Actor Oriented Architecture

Theo van der Donk

DRAFT – V0.2

Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

Grady Booch, Author of Object Oriented Analysis and Design with Applications

* * *

Actor Oriented Architecture (AOA) offers a refreshing new way of designing software applications. It combines the best practices of screaming architecture, clean architecture, domain driven design and actor oriented programming to form a unified set of patterns and best practices that gives hold to engineers that strive for a pure and clean design of their software, but are troubled by the implementation challenges that arise when a pure and clean design has to be combined with everyday reality of not so pure demands.

AOA mitigates the DDD trilemma by considering readability, understandability, testability and maintainability all even more important than achieving 100% purity. It provides the necessary patterns and best practices to do that, whilst compromising a little bit on purity to achieve a way of working that is both clean, pragmatic and fun.

It exploits the characteristics of actors (active objects) to combine pure logic with non-pure infra concerns in a natural way, understandable for both developers and interested customers. In combination with a suitable virtual actor framework, this results in scalable, highly redundant and concurrency-safe solutions with no additional work. Projects can start small, and when they evolve and gain traction, they can be scaled out gradually.

Especially when such frameworks also provide typical infra concerns like scalable persistence, timers and web gateways out of the box, and with minimal dependencies on the underlying infrastructure, developer productivity is conveniently raised to the next level as their focus shifts more and more to the actual domain logic the customer is paying for, and less and less to the peripheral matters like ORM's, databases, message buses, queues, API's, object mappers and the lethargic deployment and maintenance complexity that results from all of these.

* * *

Table of contents

1 Introduction

Many development teams struggle to keep their domain logic pure and readable, whilst at the same time dealing with the unavoidable reality of persistence, performance, scalability and concurrency.

The DDD trilemma is the source for one of these struggles. Developers, excited by the justified coolness of Domain Driven Design (DDD), start off really motivated to make their domain layer as pure as possible (no references to infra concerns) and also complete (no domain knowledge outside of the domain layer). Then, they find out that their pure logic needs information from the outside, namely whether a given email address is unique. The pure way of solving this is by fetching all thousands of email addresses in the system, and injecting them in the domain entity that must, as part of its functionality, determine whether a new email address is unique or not.

Of course, this does not scale well. The team wants to stay pure, so they decide to split their domain logic in 2: the part before the email validation; and the part after. And they have an arbitrary application service glueing the parts together, doing the intermediate query, and deciding whether part 2 should be invoked or perhaps an alternative path 2A.

However, this approach introduces domain logic in the application layer, which is also not what they want. It harms domain completeness (logic got split between domain layer and application layer) and code has become quite messy.

So, they refactor their code again, this time to inject the database query (nicely abstracted away, of course) in the domain entity that needs the lookup. But then Chief Review Engineer walks in, and properly points out that the domain is now not pure anymore.

Which brings the team, many weeks later and with a lot of accumulated frustration, back at where they originally started, but now in the 2nd iteration of the vicious circle. Trying to achieve the impossible combination of purity, completeness and performance at the same time.

1.1 Actor Oriented Architecture

Actor oriented architecture is a design philosophy that accepts that domain logic cannot be 100% pure when you also want to achieve completeness (all domain logic in the domain layer) and performance.

Readability, understandability, testability and maintainability are all considered even more important than purity.

The goal is to make it possible for every modest team to complete every project, simple or complex, on time and in a pragmatic and reasonable pure manner, practicing the great principles of DDD, without sacrificing readability, understandability, testability and maintainability.

1.2 Actors

As the name implies, actors play a key role in AOA. Or, to be more specific, virtual actors. In the traditional sense of the word, actors are independent entities within a system that receive messages, perform actions, and send out messages. Actors provide loose coupling between components.

Virtual actors are like actors, but one step beyond. Where traditional actors are usually tied to one specific process, virtual actors can move freely throughout the system. They can jump from one node to another node, without disruption of availability or loss of state. They are highly autonomous: they control their own persistence and concurrency, and can set timers on themselves for performing background tasks.

Usually, a virtual actor framework is used to provide supportive functionality, such as a message bus, a distributed locking mechanism, distributed persistence and persistent timers. But this is not required. It is also possible to practice AOA without an underlying virtual actor framework; it's just that you are then a bit hard on yourself.

> *The actor implementations themselves are, ideally, just plain objects. A good framework wraps itself around these objects, instead of imposing specific requirements on actors, such as having to descend from a certain base class, or speak HTTP or GRPC as wire protocol. All of these should be hidden and abstracted away.*

When implemented properly, actor technology is not visible for the developer (at least, not in the functional part of the application). That's the whole point of AOA: to completely hide the fact that you are using really advanced technology from your functional code.

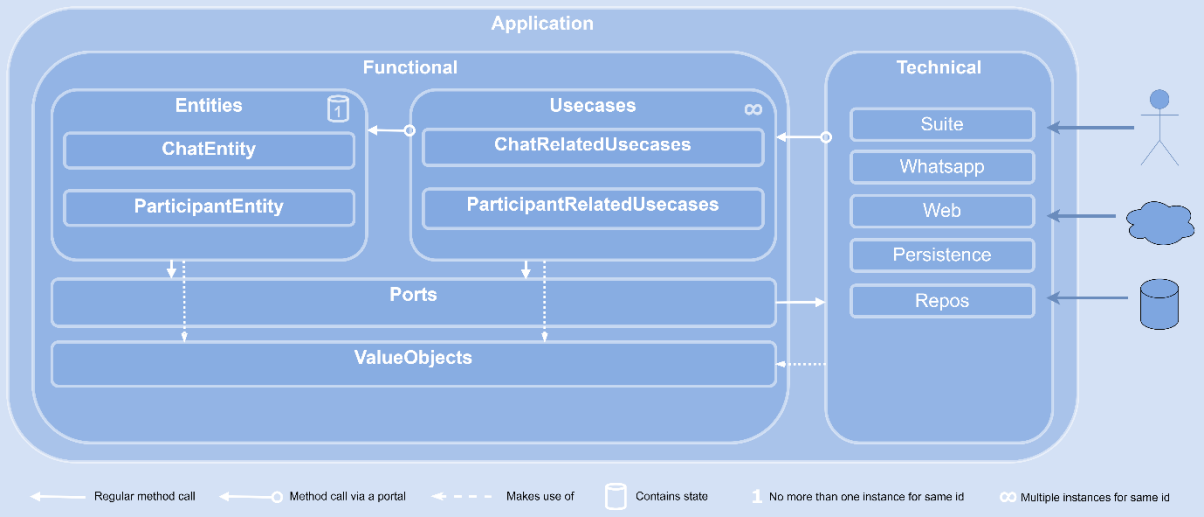
Actors are very similar to active objects. In fact, actors are one way of implementing active objects. But actors exhibit other characteristics as well that active objects do not have per se, like the possibility of reincarnating on another node. So, they are both a subset as well as a superset of active objects. Throughout this paper, the terms actor and active object are used interchangeably.

> *More background on virtual actors, code samples, tutorials and documentation is available on darlean.io.*

2 Application structure

Actor Oriented Architecture embraces the principles of screaming architecture. Just by looking at the source code folder structure alone, it should become clear to the reader what the application is doing, and where to find what he is looking for.

Let's look at an example.



The above figure shows a typical folder structure of an application that follows actor oriented design.

> *More complicated applications can recursively apply this design to submodules of the application.*

It is almost impossible to miss that at the highest level, the application consists of a *functional* compartment and a *technical* compartment. Functional is everything the customer is interested in *how* it works: which decisions are made when, and which actions are taken. Technical is everything the customer is only interested in *that* it works: persistence, api's, integrations, and the like.

> *There is an important distinction to make with regards to the technical part.* When *the technical compartment is invoked is usually functional: The customer wants a bank transaction to be persisted after all constraints have been met, and perhaps also in the middle of the transaction under certain conditions.*
>
> *But* how *the persistence works (the technicalities, like ORM's, SQL or object databases, transactions) is technical.*

The distinction between functional and technical is important, as it reduces the cognitive load on developers and customers (!) that want to understand the code. Developers can easily assess in which of the two compartments a certain piece of code belongs by asking themselves: does the customer care about the how. And customers only have to look in the functional compartment and can completely ignore the technical half.

3 The functional compartment

The functional compartment consists of value objects, ports, entities and usecases.

- **Value objects** are immutable objects that are used to transfer data between objects or processes. It is recommended that value objects automatically validate their own state, to make sure they are always in a valid state.
- **Ports** are typically interfaces that are required by the entities and usecases to interact with the outside. For example to persist internal state, to invoke actions on other entities or usecases, to invoke external API's, or to invoke a library to do specialized computations. Ports are implemented in the technical compartment.
- **Entities** are objects with state and methods. The methods interact with their state. Their state is private: it cannot be directly obtained or modified from the outside; only through the provided methods.
  Being actors, entities are also active objects, which means they are responsible for their own persistence and concurrency, for example by injecting a repository and a distributed lock.
  Entities reflect real-world concepts that have a unique identifier, which is very much like entities in DDD. Unlike DDD, the infrastructure should somehow ensure that there is never more than 1 AOA entity instance active within the entire system (which could consist of multiple connected application instances) for a given id.
- **Usecases** are similar to entities, but they do not have state. Also, the "no more than 1 instance active for a given id" restriction that entities have is not applicable to usecases. Usecases are also alive objects, but they cannot assume they are the only instance for their id.

The application's domain logic is contained within entities and usecases. Entities are similar to DDD entities in that that reflect domain entities with a given id. In the example above, we have a ChatEntity that represents a unique chat conversation and maintains its internal state, and we have a ParticipantEntity that represents one specific participant and its internal state.

3.1 Entities

Entities have state. For example, a ParticipantEntity can store its id, name and date of registration to the chat application. But it can also contain more dynamic information, like a list of chats the participant participates in, or a list of alerts about new chat messages that he has received.

Entities are similar to aggregate root entities in DDD. They are responsible for the consistency of any contained information (a ChatEntity could, for example, contain multiple messages). All interaction with the contained information (like the list of chat messages) must go through the methods of the entity.

For the outside, the contained information is just a bunch of value objects (either with or without an id) that can be obtained or modified by the methods of the entity.

How large should an entity be? The answer is: similar to an aggregate root. It should be large enough to provide strong consistency for those related information items that require strong consistency, and small enough not to become a performance bottleneck when the entity is locked for some operation.

> *AOA does not specify whether the state of an entity should be persisted as one blob or for example in a relational manner. Both are possible. But it is important to understand that even when (in the given example) individual chat messages are modeled as separate rows in a separate chatmessages table with their own unique id, that does not make them entities. They are, from the domain point of view, still integral part of the Chat entity, and in interaction, no more than valueobjects with an id field that go in and come out of the methods of the Chat entity.*

The infrastructure must provide a means to ensure that for any combination of entity type and id, there will never be more than one object instance active within the entire cluster. For a singe-process application this is, despite some catches, quite trivial; for a multi-process application, a virtual actor framework like Darlean takes away this hassle.

3.2 Ports

Persistence of internal state and other interaction with the outside is performed through ports.

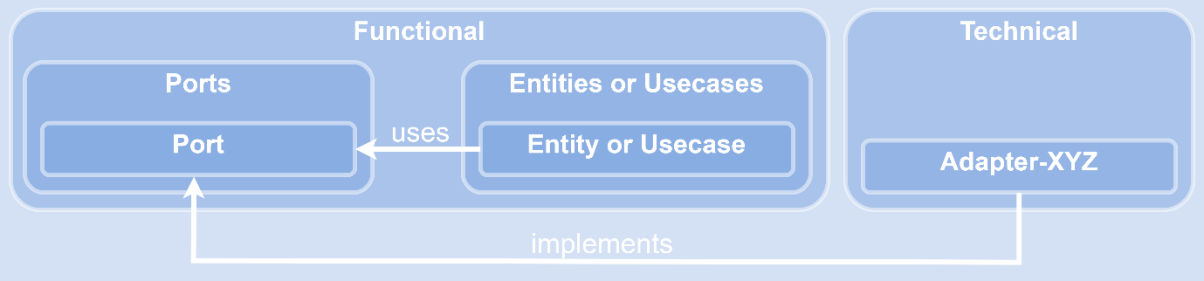


Figure 1 Entities and use cases use ports. They are injected during creation. Runtime, they invoke methods on the ports. Adapters provide the implementation. They reside in the technical compartment.

Ports are programmatic interfaces that should be specifically crafted for the entity that requires the port, such that the port only provides functionality the entity really needs, and with the data types the entity wants. The entity is in the lead for the interface of the port!

> *AOA discourages the common pattern of injecting entire repositories into entities and usecases, because that obscures which functionality an entity really needs. Doing so would complicate unit tests, for example, because it is not clear what to mock. Also, injecting per-entity-type repositories is discouraged, as an entity might need only a part all provided repository functionality.*
>
> *A simple and pragmatic solution is to implement one larger repository class (for example, one such class per type of entity) in the technical compartment of your application. This one class then implements multiple port interfaces for multiple entities. When the repository class uses the same method names as in the port interface, this approach requires no additional work. The class ben simply casted to the proper port interface and injected in the entity.*

3.3 Usecases

Usecases are quite similar to services in DDD. They do not have state, and multiple instances for the same usecase can exist. Within AOA, they act as a shield around the inner circle of entities. They hide the entities (which can be considered implementation details of the domain logic) from the outside.

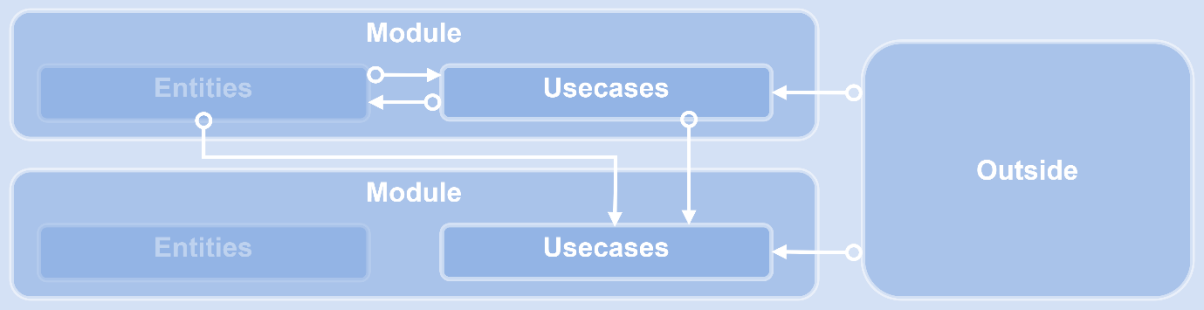


Figure 2 Usecases hide entities from the outside. The outside can invoke usecases from any module, but not entities. Usecases can invoke entities and usecases within their own module, and usecases (but not entities) from other modules. Entities can invoke entities and usecases within their own module, and usecases (but not entities) from other modules. All invocations go through portals.

Usecases can perform orchestration by invoking multiple entities. They can even make domain decisions, because they are still the domain layer. But when performing such orchestration, it is important to keep into mind that even though entities should provide strict consistency within themselves, the overall process is prone to concurrency issues.

> *Example: Let's consider the case in which we want a chat to have a maximum of 10 messages. A chat-related-usecase for posting a chat message could first requests the current list of messages from an entity, ask the entity to remove abundant messages, and then add their current message. But in parallel, another message for the same chat might come in and disrupt the process.*
>
> *A better solution here would be to move the removal logic to the chat entity, where it would be processed atomically, provided that the chat entity has proper synchronization implemented – which is his responsibility.*

Like entities, usecases use ports to communicate with the outside.

3.4 Interaction between entities and usecases

Interaction between entities and usecases (and vice versa) must never be direct through object references. That would harm scalability.

> *An important design principle of AOA is that applications must be designed to scale. Even when scalability is not technically required for a project (although you never know that from the start), designing for scalability improves software quality and architecture because of the decoupling of objects that is required to achieve scalability.*

Entities and usecases interact with each other via *portals*. A portal is similar to a factory, but different. A factory creates new object instances; a portal can create a new instance, but only when no instance is already active within the cluster. Otherwise, a portal creates a stub with the same interface as the entity or usecase that internally performs RPC (or any other suitable mechanism) to invoke the actual instance.

Portals help to decouple functionality. They also reduce the dependency chain by breaking up large dependency chains into subchains. To give an example, without AOA, a ChatRelatedUsecases object would require a ChatFactory which requires a Repo which requires a SQLDatabase.

By decoupling, the ChatRelatedUseCases would only require a dependency to a ChatEntityPortal (which, technically speaking, requires a dependency on the RPC framework or some other in-process or out-process communication mechanism, but that is so low level infra related, we ignore that for now).
But the ChatRelatedUseCases now has no more dependencies on that chat repo and underlying database. In fact, the ChatEntity could even physically be running in a different process or even at another computer.

Even when no RPC mechanism is used, developers must write their code as if every method call to an entity or usecase is serialized. That means, only passing value objects. It is not possible to pass a pointer to a real function or object (for example to do a database lookup) because that cannot be serialized. When an entity or usecase needs such functionality, these dependencies it should be injected during construction; not passed during invocation.

3.5 Value objects

Value objects are immutable objects that carry data throughout your application.

It is recommended that value objects validate themselves during creation to ensure they can never be in an invalid state.

Such validation can be related to functionality (a customer name must be at least 2 characters long and start with a capital) or technicality (a string must, for performance and security purposes, not be longer than 256 characters).

Value objects help against data corruption. For example, than can detect that a FirstName and a LastName object value are passed in the incorrect order to a function which assigns them to a Person value object.

Value objects can also greatly reduce the amount of code required to implement an application. Lesser code means lesser programming time, lesser reading time, lesser refactoring time, lesser testing time, lesser room for errors – so more quality and productivity.

When both functional and technical validation is in place, the value objects can be safely used anywhere in your application; from frontend to backend.

Yes, even the frontend (especially when it uses the same language as the backend, like TypeScript) can directly use value objects. The backend will automatically validate them when they come in. It can perform some authentication, and pass the value objects on to the corresponding use case. The use case can extract relevant element elements of the value objects, and pass those directly to the entities. The entities can use the same value objects as part of their internal state, and the persistence layer can choose either to directly store them in a blob database (provided that some versioning is applied to be able to read back old versions of value objects when the domain logic has changed) or make a mapping to another data structure.

> *The are some caveats. One of them is versioning. When frontend and backend are isolated (not same code base, not deployed simultaneously), it may be necessary to decouple frontend and backend. Also, when external parties are involved, a more stable API may be required.*

Using the same value objects within the entire application, especially when combined with an object database that can directly store and retrieve value objects, saves an enormous amount of code and effort: client-side objects; REST API specs and objects; domain objects; internal domain objects; ORM entities; and all the conversions in between become mostly abundant.

4 The technical compartment

The technical compartment of your application consists of all of those implementations the customer does not really care about how they work. It is enough for them to know *that* they are there to help facilitating their domain logic.

4.1 Structure

Unlike the functional compartment, where compartment names were really functional (persistence instead of database or sql), the technical compartment is about… well, technical things, so it is very appropriate (and even helpful!) to use technical names to help the reader understand what is going on: we do not have *just* a generic persistence implementation; no, we have a specific Postgres SQL persistence, or MongoDB persistence, or in-memory persistence. It is extremely helpful to reflect that in the file and folder names within the technical compartment.

The technical compartment contains all technicalities, which includes:

- Implementations of ports (also called *adapters* in hexagonal design)
- ORM's and database access
- Integrations with external API's
- Custom libraries that support the purpose of the application (but by themselves do not contain domain logic)
- Dependency injection root functionality or application setup

Which compartment structure suits best is project dependent. Some guidelines:

- For port implementations (adapters), it may be helpful to follow the same folder structure as for the ports. But make it specific by adding the kind of technology you use for the adapter, like chat-persistence-mysql.
- When you practice AOA, especially with a framework like Darlean, you will find out that dependency injection becomes a breeze. The dependency graph becomes less deep than for traditional software because every entity and every usecase forms in essence its own little mini program. All dependencies to other entities and usecases are by means of portals. You may even find out that using a DI framework becomes totally unnecessary.

4.2 Use of actors in the technical compartment

Despite the name, Actor Oriented Architecture does not imply that everything should be an active object. Only entities and usecases should be active objects, but value objects, for example, are an example of just regular objects.

For the technical compartment, actors *can* be used when they add value, but that is certainly not required.

A repository object, for example, does not have to be an actor. It does not need to be scalable (scalability lies in the underlying database); it just runs within the current process, and can be invoked directly (via a port that it implements) by entities and usecases.

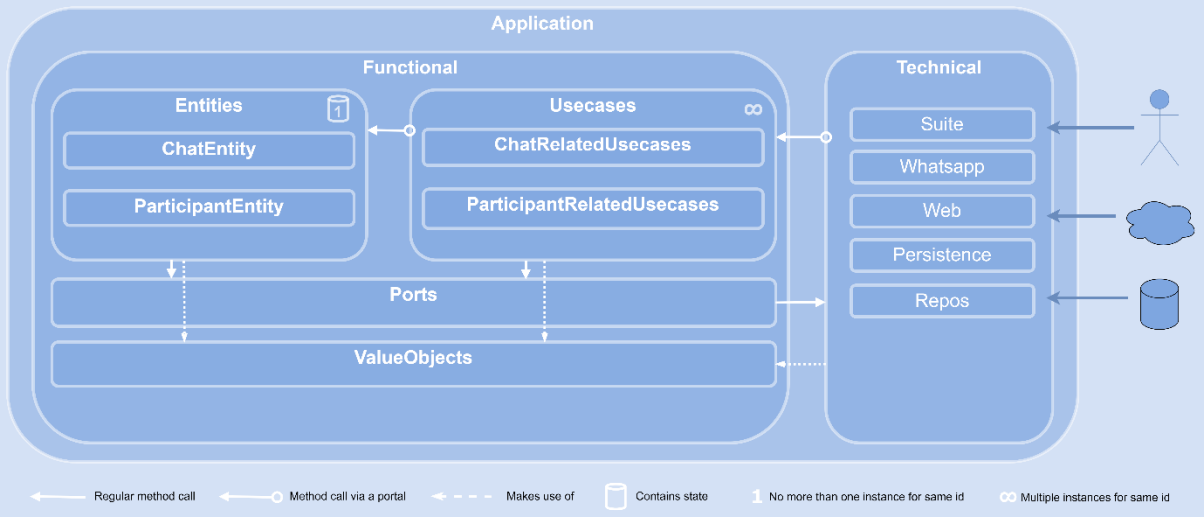
Also, an algorithm that performs certain computations can simply be a normal object. Or the controller that handles HTTP requests.

Where actors can play a role is when scalability is required within the technical compartment. An application could use a very low-level route planning component that does not include domain knowledge, but just performs all the computations (and possibly external API keys to traffic information services) required to compute the optimal route from A to B. The computation is very computationally expensive, so the scaling requirements would be different than for the rest of the application.

In such a situation, it pays off to make the route planner an actor, perhaps place it in its own application, scale that application up and down conform the needs, and implement a simple client object in the technical compartment of the applications that invoke the route planning. The client object does little more than obtaining a reference to one the actor instances, and invoking the proper method on it. The client object can be a regular object that implements a port interface via which an entity or usecase can invoke the route planning functionality.

5 Folder structure

Let's recall the design for a typical application that uses AOA:



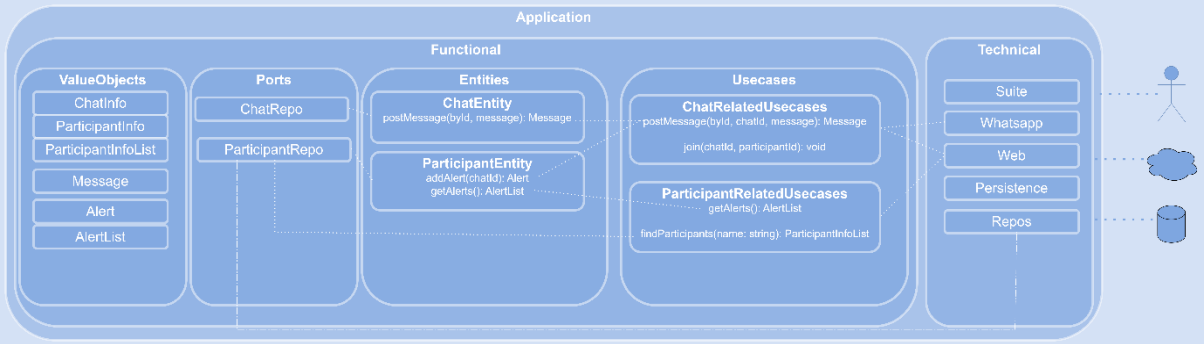
The recommendation of AOA is to separate your code at the highest level by functional vs technical. The functional part has functional names: entities, usecases, ports and value objects. The technical part has technical names: `persistence-sql`, `express-server`.

The nice thing about actor oriented design is that this exact same structure is reflected in the folder structure of the application source code, which makes it very readable and understandable:

```
src/
  functional/
    entities/
      chat-entity/
        chat-actor.ts
        some-supportive-file.ts
     participant-entity/
    usecases/
      chat-related-usecases/
      participant-related-usecases/
    ports/
      chat-repo.ts
      participant-repo.ts
    value-objects/
  technical/
    suite/
      entry-point.ts
    web/
      controller.ts
    persistence/
      chat-persistence-sql/
    repos/
```

6 An example application

Let's dive a bit deeper. In the figure below, I have elaborated a bit on the details of the chat application:



Just by looking at this diagram, one should already be able to get a grasp of the program. It is about chats and participants. End users can connect via whatsapp and the web. They can post chat messages (ChatRelatedUsecases.postMessage) and join a chat (ChatRelatedUsecases.join). It is also possible to obtain alerts for a participant (ParticipantRelatedUsecases.getAlerts), so apparently there is some alerting functionality implemented.

AOA advocates a layered or onion-like architecture. The outside world only interacts with the technical compartment (usually via controllers that handle the API calls). The technical compartment only interacts with the usecases (never directly with the entities). The use cases can invoke other usecases within the system or other entities within their module. The usecases can contain domain logic that does not naturally fit an entity, but should not contain state. The entities contain both domain logic and state. They also provide their own synchronization and persistence *control*. They to this by means of ports, that are implemented in the technical compartment. Value objects are used to carry (immutable and validated) data around.

Let's take a look at the "post message" use case, which allows a participant to post a message to a chat. The system will remember the message, and send an alert to all participants in the chat.

> *For this example, I use TypeScript because the Darlean virtual actor framework currently only supports TypeScript (work is in progress to support Go and .NET). The principles of AOA apply to other object oriented languages as well, and can also be applied (albeit a bit more harsh) without virtual actor framework.*
>
> *If you want Darlean support for your language, feel free to contact me! Or take the plunge towards TypeScript. It is a perfect language for 95% of projects that do little more than shoving data back and forth whilst making api calls and waiting for their response. And it can be used backend as well as frontend, so you can use the same code base (especially the value objects) frontend and backend without the need for an awkward REST interface in between that maps function calls via a CRUD-base REST interface to functional usecase calls.*

6.1 Functional: The value objects

Let's start by defining some value objects. We use the @darlean/valueobjects NodeJS library for that. It provides strong typing, validation, and cross-language exchange of data.

/functional/value-objects/index.ts

```
@stringvalue() export class ChatId extends StringValue {}
@stringvalue() export class ParticipantId extends StringValue {}
@stringvalue() export class ChatContent extends StringValue {}
@stringvalue() export class ParticipantName extends StringValue {}

@structvalue() export class ChatMessage extends StructValue {
  get id() { return ChatMessageId.required() }
  get chatId() { return ChatId.required() }
  get content() { return ChatContent.required() }
  get sender() { return ParticipantId.required() }
}

@sequencevalue(ChatMessage)
export class ChatMessageList extends SequenceValue<ChatMessage> {}

/**
 * Contains high-level information about a chat.
 * The messages are optional as they are only present when requested.
@structvalue() export class ChatInfo extends StructValue {
  get id() { return ChatId.required() }
  get participants() { return ParticipantIdList.required() }
  get messages() { return ChatMessageList.optional() }
}

@structvalue() export class ChatMessagePostOptions extends StructValue {
  get chatId() { return ChatId.required() }
  get content() { return ChatContent.required() }
  get sender() { return ParticipantId.required() }
}

@structvalue() export class ChatMessagePostResult extends StructValue {
  get message() { return ChatMessage.required() }
  get receivers() { return ParticipantIdList.required() }
}

@structvalue() export class Participant extends StructValue {
  get id() { return ParticipantId.required() }
  get name() { return ParticipantName.required() }
}

@sequencevalue(ParticipantId)
export class ParticipantIdList extends SequenceValue<ParticipantId> {}
```

6.2 Functional: The use case implementation

With our basic types, we can create the use case implementation:

/functional/usecases/chat-related-usecases/index.ts

```
export class ChatRelatedUsecases implements IChatRelatedUsecases {
  constructor(
    private retrieveChatEntity: (id: ChatId) => IChatEntity,
    private retrieveParticipantEntity: (id: ParticipantId) => IParticipantEntity
  ) {}

  public async createChat(): Promise<ChatMessage> {
    const id = ChatId.from(randomUUID());
    const chat = this.retrieveChatEntity(id);
    return await chat.create(id);
  }

  public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {
    const chat = this.retrieveChatEntity(message.chatId);
    const result = await chat.postMessage(options);

    for (const receiverId of result.receivers) {
      const receiver = this.retrieveParticipantEntity(receiverId);
      await receiver.addAlert(message.chatId);
    }

    return message;
  }
}
```

The use case has 2 portals injected: one for finding a chat entity for a given id, and one for finding a recipient entity. Note that these functions are synchronous. They do not really make a network call to find the entity; they just create a stub that can make RPC calls later.

The postMessage performs 2 tasks: posting the message to the chat entity, and alerting the participants in the chat. Our design choice here is to do this from the usecase (the other option would be to do this from within the chat entity). Our rationale is that we do not need strong persistency (it is no problem if a receiver is not notified in case of a rarity) and do not want to make the chat entity dependent on receivers and their alerting functionality.

The reason for naming the use case "chat-related-usecases" instead of just "chat-usecases" is to clarify the intent that the use case does things related to chats, but does not necessarily only operate on chat entities. It can even not operate on chat entities at all: creation of a new chat could, for example, be provided by means of SystemEntity.createChat. That would allow one system wide entity to administer all available chats. The createChat could in that scenario still be part of the chat-related-usecases. (In fact, which entity type is invoked is an implementation details that should not be reflected in the signature of the usecases).

Note: We could have made our code even purer by replacing the findParticipantEntity with:

findAlertableParticipant: (id: ParticipantId) => IAlertable

where IAlertable would simply contain the addAlert signature.

Or, we could even go one extra step beyond, and use

alertParticipant: (id: ParticipantId, chatId: ChatId) => Promise<void>

In our implementation, we could then suffice with

await alertParticipant(receiverId, message.chatId)

It is a matter of pragmatism which level of purity suits a project in its current stage.

6.3 Functional: The chat entity implementation

Now let's look at the implementation of the ChatEntity.

6.3.1 Chat entity internal state

Let's start with defining the internal state. See how we added helper/convenience methods to the value object to add a message and to create a new instance from provided info. That helps us later to keep the domain and testing logic clear.

/functional/entities/chat-entity/chat-state.ts

```
/**
 * Internal state of the chat entity.
 * We choose to use a value object here to simplify persistence.
 * For complex scenario's, non-immutable state objects could be a
 * better choice.
 */
@structvalue() export class ChatState extends StructValue {
  get id() { return ChatId.required() }
  get participants() { return ParticipantIdList.required() }
  get messages() { return ChatMessageList.required() }

  /**
   * Creates a new ChatState from the provided info.
   */
  public static fromInfo(chatId: ChatId, participants: ParticipantId[] = []): ChatState {
    return ChatState.from({
      id: chatId,
      paricipants: ParticipantIdList.from(participants),
      messages: ChatMessageList.from([])
    });
  }

  /**
   * Returns a copy of ChatState with the provided message added
   * to the end of the list of messages.
   */
  public withMessage(msg: ChatMessage): ChatState {
    return ChatState.from({
      ...this,
      messages: this.messages.push(message)
    });
  }
}
```

Now that we have the state, we can implement that ChatEntity object.

6.3.2 The ChatEntity outline

Let's first focus on the overall structure before we focus on the implementation of the actual methods.

/functional/entities/chat-entity/chat-entity.ts

```
/**
 * The chat entity itself
 */
export class ChatEntity {
  private state: ChatState | undefined;
  private fetched = false;

  constructor(
    private id: ChatId;
    private fetch: () => Promise<ChatState | undefined>,
    private persist: (state: ChatState) => Promise<void>,
  ) {}

  @mutex()
  public async create(): Promise<ChatInfo> {}

  @mutex()
  public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {}

  @mutex()
  public async getInfo(includeMessages: boolean): Promise<ChatInfo> {}

  /**
   * Maps the provided private state to a public ChatInfo object.
   */
  private mapToChatInfo(state: ChatChat, includeMessages = false): ChatInfo {
    return ChatInfo.from({
      id: state.id,
      participants: state.participants,
      messages: includeMessages ? state.messages : undefined
    });
  }

  private async ensureFetched(): ChatState | undefined {
    if (!fetched) {
      this.state = await fetch();
      this.fetched = true;
    }
    return this.state;
  }

  private async checkExists(): ChatState {
    if (!await this.ensureFetched()) {
        throw new Error('Chat does not exist');
      }
      return this.state;
    }
  }

  private async checkNotExists() {
    if (await this.ensureFetched()) {
        throw new Error('Chat already exists');
      }
    }
  }
}
```

The class has a private field state of type ChatState | undefined. The state is fetched when one of the methods needs the state by calling checkExists. As we will see below, the state is only

assigned during an external call to `create`. When the state is not assigned, and can also not be fetched because it is not present in the underlying storage, an error is thrown by `checkExists`.

So, the only way to do anything meaningful with the ChatEntity (like invoking the `getInfo` or `postMessage`) is by first having officially created it by calling `create`.

> *Of course, you only have to call `create` once. When you would restart your application, a new instance would of course be instantiated, and the `ensureFetched` would the find the state in the underlying database, so no exception would be thrown and the `getInfo` and `postMessage` can be used directly.*

So, we have some helper methods (`ensureFetched`, `checkExists` and `checkNotExists`) that are supportive to the actual domain logic that we will cover in a while. It's a small price you pay for being an active object – but the good news is, that this logic is very similar between actors, so it can easily be moved out to a project specific library or base class.

6.3.3 The create method

The `create` method is invoked from the outside when someone wants to create a new chat entity. The way actors work, one can ask the portal for any id you like, and the portal will return a valid reference. The portal has no means of knowing for which id's there exist actor entities.

As a corollary, it is possible to invoke methods on an actor that "does not exist". Actors by themselves have no such concept as existence. When they are instantiated, they exist. Physically. But not logically, from a user point of view. In normal programming, invoking an action on a not-yet-explicitly-created object would give a `NullPointerException`. We want something similar for our ChatEntity.

And that's where the `create` method kicks in:

```
@mutex()
public async create(): Promise<ChatInfo> {
  // Check that we do not yet exist. We cannot create ourselves twice!
  await this.checkNotExists();

  // Set our initial (quite empty) state
  this.state = ChatState.fromInfo(this.id);

  // Persist our initial state.
  await this.persist(this.state);

  // Obtain our chat info and return it
  return this.mapToChatInfo(this.state);
}
```

The create method checks that we do not yet exist. How? By fetching our state. When our state is defined (present, assigned), we exist, and `checkExists` throws an error. When our state is undefined (not present, not assigned), we have not yet been created.

In that case, we set our state to a new, empty state that only contains our `id`. Then, we persist our state, so that, should our application be restarted, the next instance for the same id finds the state and knows that it is already created.

Finally, we derive and return some high level information about ourselves.

> *It is good practice that methods that change something, return the new state. It may introduce some overhead when the receiver is not interested, but it is the only atomic way for a receiver to obtain the new state immediately after its own operation.*

6.3.4 The getInfo method

The getInfo method returns some basic information about the chat entity.

```
@mutex()
public async getInfo(includeMessages: boolean): Promise<ChatInfo> {
  // Check that we exist, and return our chat info.
  return this.mapToChatInfo(await this.checkExists(), includeMessages);
}
```

The implementation is simple and straightforward. It checks whether we exist (if not, an error is thrown), and derives our info. We deliberately return a ChatInfo object instead of our internal ChatState because we do not want to expose our private state to the outside.

For performance reasons, we only include the list of messages (which can become quite large) in the result when explicitly asked for by means of the includeMessages argument.

6.3.5 The postMessage method

Finally. The postMessage method, the heart of the entity.

```
@mutex()
public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {
  // Ensure we have been created before.
  const state = this.checkExists();

  // Create a new Message value object from the input options
  const message = ChatMessage.from({
    id: this.state.messages.length,
    chatId: this.state.id,
    content: options.content,
    sender: options.sender
  });

  // Add the message to our internal state
  this.state = this.state.withMessage(message);

  // Ask infra to persist our internal state
  await this.persist(this.state);

  // Craft our response and return it
  return ChatMessagePostResult.from({
    message: message,
    receivers: state.participants
  });
}
```

Let's first focus on the naming of postMessage. We might also have given it a more standard name, like addMessage. But that would be technical. What we are functionally doing is posting a chat message, and we want the name to reflect that.

> *AOA encourages method names that describe the functional act, instead of the technical workings.*

Then, the implementation. When inspecting the code, it should become obvious that:

1. It checks that we already exists; and throws an error if not.
2. Uses the provided input options to create a new ChatMessage object
3. Adds the message to our internal state
4. Persists the state (by invoking the `this.persist` port)
5. Crafts a response message and returns it.

6.3.6 The @mutex decorator

The `@mutex` decorator before the public methods ensures that the public methods will not execute in parallel with other methods of the same instance. We leave it up to the reader to provide an implementation for that, as NodeJS does not have a native mutex.

In the simplest form, it could simply be a global application-wide mutex. A more advanced form would be a mutex per instance; and the most advanced form, like Darlean's SharedExclusiveLock, would even support single-write-multiple-read functionality.

6.4 Technical: A simple portal implementation

Usually, a virtual actor framework already provides a portal implementation. For education's sake, let's implement our own very simple non-production-ready portal implementation for use in a single-process application:

/technical/portal/simple-local-portal.ts

```
export class SimpleLocalPortal<T> {
  private instances: Map<string, T>;

  constructor(private creator: (id: string) => T) {
    this.instances = new Map();
  }

  public retrieve(id: string): T {
    const existing = this.instances.get(id);
    if (existing) {
      return existing;
    }

    const instance = creator();
    this.instances.set(id, instance);
    return instance;
  }
}
```

Note that, for production use, we must implement a mechanism that also deletes old entities. Otherwise, the application will finally run out of memory. This is, however, not trivial, because we must not only remove these instances from our administration; we must also take care that parts of our software have no references anymore and that the instance itself does not keep itself alive, for example when it has active timers.

A good solution (as implemented by Darlean) is not to return direct pointer references to the instances, but to create stubs and return the stubs. The stubs can then throw an error when someone tries to invoke a method after the instance is invalidated. Also, instances should have a finalization hook that will be invoked, so that they can clean up, stop timers and finalize in a controlled manner.

6.5 Technical: A simple persistence implementation

For the sake of education, let's implement our own simple persistence provider that fetches and persists blobs from a file system.

/technical/persistence/persistence.ts

```
export interface IPersistence {
  persist(id: string, contents: Buffer): Promise<void>;
  fetch(id: string): Promise<Buffer | undefined>;
}
```

/technical/persistence/file-persistence.ts

```
export class FilePersistence implements IPersistence {
  constructor(private folder: string) {}

  public async persist(id: string, contents: Buffer) {
    await fs.writeFile(this.folder + id, contents);
  }

  public async fetch(id: string): Promise<Buffer | undefined> {
    try {
      return await fs.readFile(this.folder + id);
    } catch (e) {
      return undefined;  // File not found
    }
  }
}
```

> *Please do not use this code in production! Although it scales very well, provided the data is stored on a shared filed system which is accessible for other processes; and although there should not be concurrency issues because the entity that writes to one specific file should be protected by a mutex; corruption can still occur when the server, for example, suddenly crashes in the middle of a write.*
>
> *This solution also lacks more advanced querying functionality that might be required by real-life projects.*

6.6 Technical: Repository implementation

Now that we have the persistence, let's jump to the generic repository that can be used for any type of state that is a StructValue value object:

technical/repository/value-object-repository.ts

```
export class StructValueRepository<T extends StructValue> {
  private serializer: CanonicalJsonSerializer;
  private deserializer: CanonicalJsonDeserializer;

  constructor(private persistence: IPersistence) {
    this.serializer = new CanonicalJsonSerializer();
    this.deserializer = new CanonicalJsonDeserializer();
  }

  public async persist(id: string, value: T) {
    const serialized = this.serializer.serialize(T);
    await this.persistence.persist(id, value);
  }

  public async fetch(id: string, valueClass: typeof T): T | undefined {
    const contents = await this.persistence.fetch(id);
    if (!contents) {
      return undefined;
    }
    const canonical = this.deserializer.deserialize(contents);
    return valueClass.fromCanonical(canonical);
  }
}
```

6.7 Technical: Let's glue it all together

Now that we have all the ingredients, let's write our application's entry point in the form of a ChatSuiteBuilder class that builds references to all contained components, wires them together, and provides a method to actually build the chat suite:

/technical/suite/entry-point.ts

```
export class ChatSuiteBuilder() {
  public chatPersistenceRef: Ref<IPersistence>;
  public chatRepoRef: Ref<FileRepository>;
  public chatPortalRef: Ref<SimpleLocalPortal<ChatEntity>>;
  public cruPortalRef: Ref<SimpleLocalPortal<ChatRelatedUsecases>>;
  public chatRelatedUsecasesCreatorRef: Ref<(id: ChatId) => ChatRelatedUsecases>;
  public chatRelatedUsecasesRef: Ref<ChatRelatedUsecases>;

  constructor( public persistenceFolder: string ) {
    this.chatPersistenceRef = ref(
      () => new FilePersistence(persistenceFolder)
    );

    this.chatRepoRef = ref(
      () => new StructValueRepository<ChatState>(this.chatPersistenceRef()
    );

    this.chatPortalRef = ref(
      () => new SimpleLocalPortal<ChatEntity>((id) => {
        const fetcher = () => this.chatRepoRef().fetch(id, ChatEntity);
        const persister = (state: ChatState) => this.chatRepoRef().persist(id, state);
        return new ChatEntity(id, fetcher, persister);
      })
    );

    this.chatRelatedUsecasesCreatorRef = ref(
      (id) => {
        const chatFinder = (id: ChatId) => this.chatPortalRef().retrieve(id);
        const participantFinder = ...;
        return new ChatRelatedUsecases(chatFinder, participantFinder);
      }
    );

    this.cruPortalRef = ref(
      () => new SimpleLocalPortal<ChatRelatedUsecases>(
        this.chatRelatedUsecasesCreatorRef()
      )
    );

    this.chatRelatedUsecasesRef = ref(
      () => this.cruPortalRef().retrieve('')
    );
  }

  public build() {
    return {
      chatRelatedUsecases: this.chatRelatedUsecasesRef()
    };
  }
}
```

For those who are familiar with DI, this class forms the application's composition root. That is the place where you have all the knowledge about how you want to glue everything together. It's the place where you know that you want, for example, to use a FilePersistence as implementation of IPersistence. It can also be the place where you load relevant configuration

from the command line, environment or configuration files – or you can provide them as arguments, like we did for `persistenceFolder`.

We have deliberately made all member public. As we will see later in the section about integration testing, this greatly simplifies customization.

> **About pure DI and the use of Ref's**
>
> *You may have noted the use of Ref's in the chat suite builder. AOA advocates simplicity; simplicity means to leave complicated things out; and a DI framework is complicated as soon as your application becomes more complicated. As soon as you end up in a situation in which a certain interface can sometimes have one implementation, and in other parts of your code another one, most frameworks and developers begin to flip.*
>
> *Pure DI provides simple DI without additional frameworks simply by using constructor injection applied recursively. But, it is a little bit simple for advanced use cases, one such use case being reusing your DI code for automatic tests, but then slightly adjusted, with some implementations replaced.*
>
> *A simple solution is to make the composition root an object, with the dependencies as protected fields. Subclasses can override these fields with their own implementation before the actual construction takes place.*
>
> *To make it possible for dependencies to use other dependencies that might later be overridden by subclasses, we need a level of indirection. And that's where the Ref's (short for references) kick in. A Ref is a reference to a possibly-later-to-created instance (but it can also contain an already-create instance). A Ref is a function without arguments. As soon as it is invoked for the first time, it will create a new instance and return that instance. The instance is internally cached and returned on subsequent invocations of the Ref function.*
>
> *The implementation of ref is provided in a separate section below.*

What the code is doing:

1. Create chat persistence using `FilePersistence`
2. Create chat repo that uses the chat persistence for actual persistence
3. Create a chat portal
    a. As fetcher, we define an anonymous function that invokes the repo
    b. As persister, we define an anonymous function that invokes the repo
4. Create a portal for the chat-related-usecases actor
    a. The portal requires a creator function that creates new actor instances for a given id. We use an anonymous function that creates a new `ChatRelatedUsecases` instance.
    b. The instance requires a function that can be used to get a reference to a `ChatEntity` for a given id. We use an anonymous function that invokes the chat portal.
5. Obtain a reference to a `ChatRelatedUseCases` instance and return it.
    a. We need only one such usecases object; not one per chat. Therefore, we simply use the empty string as an id.

The code may feel a bit complicated, which is caused by the fact that we try to abstract a lot of details from the actual actors. The wiring complexity is nicely isolated within the composition

root, which keeps the domain logic entities and usecases clear. They have absolutely no knowledge of the underlying technicalities of portals, repo's and persistence.

6.7.1 Using the entrypoint

Now that we have the entry point, we can actually start chatting!

/index.ts

```
// Create the suite – done once per application
const suite = new ChatSuiteBuilder('/data/chat/').build();

// Obtain the usecases stub
const chats = suite.chatRelatedUsecases;

// Create a new chat (not necessary when a chat had already been created
// before – but we'd have to know the id then to be able to send a message).
const chatInfo = await chats.createChat();

// Send a chat message. Use chatInfo.id as chat id. The message will have been added
// to the chat, and alerts will have been sent to the chat participants.
const result = await chats.postMessage(ChatMessagePostOptions.from({
  chatId: chatInfo.id,
  content: ChatContent.from('Hello world!'),
  sender: ParticipantId.from('P5370')
}));
```

The message has been added to the chat, and alerts have been sent to all participants. The result object (of type ChatMessagePostResult) now contains a reference to the new message (including the assigned internal id that we could use for any future modifications to the message, or to delete the message from the chat, for example) and the participants to which an alert was sent.

6.7.2 Implementation of Ref

In the text frame on 'Pure DI and the use of Refs', we have explained what refs are and which purpose they serve.

Here we provide an implementation for the interested reader.

/technical/suite/di.ts

```
/**
 * Represents a function that, when invoked repeatedly, always invokes the same
 * instance of T. When required, the instance will be created on the fly, invisibly
 * to the caller of the function.
 */
type Ref<T> = () => T;

/**
 * Returns a Ref of type T. When invoked for the first time, the returned function
 * returns a new instance of T, created by the `creator` function. When invoked
 * subsequently, keeps returning that same previously created instance.
 */
function ref<T>( creator: T | () => T ): Ref<T> {
  let instance = (typeof creator === 'function') ? undefined : T;
  let creating = false;
  return () => {
    if (creating) {
      throw new Error('Circular reference');
    }
    if (!instance) {
      creating = true;
      instance = creator();
      creating = false;
    }
    return instance as T;
  }
}
```

7 Scalability and concurrency: Invoking an LLM

We did not spend much coding effort on concurrency. At least, not in our domain layer. We just decorated the entity methods with `@mutex`, and that's it. And that's enough to become concurrency-safe. As mentioned before, when using a real framework like Darlean, there are more advanced locking strategies that we can use instead of plain `@mutex`, like a multi-read-exclusive-write strategy.

Without making any changes to the domain logic itself, we could replace our self-crafted little exemplary framework of `SimpleLocalPortal` and `FilePersistence` with an advanced framework like Darlean and obtain full scalability across multiple processes with powerful multi-read-exclusive-write synchronization and the hard guarantee that for any given entity, no more than 1 instance will ever be active within the entire cluster at the same moment.

We would be fully scalable and at the same time fully protected against concurrency issues – provided that we put code that we want to protect together within the same method with the proper mutex, of course.

Let's investigate what would happen when we would add a long-running step to our domain logic. Assume we want to invoke an LLM to give the sender of a message a friendly automatic response. And not just that – when multiple messages for the same chat are received in parallel, we nicely want to have each message and its corresponding AI response directly below each other; not interleaved with other messages.

Invoking an LLM is a long running operation. To provide the guarantee that the AI response comes directly after the original message, we place the LLM invocation command in the `ChatEntity` itself (not in the `ChatEntityRelatedUsecases`).

> *Without this requirement, we could also have invoked the LLM from our chat-related-usecase. But because we want strong consistency, we must place it in our entity.*

We modify our `ChatEntity.postMessage` as follows. The changes are in bold.

/functional/entities/chat-entity/chat-entity.ts

```
@mutex()
public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {
  // Ensure we have been created before.
  const state = this.checkExists();

  // Create a new Message value object from the input options
  const message = ChatMessage.from({
    id: this.state.messages.length,
    chatId: this.state.id,
    content: options.content,
    sender: options.sender
  });

  // Add the message to our internal state
  this.state = this.state.withMessage(message);

  // Ask infra to persist our internal state.
  // Note: That we do this is domain logic. We could also first have invoked the
  // LLM, and persist when everything is ok. But we assume here that the
  // business made the decision that even when LLM fails, the message must be
  // preserved.
  await this.persist(this.state);

  const llmResponse = await this.invokeLlm(
    LlmPrompt.from('Provide a friendly response to the following message'),
    LlmInput.from(options.content)
  );

  // Create a new Message value object from the LLM response
  const message = ChatMessage.from({
    id: this.state.messages.length,
    chatId: this.state.id,
    content: MessageContent.from(llmResponse.content),
    sender: ParticipantId.from('LLM')
  });

  // Add the message to our internal state. We have the lock, so no other message
  // could have come in between.
  this.state = this.state.withMessage(message);

  // Persist our state again
  await this.persist(this.state);

  // Craft our response and return it. We leave it as an exercise to the reader
  // to add the LLM response message to the result object.
  return ChatMessagePostResult.from({
    message: message,
    receivers: state.participants
  });
}
```

> *We can make some improvements to this software. We could (and should) move parts of the functionality to separate private methods. We could (and perhaps should) have abstracted the LLM even more by placing the prompt outside of this entity, and just providing a port to "askLlmForMessageResponse". But these are not the point of this example.*

The example illustrates how simple it is to perform concurrency-safe operations. We do not need database-level transactions and rollbacks – in fact, they would incorrectly delegate a domain responsibility to an infra implementation, which would strongly couple domain logic with persistence implementation. Database level locking would feel awkward because the act

of invoking an LLM is not database related at all. It would be unnatural to use a database lock or transaction for that.

We just lock our code by means of a mutex; the code around it (mainly the portal in our simplified example) guarantees that we are the only active instance; and together these 2 concepts provide strong consistency at domain level.

8 Querying entities

A wonderful concept in DDD is that entities are responsible for their own data, and that all access to their data must be performed through their own exposed methods.

But then, how do we efficiently get a list of all ChatEntities in the system? Or a list of all ChatMessages with certain contents? Do we really have to go through all ChatEntity instances in the system to find those ChatMessages?

The pure answer is: yes. Define a SystemEntity at the root of your hierarchy that has a list of all chat entities. And then ask the system entity to iterate all chat entities to find the requested chat messages. That's the pure approach. And the nice thing is that most of this can be performed in parallel. But still, it does not always feel right…

So now the pragmatical approach. I think it is reasonably fair to give the ChatRelatedUsecases actor (which is conceptually very close to the ChatEntity that owns the data) **read-only** access to the underlying storage. For example, by injecting a ChatRepo.findMessagesByContent method as port into ChatRelatedUsecases instances. The ChatRepo instance would have to understand the format of the data stored: the record fields for a relational database, or the object structure of an object database. But since it is already persisting value state for the ChatEntity instances, that is not new. And the repo must map the results to a public format (not in the internally stored ChatState format which is private to the entity).

Here we enter a gray area. It could be that the entity has some 'data enrichment functionality'. It could be implemented in such a way that whenever someone asks for the first name, all characters are first capitalized. The repository does not have that logic, so it will return the raw first name, and also use the raw first name instead of the capitalized first name in queries.

Either that logic must be duplicated to the repository; or the logic must be duplicated to the ChatRelatedUsecases. Both solutions are suboptimal and not pure. A poor man's optimization might be to define this mapping logic as a separate function, which is then used by both the entity and the repository or the usecases. This ensures that they all use the same functionality, but it exposes the functionality that should be private to the entity.

Unfortunately, AOA does not provide a 100% pure solution for this challenge. It's a matter of simply playing the hand you're dealt, and being pragmatic.

> *When using such an approach, do not give the repository (or another part of your software outside of the entity actor itself) write-access to its data. That would seriously breach all consistency guarantees that AOA brings.*
>
> *Even when you have to update a lot of entities, do it nicely through the corresponding publicly exposed entity methods. When performance is an issue, consider doing it massively parallel. A decent actor framework can handle hundreds of parallel transactions or more.*

9 Stateless actors

The true power of virtual actors is that they are *stateful*. They can remember state between subsequent method invocations, which has some important advantages:

- Increased performance and responsiveness. Actors do not have to load their state from persistence on every request, but simply keep it in memory. They can respond faster to use requests, especially when they involve reading only, and when internal state is large in size.
- Reduced development time. Developers are used to working with objects that have state. And having state removed the burden of having to persist every piece of information after every request.

As an example, let's assume a forum post entity wants to keep a count of the number of times the post was viewed. Without state, this number would have to be persisted after every request. With state, this number can be kept in memory and only stored once per minute, for example. Or only every 100 views.

But there also can be a downside: You need a powerful actor framework.

With statefulness comes the responsibility of ensuring no more than 1 actor instance for a given type and id is active within the entire cluster. That not only requires a distributed locking mechanism that ensures that only one lock for one actor is assigned; it also requires quite complex client-side logic to ensure that actor instances will not perform operations without such a lock; that the lock is refreshed at regular times; that the actor is informed when the refresh fails; and that the lock is released when the actor is done.

A framework like Darlean provides all of this functionality, but it is currently available for TypeScript only, with ports for Go and .NET in progress.

9.1 AOA without a Framework

So, what's the alternative? Regardless of whether you are motivated by pureness (not willing to use an actor framework), curiosity, or other drive, I'll show you how to practice AOA without a full-fledged framework.

To practice AOA without a framework, you must give up on the following:

- Give up on actor state. Actors that persist their state between calls severely complicate matters.
- Give up on the ability to distribute your functionality over multiple applications that work together, because that would require a relatively complicated message bus, serialization and deserialization, and portals that act as proxies.

In other words,

- Actors must be stateless. They must fetch their state at the beginning of every request, and persist it at the end of every write request. But they must not store it in memory between requests.

- All actors and other functionality required by the actors to do their work (including controllers that accept incoming work) must be included in their own self-contained application, because there is no communication with other processes.

These constraints still provide scalability by scaling out horizontally with a load balancer in front.

For this to work, there is only one additional really important requirement:

- Actors must obtain a cluster-wide lock at the beginning of each method call; and release the cluster-wide lock after they are finished. Only within the lock are they allowed to perform work, including but not limited to fetching and persisting their state.

9.2 The shared lock: Implementation hints

So, how to best implement the shared lock?

For optimal performance, the coolest way would be to have a shared lock that does not require disk access. Darlean provides such a scalable, redundant distributed in-memory lock, but the whole point of using stateless actors is that, for some reason, we do not can or want to use such a framework.

We need a central place where we can achieve atomic operations to acquire or to release a lock.

Simplicity is the corner stone of AOA, and that also covers deployment. The less moving parts, the better. We could deploy a tool like Apache Zookeeper, with a thin object wrapper around it, but that would mean more complicated deployment. When that is fine with you, such an approach would be very viable.

Another approach is to use what we might already have. Let's assume that a relational database is already in use to persist state. We can combine persisting state with the shared lock functionality in the same database; provided that the relational database supports truly atomic operations, which most SQL databases do.

Let's define our LOCKING table as:

Column	Type	Description
ACTOR_TYPE	VARCHAR	Type of the actor
ACTOR_ID	VARCHAR	ID of the actor
LOCKED_BY	VARCHAR	Unique id of the component that manages the lock on behalf of an underlying actor. It could be a unique process name or unique id of the object the implements the lock.
EXPIRES	DATETIME	Timestamp before which the lock should be renewed (by updating the EXPIRES value) before it can be claimed by another party. This to prevent stalling when locks are not properly released due to technical issues.

We then leave it as an exercise to the reader to write an object that invokes an upsert query that atomically

- Acquire new lock: Adds a record when no record yet exists for a given ACTOR_TYPE and ACTOR_ID with LOCKED_BY set to its own unique identifier, and EXPIRES to a moment in the future (like 30 seconds away);
- Refresh own lock: Updates the EXPIRES time of a record when it already exists for the same ACTOR_TYPE and ACTOR_ID and LOCKED_BY;
- Take over expired lock: Updates a record when it exists for the same ACTOR_TYPE and ACTOR_ID and EXPIRES is before the current moment with our LOCKED_BY process name and a new EXPIRES moment;
- Does nothing otherwise. That means that another process has a valid lock. Our code should wait a while (presumably with exponential backoff) and try again.

> *In the chat example, we used a simple file system persistence instead of relation database persistence. So, why didn't we use that for our distributed lock?*
>
> *Well, it is tricky to rely on the locking capabilities of file systems, especially in environments where shared volumes are used. It is possible to perform atomic operations – but it would require quite some tricks to make that work, like creating a shadow file and then atomically renaming it to the proper name. But there are still all kinds of race conditions involved that one should properly manage and account for.*

9.3 Stateless actors

Stateless actors are responsible for their own locking. They must acquire and release the lock at the proper moments. For this, a lock should be injected:

```
export class MyStatelessActor {
  constructor( private lock: ActorLock ) {}
}
```

where ActorLock could be a function that repeatedly tries acquires an actor lock; throws an error when not possible; and returns an object that can be used to refresh or release the lock:

```
export type ActorLock = ()=> Promise<{
  refresh(): Promise<void>;
  release(): Promise<void>;
}>;
```

Actor methods now become as follows:

```
public doSomething() {
  const lock = await this.lock();
  try {
    // Do something useful
  } finally {
    await lock.release();
  }
}
```

9.4 Example: A stateless chat entity

As an example, we show stateless version of the ChatEntity with exactly the same functionality as its stateful counterpart.

Let's start with the skeleton:

```
export class ChatEntity {
  constructor(
    private id: ChatId;
    private lock: ActorLock;
    private fetch: () => Promise<ChatState | undefined>,
    private persist: (state: ChatState) => Promise<void>,
  ) {}

  public async create(): Promise<ChatInfo> {}

  public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {}

  public async getInfo(includeMessages: boolean): Promise<ChatInfo> {}

  private async checkExists(): ChatState {
    const state = await this.fetch();
    if (!state) {
        throw new Error('Chat does not exist');
    }
    return state;
  }

  private async checkNotExists() {
    if (!(await this.fetch())) {
        throw new Error('Chat already exists');
    }
  }
}
```

Differences with the stateful version are:

- Addition of a lock member
- Removal of state, fetched and ensureState()
- Removal of the @mutex decorators, because the shared lock already brings this functionality.

The stateless create method becomes:

```
  public async create(): Promise<ChatInfo> {
    const lock = await this.lock();
    try {
      await this.checkNotExists();
      this.state = ChatState.fromInfo(this.id);
      await this.persist(this.state);
      return this.mapToChatInfo(this.state);
    } finally {
      await lock.release();
    }
  }
```

The stateless getInfo method becomes:

```
public async getInfo(includeMessages: boolean): Promise<ChatInfo> {
  const lock = await this.lock();
  try {
    return this.mapToChatInfo(await this.checkExists(), includeMessages);
  } finally {
    await lock.release();
  }
}
```

In this simple case, we *could* technically have omitted the lock when we would know how our persistence works, and that it would always return consistent information.

However:

- Our simple file system persistence is not atomic. When a read occurs whilst a state is written only partially, the returned data is corrupt.
- When our state would be persisted in a database as multiple objects (like every ChatMessage its own record), the returned results would not have to be consistent when messages are added or removed in parallel.

> *Domain logic should not assume or depend on particular implementation details, to prevent issues like the previous bullets. Therefore, AOA recommends always to use a lock; even in the case of simple read operations.*
>
> *When actors are subject to mostly read method calls, a more advanced shared lock could be implemented that allows concurrent reads but exclusive writes.*

Finally, the stateless postMessage method:

```
public async postMessage(options: ChatMessagePostOptions): Promise<ChatMessagePostResult> {
  const lock = await this.lock();
  try {
    const state = this.checkExists();

    const message = ChatMessage.from({
      id: this.state.messages.length,
      chatId: this.state.id,
      content: options.content,
      sender: options.sender
    });

    this.state = this.state.withMessage(message);

    await this.persist(this.state);

    return ChatMessagePostResult.from({
      message: message,
      receivers: state.participants
    });
  } finally {
    lock.release();
  }
}
```

9.5 Using a decorator

With some additional software engineering, we can further optimize developer experience by making the locking part of a decorator. We can combine it with the functionality currently provided by the @mutex decorator. In fact, when we do have the shared lock in place, we do not need the mutex anymore.

This brings back the @mutex in the code, but eliminates the other boiler plate code:

```
@mutex(this.lock)
public async getInfo(includeMessages: boolean): Promise<ChatInfo> {
  return this.mapToChatInfo(await this.checkExists(), includeMessages);
}
```

10 Unit and integration testing

An important part of software engineering is testing. Domain Driven Design in its purest form is easy to test, because the domain logic is nicely isolated from the infra concerns.

At first glance, this might be more complicated with Actor Oriented Architecture, because we are not dealing with passive objects anymore, but with active objects, that have their own life cycle and infra interactions.

In practice, this is not true. When properly designed, AOC is easy to test. Why?

- The entities are in fact plain objects.
- When unit and integration testing, locking can be ignored. The `@mutex` decorators are still there, but simply do not do anything. Or, depending on the implementation, they still work, but that also do not stand in the way of testing.
- For testing, you do not need the 'global actor lock' that guarantees that within the cluster, there is no more than one instance. In a good implementation, the entities and usecases themselves do not depend on the global actor lock; it's the framework that must arrange all of that. For unit and integration testing, you do not need all of that; just instantiate the objects.
- For testing, you don't need advanced portals that perform RPC and the like. In fact, you even do not need the stubbing functionality. You can directly inject and invoke other instances just by pointer method calls.
- When the recommendations about infra abstractions are followed, it is trivial to stub infra adapters. Even persistence can easily be stubbed by a fake that simply stores state in memory.

To summarize: when the recommendations for AOA are followed, it is very simple to perform unit and integration testing.

Of course, I will provide an example to prove my wordings.

10.1 Example of a unit test

Let's start with a unit test that tests the ChatEntity.

```
// Arrange
const chatId = ChatId.from('12345');
const sender = ParticipantId.from('1234');

let chatState: ChatState | undefined = ChatState.fromInfo(chatId, [sender]);

const chat = new ChatEntity(
  /* id      */ chatId,
  /* fetch   */ async () => return chatState,
  /* persist */ async (state: ChatState) => chatState = state
);
```

Note how we provide fake implementations for fetch and persist that provide simplified but valid and working implementations. The persist simply copies the provided state into our variable chatState; and the fetch returns the current value for chatState. For this simple test, that only involves 1 entity instance, this is sufficient.

To ensure that we can invoke the postMessage during the Act phase, we must trick the entity that it has already been created (otherwise the entity will throw an error from checkExists within postMessage. We do that by assigning an initial value to chatState. The entity will fetch that value, see that it is defined, and assume it has been created before. Exactly as it would go in real life – but then the state would be set by first invoking the create method.

```
// Act
const result = await chat.postMessage(
  ChatMessagePostOptions.from({
    chatId: chatId,
    content: ChatContent.from('Hello world'),
    sender: sender
  })
);
```

The act simply posts a new message and awaits the result.

```
// Assert
expect(result.message.id.value.length).ToBeGreaterThan(0);

const infoWithMessages = await chat.getInfo(true);
expect(infoWithMessages.messages.length).toBe(1);
expect(infoWithMessage.messages.get(0).id).toBe(result.message.id);
```

The Assert begins to check that the resulting message id is not empty.

> *We could check for the exact expected value, using our knowledge about how the ChatEntity creates message id's, but that would tie our test so much to the implementation details of the ChatEntity that it would hinder future refactorings.*

Then, it asks for the list of messages, and checks that there is at least 1 message, and that the id of the first message (with index 0 in the array) is the same as the message it returned after postMessage.

10.2 Example of an integration test

We can apply the same unit testing approach for the ChatRelatedUsecases object by stubbing the retrieveChatEntity and retrieveParticipantEntity ports. We leave that as an exercise to the reader.

Instead, we will use the ChatRelatedUsecases object as an example for an integration test, which combines the ChatRelatedUsecases object with a *real* instance of the ChatEntity (we still stub the ParticipantEntity as we did not implement it yet in our example).

Let's start by defining a fake for the participant entity. The fake provides a valid implementation of a participant entity, but much simpler. The fake exposes a public member that can be modified or inspected from within the actual test code.

```
class FakeParticipantEntity implements IPartitipantEntity {
  public receivedAlerts: chatId[] = [];

  public addAlert(chatId: ChatId) {
    receivedAlerts.push(chatId);
  }
}
```

With the fake participant entity we can start writing our test. Let's start with the Arrange part.

```
//// Arrange

const chatId = ChatId.from('12345');
const sender = ParticipantId.from('1234');

// Create FAKE participant entity
const participant = new FakeParticipantEntity();

// Create REAL ChatEntity instance
let chatState: ChatState | undefined = ChatState.fromInfo(chatOd, [sender]);
const chat = new ChatEntity(
  /* id      */ chatId,
  /* fetch   */ async () => return chatState,
  /* persist */ async (state: ChatState) => chatState = state
);

// Create the REAL ChatRelatedUsecases instance
const usecases = new ChatRelatedUsecases.create(
  /* retrieveChatEntity        */ return (id) => chat,
  /* retrieveParticipantEntity */ return (id) => participant
);
```

```
//// Act

const result = await usecases.postMessage(
  ChatMessagePostOptions.from({
    chatId: chatId,
    content: ChatContent.from('Hello world'),
    sender: sender
  })
);
```

```
//// Assert

expect(result.message.id.value.length).ToBeGreaterThan(0);
expect(participant.receivedAlerts[0].value).toEqual('1234');
```

10.3 Reusing the composition root

These tests are still a bit verbose (especially the 'arrange' part), but that can be solved by moving parts of the code out to shared functions or shared objects. Or by reusing the existing composition root ChatSuiteBuilder and see what it brings.

Let's start by creating a repository fake that we can reuse between tests:

```
class FakeRepository<T> implements IRepository {
  constructor(private state?: T) {}

  public async fetch() { return this.state; }
  public async persist(value: T) { this.state = value; }
}
```

We can now rewrite the Arrange part by modifying the ChatEntityBuilder before we invoke build():

```
//// Arrange

const chatId = ChatId.from('12345');
const sender = ParticipantId.from('1234');

const chatRepo = new FakeRepositoty<ChatState>( ChatState.fromInfo(chatOd, [sender]) );
const participant = new FakeParticipantEntity();
const participantPortal = new SimpleLocalPortal( (id) => participant );

const builder = new ChatEntityBuilder('/data/chat');
builder.chatRepoRef = ref(chatRepo);
builder.participantPortalRef = ref(participantPortal);

const usecases = builder.build().chatRelatedUsecases;
```

In this code, we override the chat repository reference with our own fake chatRepo, and the participant portal reference with our own participantPortal that always returns our fake participant, regardless of the provided id.

In this simple system with only a few dependencies, we do not gain much from reusing our composite root. At least, not in terms of lines of code. For more complicated systems with more dependencies, it may help a lot. In addition to that, it ensures that the tests are as realistic and as close to the production setup as possible.

Fakes versus mocks and stubs

Note that our tests did not use any mocking / stubbing framework. We simply used code, local variables and fake implementations as test doubles. This choice follows from the core principle behind AOA: simplicity. Eliminate everything that is not strictly necessary.

In this situation, I have eliminated everything that a random developer might not know. A developer might not know the particular mocking framework I'd used, or it may not be familiar at all with such tooling.

The tests here are also following the principle of making things explicit. Any developer should be able to read and understand what's happening, without having to google the specifics of the mocking framework.

In addition to that, using fakes tends to make tests more robust against refactoring of the underlying software. To give an example, let's assume a box that counts how many balls have been thrown in the box. The box has a `hit(n: number)` method that adds n to the counter. Imagine a unit test that has a `Person` object (the system under test) make 6 hits in the box. The unit tests stubs the box, and checks that the 'hit' was invoked 1 time with value 6.

What if someone refactored the `Person` class, for whatever reason, and found it more convenient or performant or whatever not to call hit one time with 6 balls, but 3 times with 2 balls each.

The `Person` is still doing nothing wrong – but the test fails. Would the tester have used a fake box that simply keeps the total count, the test would still work.

In general, fakes make tests more stable, and more robust against refactoring. Which is exactly what tests are for: to make it safe to refactor because you have tests.

11 Authentication & authorization

Authentication and authorization are related topics that can easily pollute your code base. How to solve it is very project specific, and does not really belong to the scope of AOA.
Nevertheless, I'd like to provide some best practices that may work well in other situations.

11.1 Authentication

Authentication is the act of determining who a user is. How this is performed is typically highly coupled to the specific kind of technology that connects the user to the system. It could be by means of OpenID, by means of API token, by means of usernames and password, et cetera. It is usually best to have the authentication initiated by the controllers that interact with the user.

> *Actor frameworks provide a stateful environment. It is thus much easier to store session state server-side than it is for stateless environments. That also means that one could go back to the good old simple way of storing session information server-side, thus eliminating much the complexity that is associated with token-based authentication mechanisms like client-side OpenID.*

11.2 Authorization

Then the authorization part. Given that a controller knows who a user is. How do we manage access to our resources?

The advocated AOA pattern is that controllers invoke usecases (via portals). We must assume that we can trust the internals of our system (including message buses that connect the various components).

Building upon this pattern, we could add a principal object as additional argument to usecases. The principal is a value object, created server-side to avoid tampering, that contains the identity of the user and for example, which roles he has.

The usecase can then validate the principal against the requested operation. It could even invoke methods on other usecases or entities to find out whether access should be granted.

When access is granted, the usecase then invokes the underlying entities *without* the principal structure. This keeps the underlying entities clean in their interface and does not pollute the implementation. The burden of authorization is at the usecase.

> *This pattern is just a guideline. There could be complex scenario's in which it would not be feasible or practical to determine access rights from a use case. In those cases, it is of course possible to pass the principal to the individual entities.*

12 Key Take-Aways

The key take-aways of Actor Oriented Architecture are:

- Use screaming names to make the intent of your application clear.
    - Divide code in a *functional* and a *technical* part
        - Functional is what the customer cares about *how* it works
        - Technical is what the customer cares about *that* it works.
    - For the functional part,
        - Use functional names
        - No technical names
        - Entities, Usecases and Ports each in their own folders
- Entities are alive objects that are self-responsible for persistence and concurrency control.
    - They decide autonomously when to persist what, and which part should have which concurrency constraints.
    - They can interact with infra and other alive object through ports.
    - A framework should ensure that for every entity, there is never more than 1 instance active within the entire cluster.
    - Entities should only be interacted with through Portals. Never directly.
    - Entities should only be invoked from other entities within the same module; or from usecases within the same module. They are like implementation details that should not be exposed to the outside.
- Usecases are alive objects that could offer persistence and concurrency control, but they are typically not limited to only one instance and they typically do not contain state, so those is not applicable.
    - Usecases can invoke multiple entities and other usecases. Always through Portals.
    - Usecases shield entities from exposure to the outside. They hide the implementation details that entities form.
- Ports are interfaces that abstract infra services away from entities and usecases.
    - Ports are implemented by adapters.
    - Adapters are typically implemented as classes and reside within the *technical* compartment of the application.
- Value objects are immutable classes that carry data from one part of the system to the other.
    - Value objects should provide both functional and technical validation.
    - Functional validation relates to the domain constraints, like a name should not be empty or start with a capital.
    - Technical validation relates to technical constraints, like a maximum length of strings or blobs to protect system stability.
    - Value objects can also be used in the technical compartment of applications. Because of their functional and technical validation, they can even be used in the frontend.
- Portals connect the application code with active objects.
    - Portals decouple implementations
    - Portals in distributed systems typically provide some kind of RPC mechanism.

    - A portal is often implemented as a stub that exposes the same interface as the object it provides access to.
    - Together with an actor framework, a portal ensures that only one instance of an entity is active within the entire cluster. That is a difference with factories, that create a new instance every time. A portal can be thought of as a factory for stubs.
- Domain logic is primarily modeled in entities.
    - Entities play the role of aggregate root in DDD.
    - Entities guard the invariants and protect the integrity of its data, even in the case of concurrent access.
    - When implemented properly, the visible overhead of infra concerns is very small. It can usually be limited to a method decorator that indicates the kind of locking (exclusive or shared) and a one-liner every here and there to load or persist state.
    - This infra control is both a strength and a weakness. The weakness is that it reduces domain purity; the strength is that it greatly simplifies code that *does* need these interactions. Simply doing it inline is much clearer than artificially splitting up domain logic in separate steps and leaving the orchestration (which is also domain logic) to an arbitrary application service.
    - Entity actions are atomic. During processing of the action, no other parallel actions can take place or interfere with the action, provided that proper locking is applied (like exclusive locking for methods that perform write operations).
- Domain logic is secondarily modeled in usecases
    - Those parts of domain logic that do not naturally fit within an entity can be modeled by usecases.
    - Usecases are not atomic. The individual actions are atomic, but the flow as a whole could be interleaved by other requests.
    - Usecases shield entities from the outside. For every usecase that involves an entity, a corresponding usecase must exist. Application code must only call these usecases; not the underlying entities.
- Authentication and authorization can be modeled quite nicely.
    - Authentication is best performed by the controllers in the technical compartment of the application.
    - Authorization is best performed by the usecases. They can be provided with a second argument that reflects the principal information (user identity and roles, for example) that the usecase needs to provide authorization.
    - Entities normally should be free from authorization concerns. That should just focus on their domain logic, assuming that authorization is okay.
    - For really complicated situations, entities can of course do (part of the) authorization.

13 Summary

Actor Oriented Architecture (AOA) offers a refreshing new way of designing software applications. It combines the best practices of screaming architecture, clean architecture, domain driven design and actor oriented programming to form a unified set of patterns and best practices that gives hold to engineers that strive for a pure and clean design of their software, but are troubled by the implementation challenges that arise when a pure and clean design has to be combined with everyday reality of not so pure demands.

AOA mitigates the DDD trilemma by considering readability, understandability, testability and maintainability all even more important than achieving 100% purity. It provides the necessary patterns and best practices to do that, whilst compromising a little bit on purity to achieve a way of working that is both clean, pragmatic and fun.

It exploits the characteristics of actors (active objects) to combine pure logic with non-pure infra concerns in a natural way, understandable for both developers and interested customers. In combination with a suitable virtual actor framework, this results in scalable, highly redundant and concurrency-safe solutions with no additional work. Projects can start small, and when they evolve and gain traction, they can be scaled out gradually.

Especially when such frameworks also provide typical infra concerns like scalable persistence, timers and web gateways out of the box, and with minimal dependencies on the underlying infrastructure, developer productivity is conveniently raised to the next level as their focus shifts more and more to the actual domain logic the customer is paying for, and less and less to the peripheral matters like ORM's, databases, message buses, queues, API's, object mappers and the lethargic deployment and maintenance complexity that results from all of these.