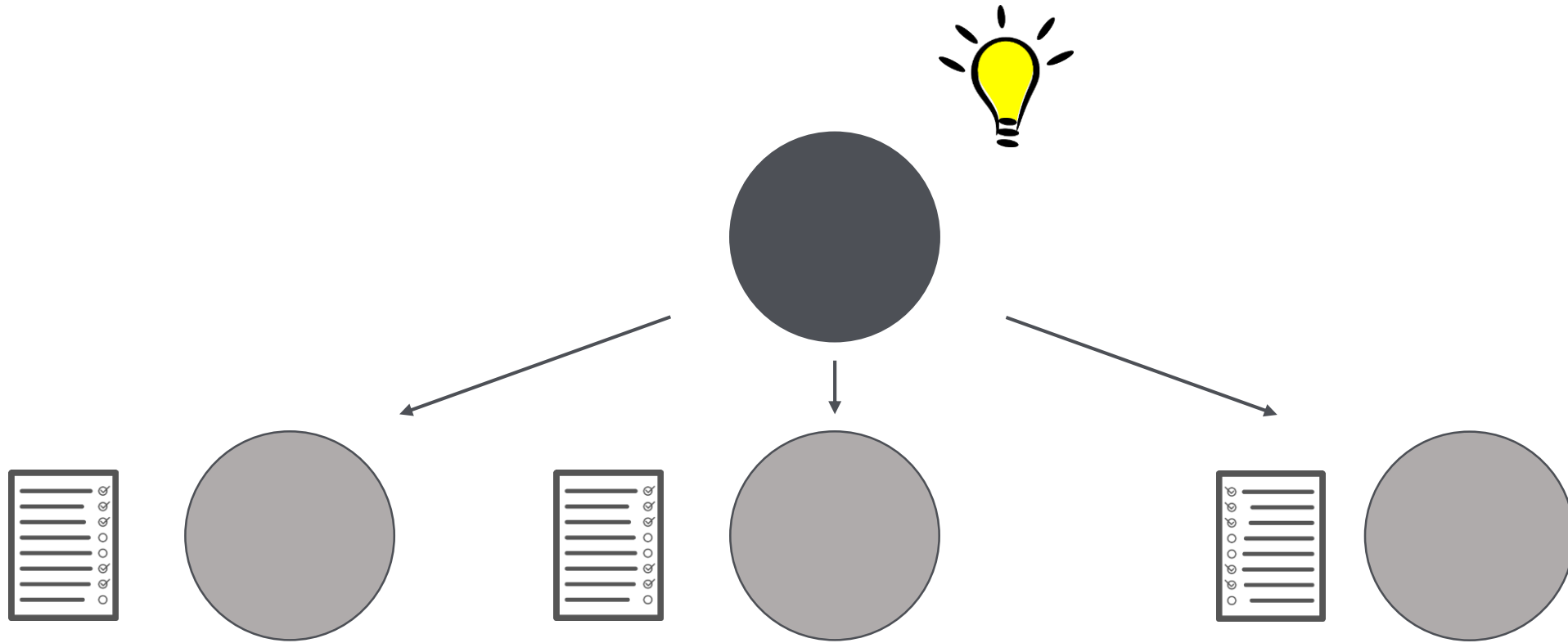
Let's step back for a moment...



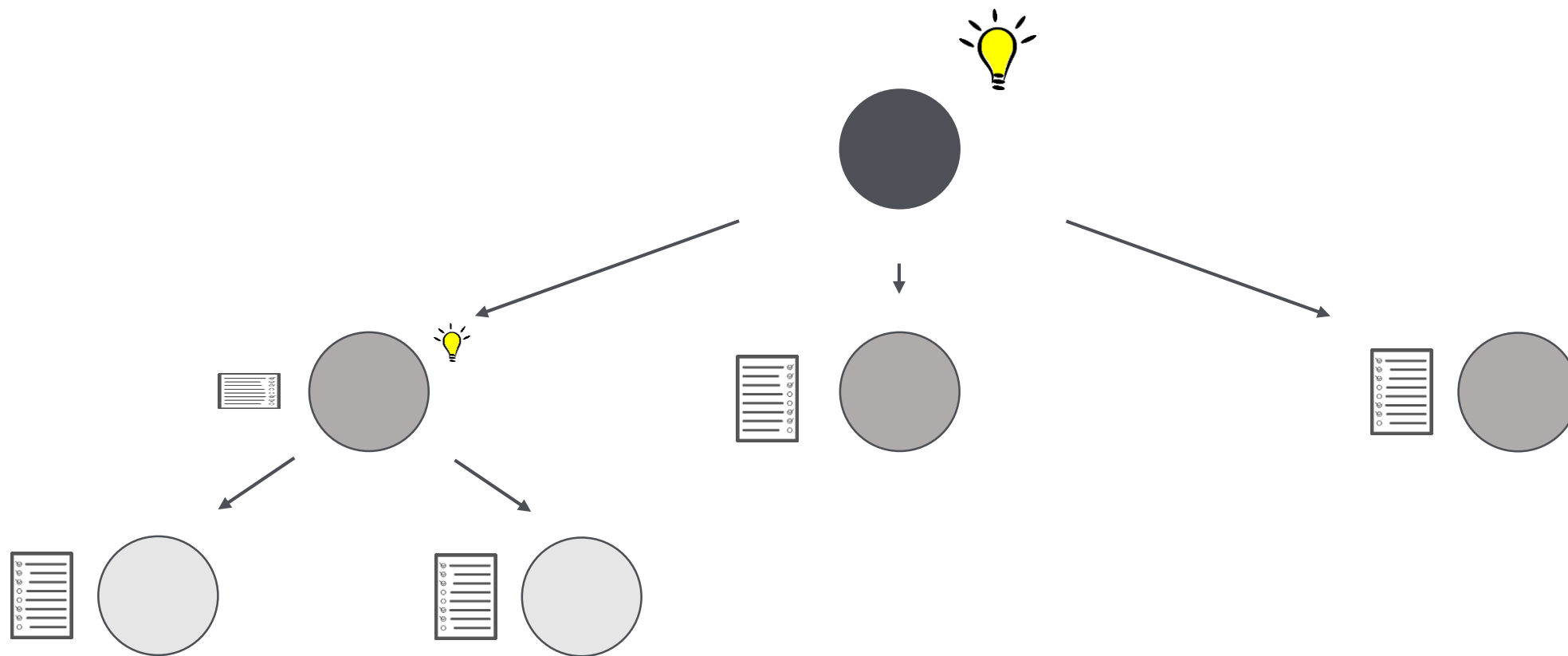
Let's step back for a moment...



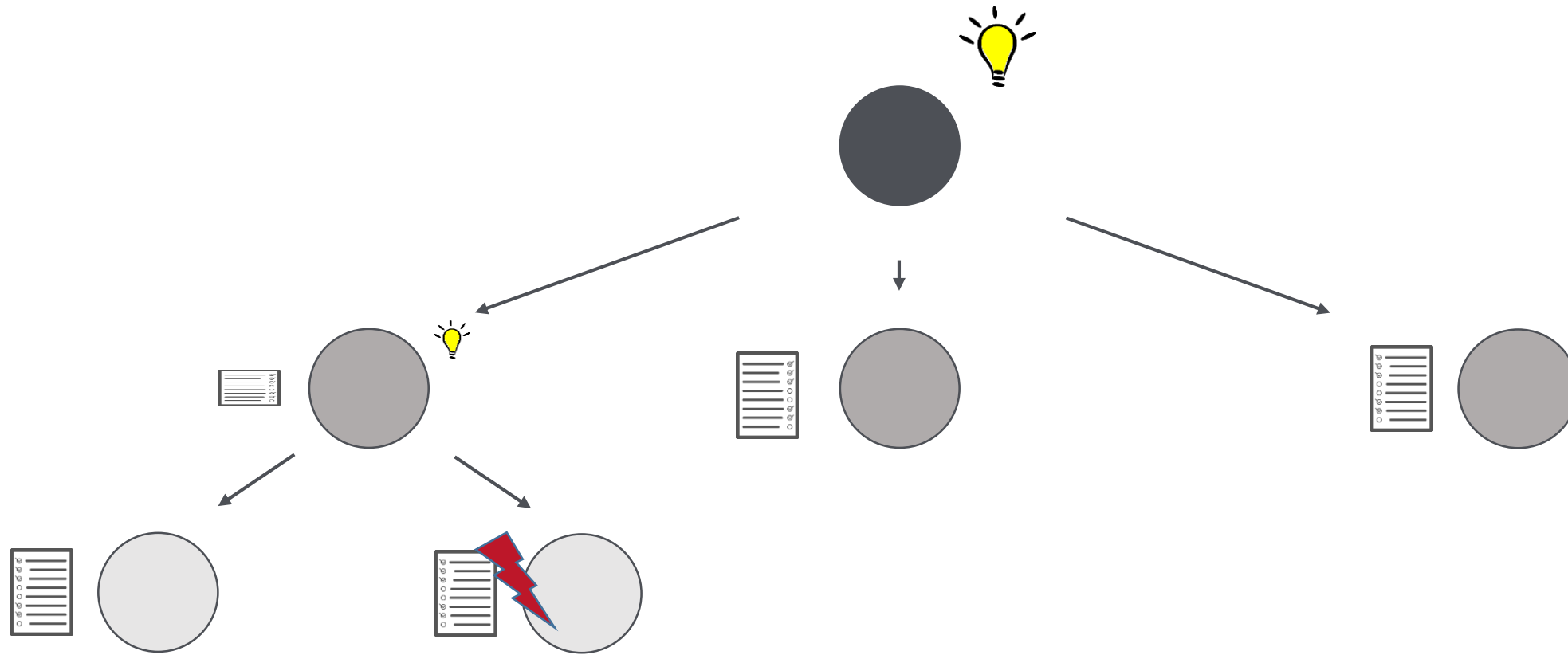
Let's step back for a moment...



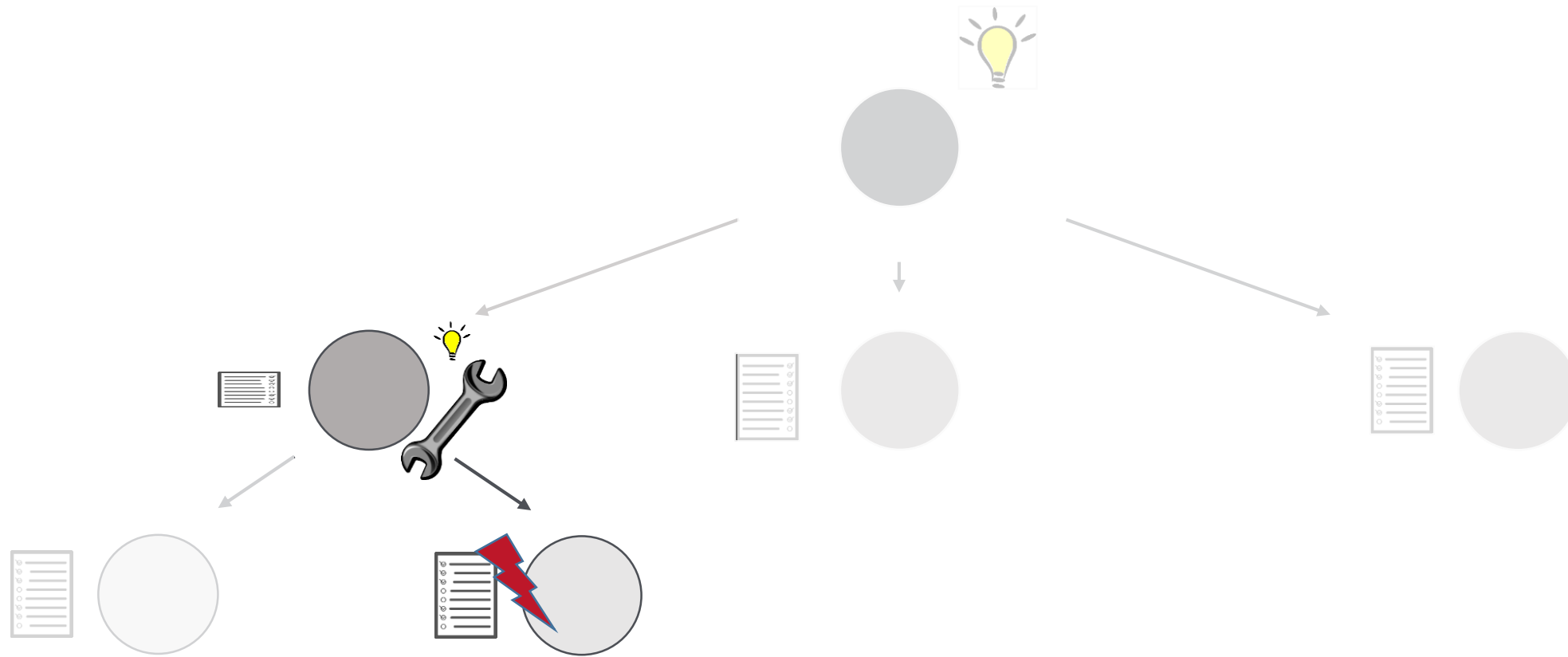
Let's step back for a moment...



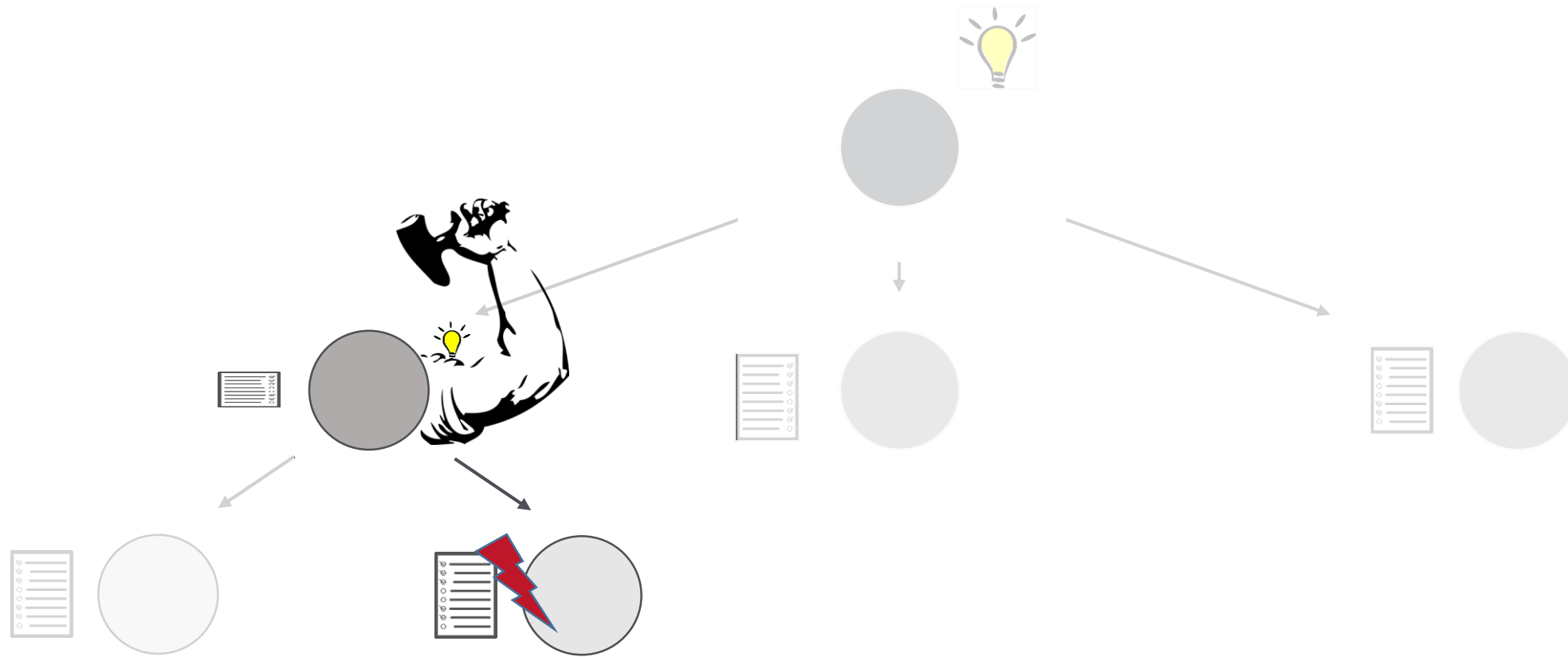
Let's step back for a moment...



Let's step back for a moment...



Let's step back for a moment...



Introducing...

Metro Goldwyn Mayer

TRADE

MARK



Actor model

mathematical model of concurrent computation

concurrent computing ~ quantum and relativistic physics (sic!)

originated in 1973

Carl Hewitt; Peter Bishop; Richard Steiger „*A Universal Modular Actor Formalism for Artificial Intelligence*”

„everything is an actor”

The world is modeled as:

stateful entities

communicating with each other

by explicit **message passing**

In response to message, actor can:

- make local decisions

- create more actors

- send more messages

- determine how to respond to the next message

Actor

may modify **its own** state

can only affect others' **through messages**



Adaptation in practice

(1973)

1986 – Erlang Joe Armstrong, Robert Virding and Mike Williams @Ericsson

1998 – Erlang is open sourced

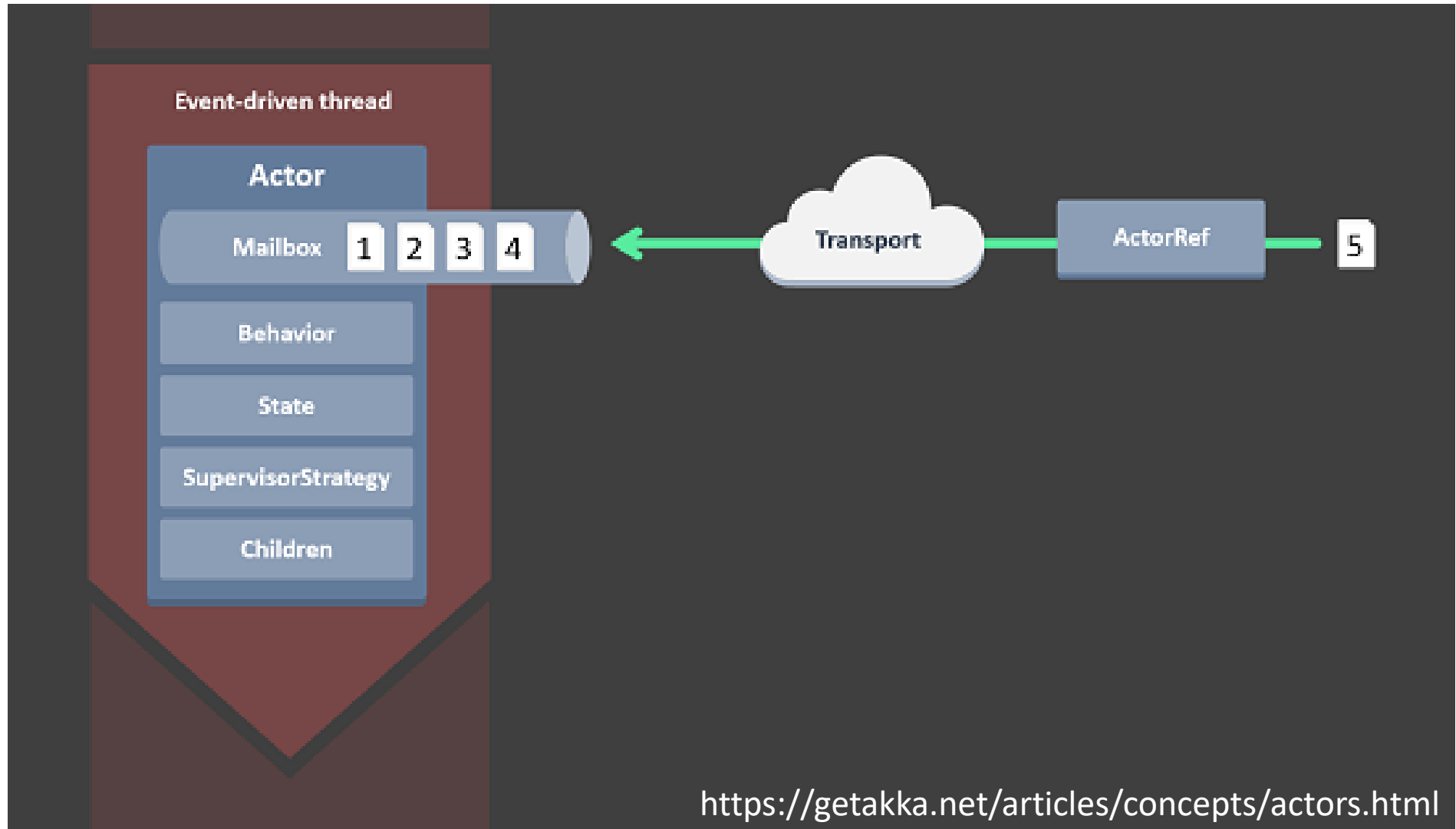
2010 – Akka for Scala & Java Jonas Bonér

2014 – **Akka.NET** Roger Johansson, Aaron Stannard et al.



akka.net

Actor = the smallest unit



Actor's properties

Self = own IActorRef

Sender = last received message sender's IActorRef

SupervisorStrategy = how handle children's failures

Context

actor's system

parent

children

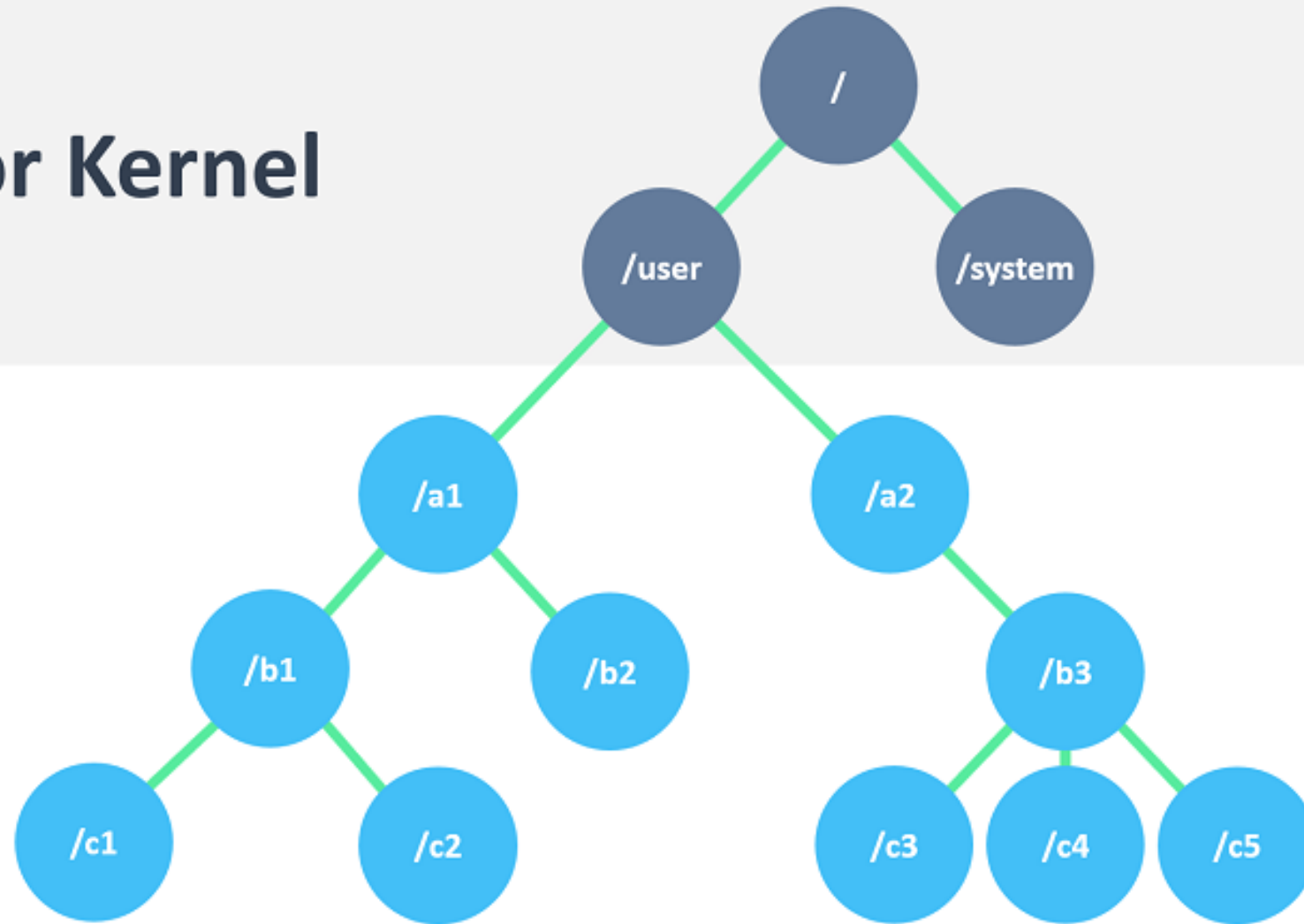
Actor system

hierarchical group of actors sharing common configuration

entry point for creating actors

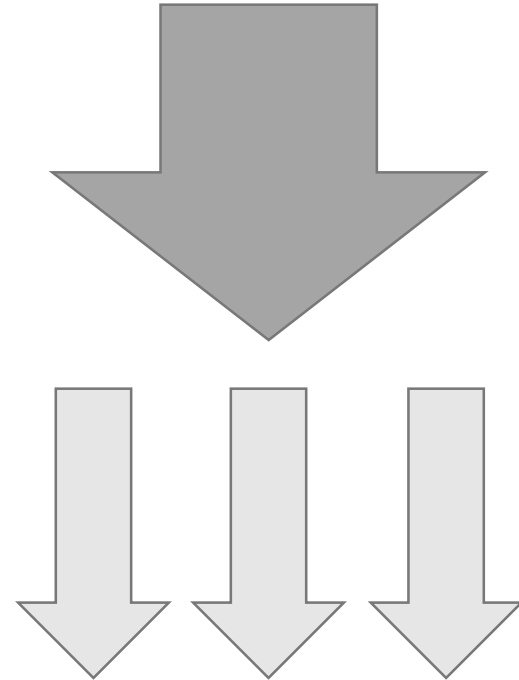
entry point for looking up actors

Error Kernel



Hierarchy #1

atomize work (divide and conquer)



Hierarchy #2

isolate errors (resilient system)



Supervision

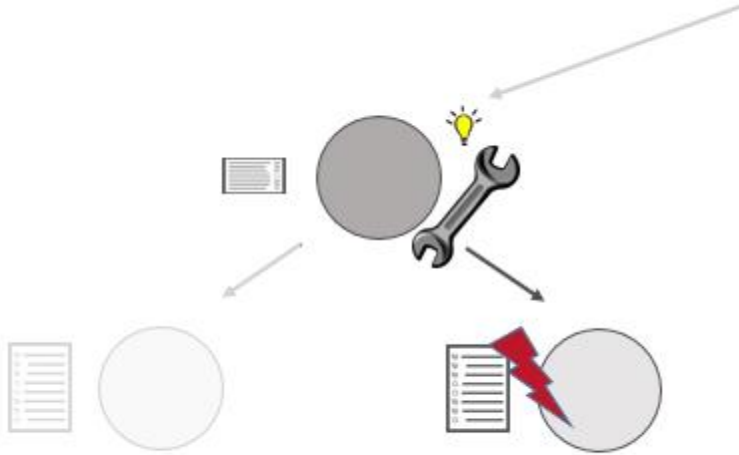
restart the child (default)

stop the child: this permanently terminates the child actor.

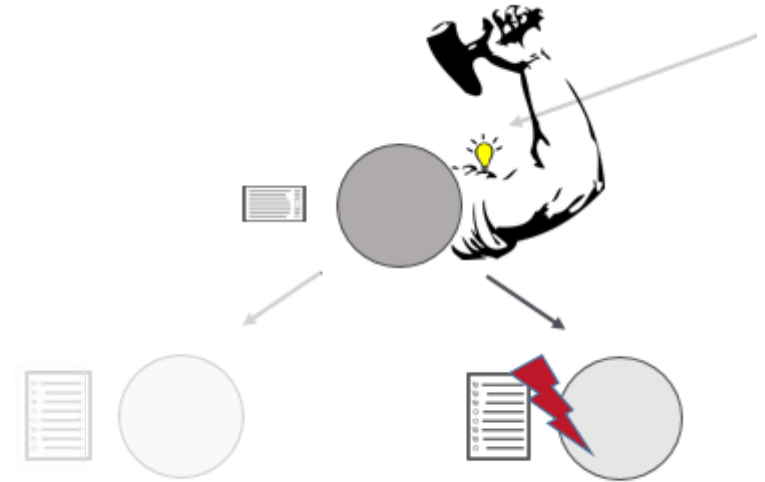
escalate the error (and stop itself)

resume processing (ignores the error)

Supervision

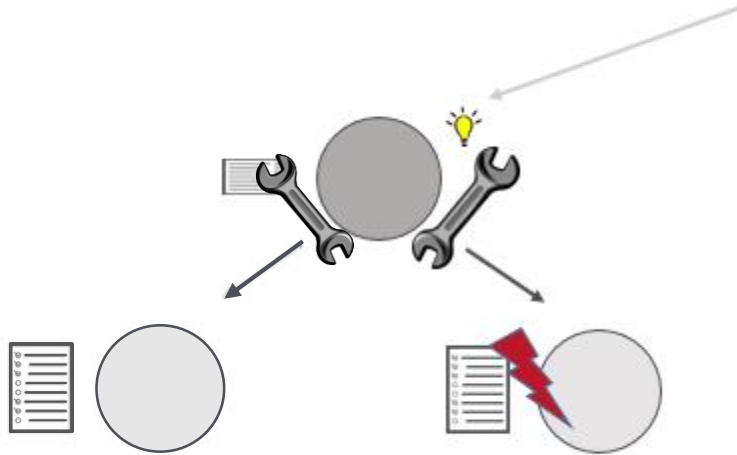


or

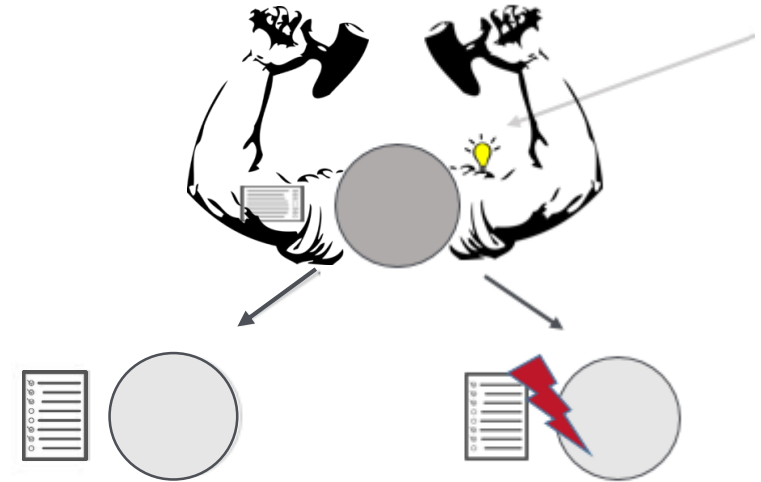


One-for-one

Supervision



or



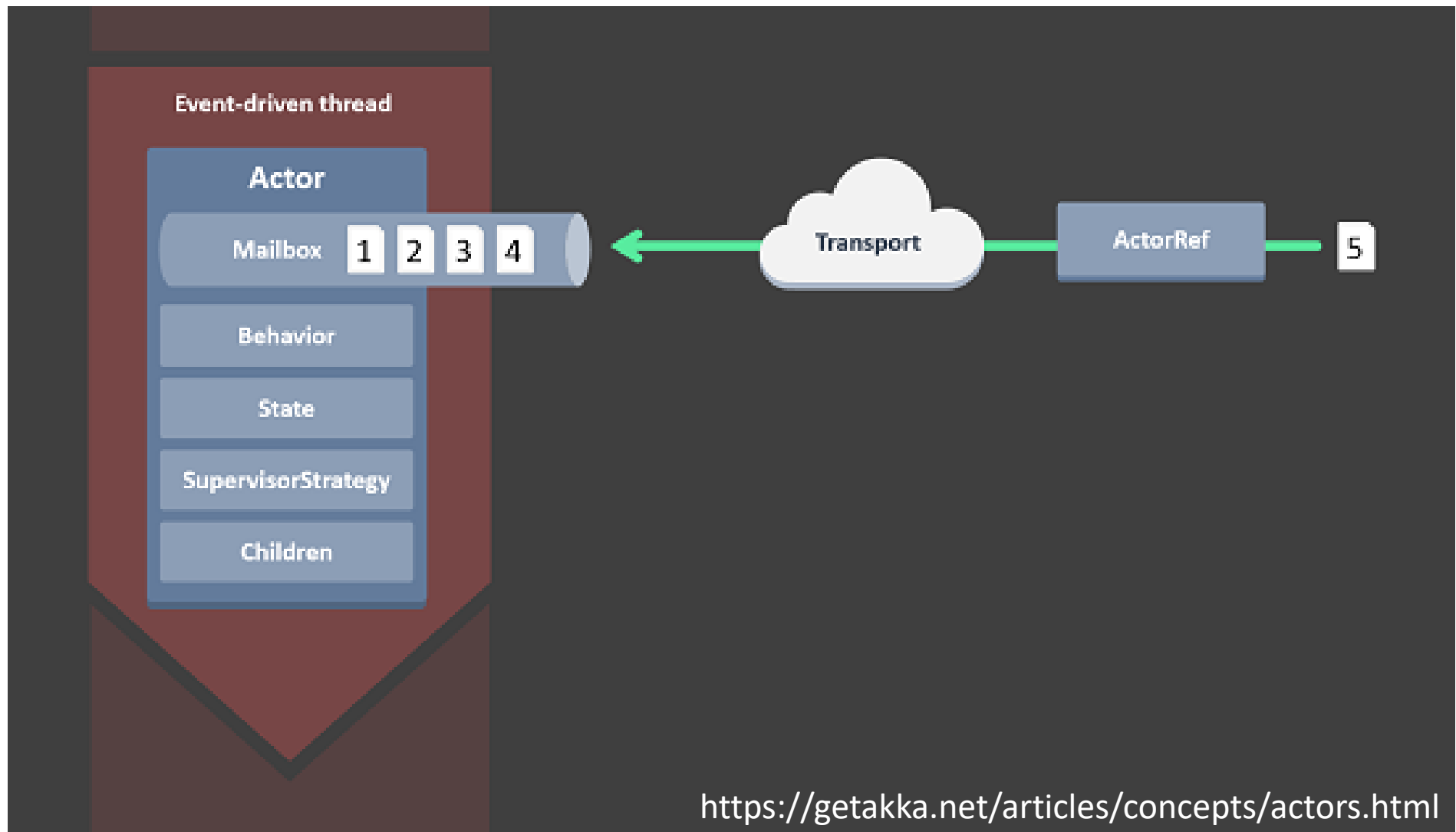
All-for-one

Message

any POJO

best practice: own custom classes

IMMUTABLE!!!



Getting started

Install-Package Akka

or

```
dotnet new -i "Petabridge.Templates::*"
```

```
dotnet new pb-lib -n "MyProject"
```



```
var actorSystem = ActorSystem.Create("MyActorSystem");
```

```
var actorSystem = ActorSystem.Create("MyActorSystem");
```

```
var actorRef = actorSystem.ActorOf(Props.Create(() => new MyActor()));
```

```
var actorSystem = ActorSystem.Create("MyActorSystem");
```

```
var actorRef = actorSystem.ActorOf(Props.Create(() => new MyActor()));
```

```
actorRef.Tell(new MyMessage());
```

```
public class MyActor : ReceiveActor
{
    public MyActor()
    {
        Receive<MyMessage>(msg => { ... });
    }
}
```

```
public class MyActor : ReceiveActor
{
    public MyActor()
    {
        Receive<MyMessage>(msg =>
        {
            var childRef = Context.ActorOf<MyOtherActor>(„myChild");
            ...
        });
    }
}
```



PLACE ORDER

Name *

First

Last

Email *

Phone

Foo Product

Price: \$10.00 Quantity:

Bar Product

Price: \$20.00 Quantity:

Total

\$0.00

Submit Order

Naïve approach...

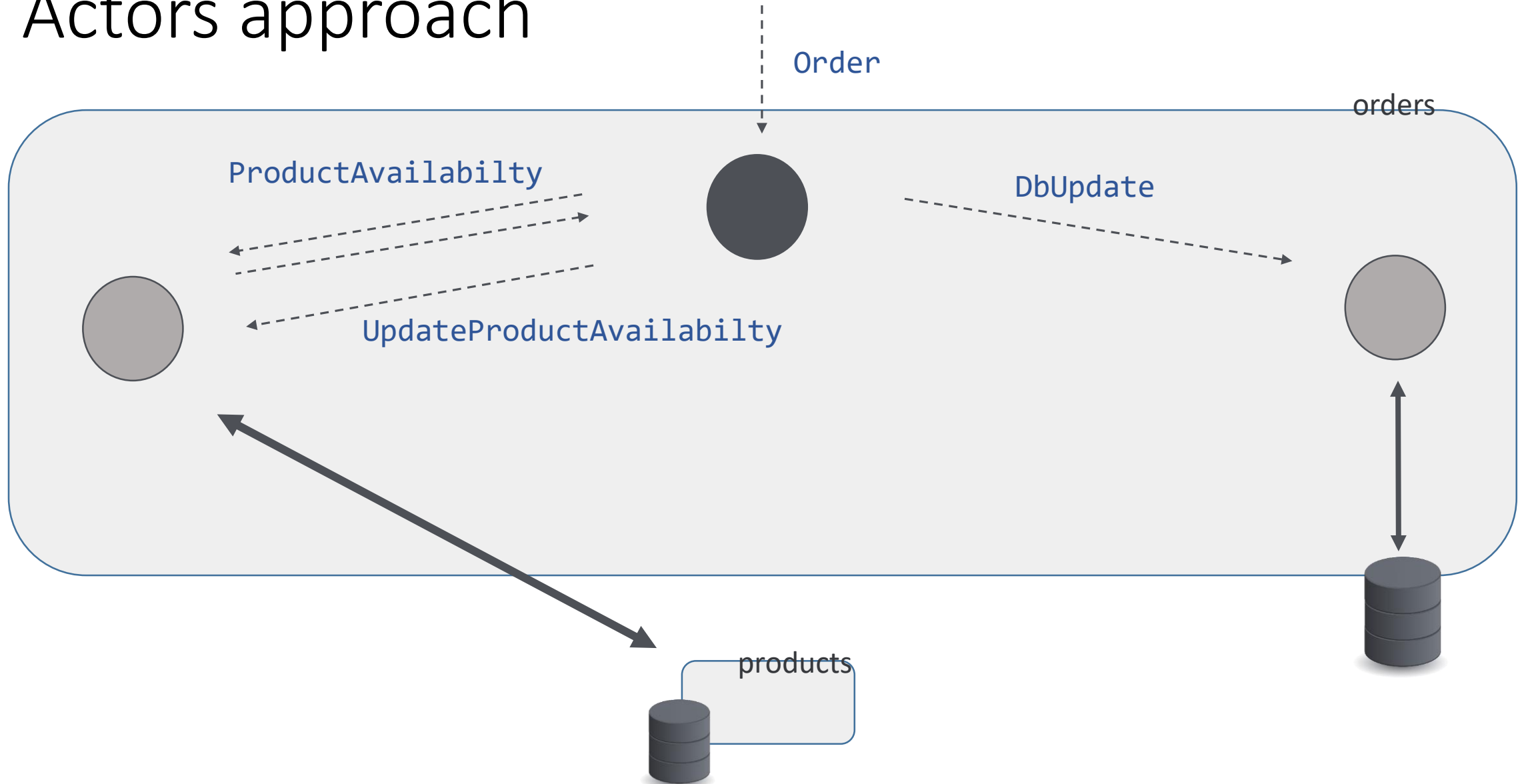
```
public Result SubmitOrder(Order order)
{
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = _productsService.Get(order.Items);
    if (inStock)
        _orderRepository.Add(order);
    else
        return Error(„OutOfStock”);
    _productsService.Update(order.Items);
    Notify(order);
    return Success();
}
```

A little less naïve approach...


```
public Result SubmitOrder(Order order)
{
    //ensure that logger won't cause issues - e.g. use nlog/log4net instead of own implementation
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = false;
    try {
        var inStock = _productsService.Get(order?.Items ?? new List<Items>());
    } catch (HttpRequestException){
        //log & return Error
    }
    if (inStock)
    {
        try {
            _orderRepository.Add(order);
        } catch (SqlException){
            //log & return Error
        }
    }
    else
        return Error(„OutOfStock”);
    try {
        _productsService.Update(order?.Items ?? new List<Items>());
    } catch (HttpRequestException){
        //log & return Error
    }
    try {
        Notify(order);
    } catch (Exception){
        //log & return Error
    }
    return Success();
}
```

```
public Result SubmitOrder(Order order)
{
    //ensure that logger won't cause issues - e.g. use nlog/log4net instead of own implementation
    Logger.Info($"Order {order.Id} submitted.");
    var inStock = false;
    try {
        var inStock = _productsService.Get(order?.Items ?? new List<Items>());
    } catch (HttpRequestException){
        //log & return Error
    }
    if (inStock)
    {
        try {
            _orderRepository.Add(order);
        } catch (SqlException){
            //log & return Error
        }
    }
    else
        return Error(„OutOfStock”);
    try {
        _productsService.Update(order?.Items ?? new List<Items>());
    } catch (HttpRequestException){
        //log & return Error
    }
    try {
        Notify(order);
    } catch (Exception){
        //log & return Error
    }
    return Success();
}
```

Actors approach



```
public class OrderReceiverActor : ReceiveActor
{
    public OrderReceiverActor()
    {
        Receive<SubmitOrder>(msg =>
        {
            var orderSubmitterActor = Context.ActorOf(OrderSubmitterActor.Props(msg.Order),
                msg.OrderId);
            //in supervisor handle InvalidActorNameException
            orderSubmitterActor.Tell(msg);
        });

        Receive<OrderProcessed>(msg =>
        {
            Context.Stop(Sender);
            //maybe respond to sender of original SubmitOrder with success
        });

        //make sure that no OrderSubmitterActor hangs - e.g. self schedule checks
    }
}
```

```
public class OrderSubmitterActor : ReceiveActor
{
    private Order _order;
    private IActorRef _productsServiceActor;
    private IActorRef _dbUpdaterActor;
    public OrderSubmitterActor(Order order)
    {
        _order = order;
        Receive<SubmitOrder>(msg => {
            _productsServiceActor = Context.ActorOf(ProductsServiceActor.Props(), "productsService");
            _productsServiceActor.Tell(new ReserveProducts(msg.OrderItems));
        });
        Receive<ProductsReserved>(msg => {
            _dbUpdaterActor = Context.ActorOf(DbUpdaterActor.Props(), "dbUpdater");
            _dbUpdaterActor.Tell(new InsertOrder(_order));
        });
        Receive<OrderInserted>(msg => {
            Context.Stop(_dbUpdaterActor);
            _productsServiceActor.Tell(new UpdateProducts(_order.Items));
        });
        Receive<ProductsUpdated>(msg => {
            Context.Stop(_productsServiceActor);
            Context.Parent.Tell(new OrderProcessed(order.Id));
        });
    }
}
```

Akka.TestKit



Pitfalls

mutable messages

long-running actions in Receive method

confusion around asynchronous operations with actors

Advanced features

Akka.Remote

Akka.Cluster, Akka.Cluster.Sharding, Akka.Cluster.Tools

Akka.Persistence

Akka.Streams

Akka.DistributedData (eventually consistent data duplication)

Where to use Akka.NET?

Resiliency & self-healing ability is essential

Need for concurrency, many requests/sec

Systems with huge complexity, many data sources

Highly parallelizable loads of work

Conclusion



photo by Caleb Jones on Unsplash

Recommended lecture

<https://getakka.net/index.html> - documentation

<https://github.com/akkadotnet/akka.net> - code

<https://github.com/petabridge/akka-bootcamp>

<https://petabridge.com/blog/> - training, articles

important to read before getting serious with Akka.NET:

<https://bartoszsypytkowski.com/dont-ask-tell-2/>

<https://petabridge.com/blog/top-7-akkadotnet-stumbling-blocks/>

how the idea of Akka.NET was born and first steps:

<https://rogerjohansson.blog/2015/07/26/building-a-framework-the-early-akka-net-history/>

Akka.NET & .NET Core

<https://havret.io/akka-net-asp-net-core>

Real world scenarios

Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)

- SNL Financial - <https://petabridge.com/blog/akkadotnet-goes-to-wall-street/>

Service backend

Concurrency/parallelism

Simulation

Batch processing

Communications Hub (Telecom, Web media, Mobile media)

Business Intelligence/Data Mining/general purpose crunching

IoT

Complex Event Stream Processing (e.g. real-time marketing automation)

Blockchain

Other actor model implementations

.NET:

Orleans

Azure Service Fabric Reliable Actors

<https://medium.com/@ericjwhuang/actor-pattern-in-action-dabff82fab53>

Akka on JVM (Java, Scala)

Akka JS

Erlang/Elixir

Cloud Haskell

Thank you!