Doggochain — A doggo game blockchain

2nd report

Guilherme Christopher Michaelsen Cardoso (14100831)
João Paulo Taylor Ienczak Zanette (14200743)

November 25, 2019

1 Contracts structure

This section will only describe contracts briefly just to clarify **what** they are (i.e. their role in the system). Further explanations of **how** they behave will be cleared in Section 2.

In short, the contracts directly related to Doggochain are structured as follows:

DoggoChain: The main contract, hence responsible for managing database in respect of Players and Doggos, as well as the smart-contract's operations (player challenging, hunting, breeding, etc.). Abstracts player's data as an internal `struct`, since there's no much use for it outside this contract[1].

Doggo: Works as a simple structure for every Doggo's stats, providing an interface that abstract how Doggos evolve (as in stat gaining) and what's their real current stats.

UsageTest: A unit test to check if the system works properly from the point of view of who's going to *use* the API.

Other implementation-specific contracts are designed to compensate some lack of features from Solidity 0.5, e.g. no proper dynamic array handling in storage data. Currently these contracts are:

DoggoList: A simple list data structure implementation, simulated using a mapping from `uint32` to `Doggo` (since there's no generics in Solidity, this implementation must be replicated for every different type) and a `length` variable for element counting.

OptionalPlayer: Actually not a contract itself, but a structure. Used to compensate the lack of Optional type[2] in Solidity. Needed since Solidity mappings assume *all* possible keys actually exist and maps to some value (meaning no `hasKey`-related checking), and it does not belong directly to `Player` structure since existence itself is not a player's property, but rather is a mapping's constraint.

Apart from that, there are libraries for the more complex operations, so the amount of responsibility in DoggoChain contract's code becomes reduced:

[1]The reason is that the `Player` structure is needed only for players to own their Doggos and mark their existence in the block-chain (i.e. in the game). Every other operation is done considering Doggo-to-Doggo interactions.

[2]An "Optional" is a type that handles existence/inexistence of data (without the security/usage issues of using `null`-like value).

Battle: For battle simulation. Its only public function is `simulate`, which gives a simulated battle's results.

Breed: For breeding operations. Its only public function is `breed`, which gives result containing the (possibly) generated Doggo and if the breeding was a success.

2 Function details

This section will be splitted in a subsection for each **main contract and libraries**, since other contracts are merely for implementation details.

2.1 DoggoChain

registerPlayer(string name): Registers a player in the blockchain, making sure the player hasn't been previously registered, and initializes their list of Doggos.

challenge(address target): Implements the rules for challenging a player for a Player versus Player battle.

breed(Doggo this, Doggo with): This function implements the rules for breeding two Doggos and spawning a new Doggo. The baby will have a combination of the hidden attributes of both parents.

trade(address to, string doggo_name): Implements the rules for trading a Doggo with a specific player.

hunt(doggo): Implements the rules for capturing a wild Doggo and adding it to the player's collection.

player(address addr): Retruns the name of the player, and their list of doggos.

totalPlayers(): Returns the number of registered players in the blockchain.

So to use any of the game's features, all the player have to do is to call each function from this contract. Each function will require that the player is correctly registered and will detect that using the `msg.sender` property as player's address.

For the `challenge`, `trade`, `breed` and `hunt`, the player will be informed of the **results** of that operation via function return data, with computations updating the storage internally. Since they're related to complex actions, some of their responsibilities are splitted into other libraries (e.g. `Battle` and `Breed`), so these functions will just call the libraries' ones, get the results and take decisions based on them.

2.2 DoggoList

This is a simple contract that implements a manual array implementation, which is necessary because Solidity doesn't implement struct copy from memory to storage.

push(Doggo doggo): Inserts a doggo to the end of the list.

pop(): Removes and returns the last doggo in the list.

isEmpty(): Returns true if the list is empty.

2.3 Doggo

Aside from the `DoggoChain` contract (which uses this contract just for storing purposes), some code units such as libraries (e.g. `Battle`) will make extensive use of the `Doggo` contract's functions. To understand how was it modelled, let's see this example: during simulation in a battle between two Doggos, called "first" and "second" for reference, the library may want to calculate the results of an physical attack action from "first" to "second". This can be achieved by doing the following:

1. The library calls `first.attack()` and holds its result;
2. Then, the library calls `second.defense()` and also holds its result;
3. Since the attack and defense values means, respectively, how much physical offensive and defensive power a Doggo has, the library can use a formula based on them to calculate a damage value, i.e. how much "second" will suffer with "first"s attack.
4. With that damage value calculated, the library will then apply that damage into "second" calling its own function `damage(target, amount)`, passing `second` as argument to `target` and the damage value for `amount`.
5. `damage(...)` will then decrease the current health points ("HP" for short), i.e. how much that doggo can still keep up with the battle, which is controlled outside "Doggo" contract using a separate struct `Battler` that holds both the "Doggo" and its current health. This way, battle can be handled without ever changing the contract's storage (except when applying battle simulation results's side-effects, e.g. giving a Doggo its experience points — potentially leveling him up).

That should illustrate the idea of this contract's modelling: do as few writes as possible to the storage (leaving it only for more "permanent" data) and let the more complex operations built atop the view-only functions.

2.4 Battle

This library handles the combat system of the game. A battle is always done between two Battlers. A Battler is a doggo and it's associated current health points. When those health points reach zero, the doggo faints and loses the battle.

The result of a battle can either be victory or defeat. The battlers are the challenger, and the guest (the doggo who has been challenged).

This library contains the following functions:

simulate(challenger, guest) : This function receives both doggos as parameters, and implements the battle loop, until a result is reached, and then returns that result.

createBattleData(challenger, guest) : This function prepares both combatants to fight by setting their current health points to their maximum health points.

isDead(battler) : This function tests if the doggo has zero health points.

damage(batler, ammount) : This function receives a pre-calculated ammount of damage as a parameter and subtracts it from the battler's current health (or setting it to zero if the damage received is greater than it's current health).

heal(battler, ammount) : Similar to the damage function, but it heals the battler by adding a pre-calculated ammount of health to their current health (checking if it's not greater than it's maximum health).

3 Diagrams

For a more top-level visualization of the contracts's schema, Figure 1 shows a Class Diagram for the system as a whole, while Figures 2 and 3 shows Sequence Diagrams for the main complex transactions.

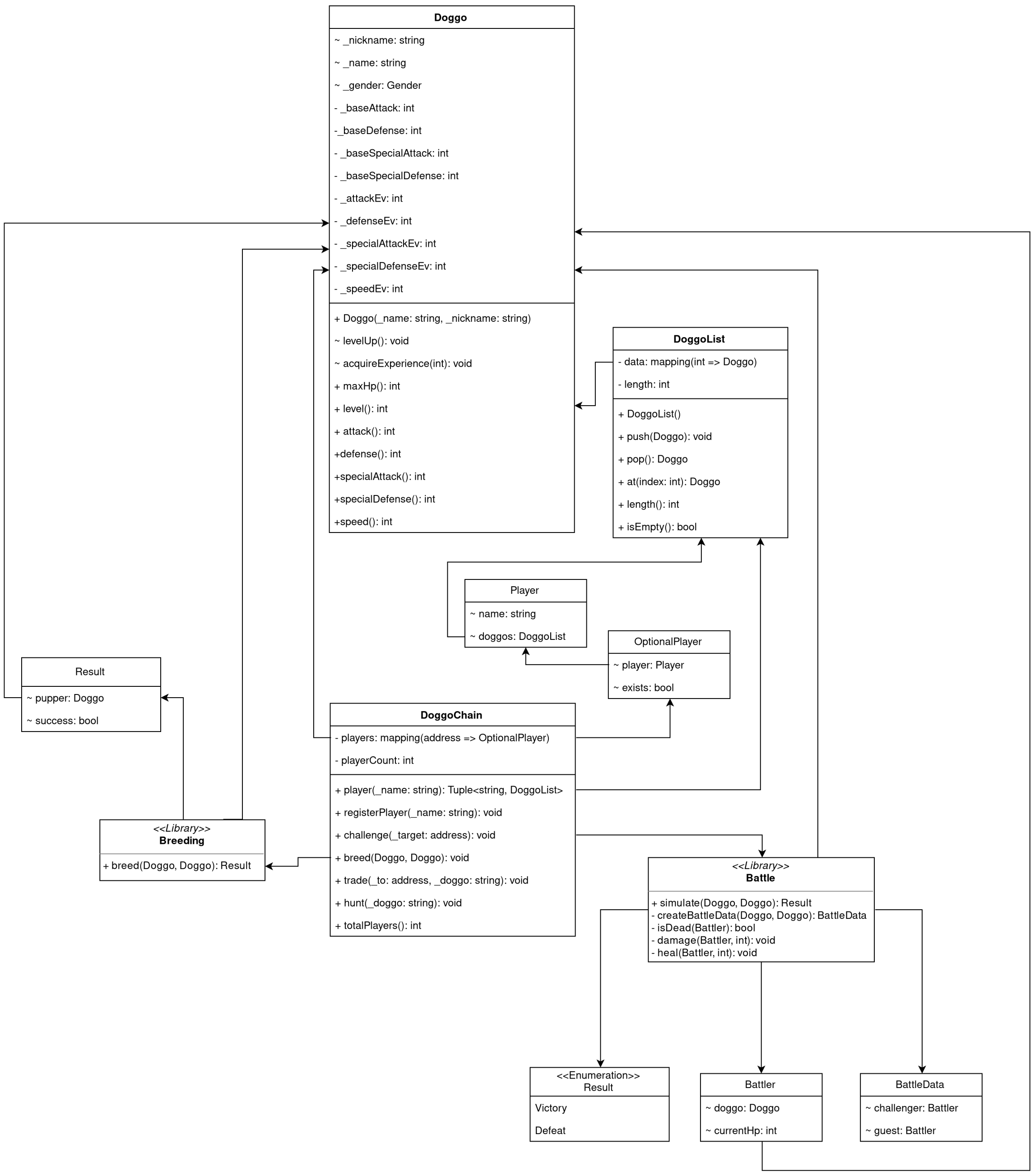

Figure 1: Class Diagram for DoggoChain system.

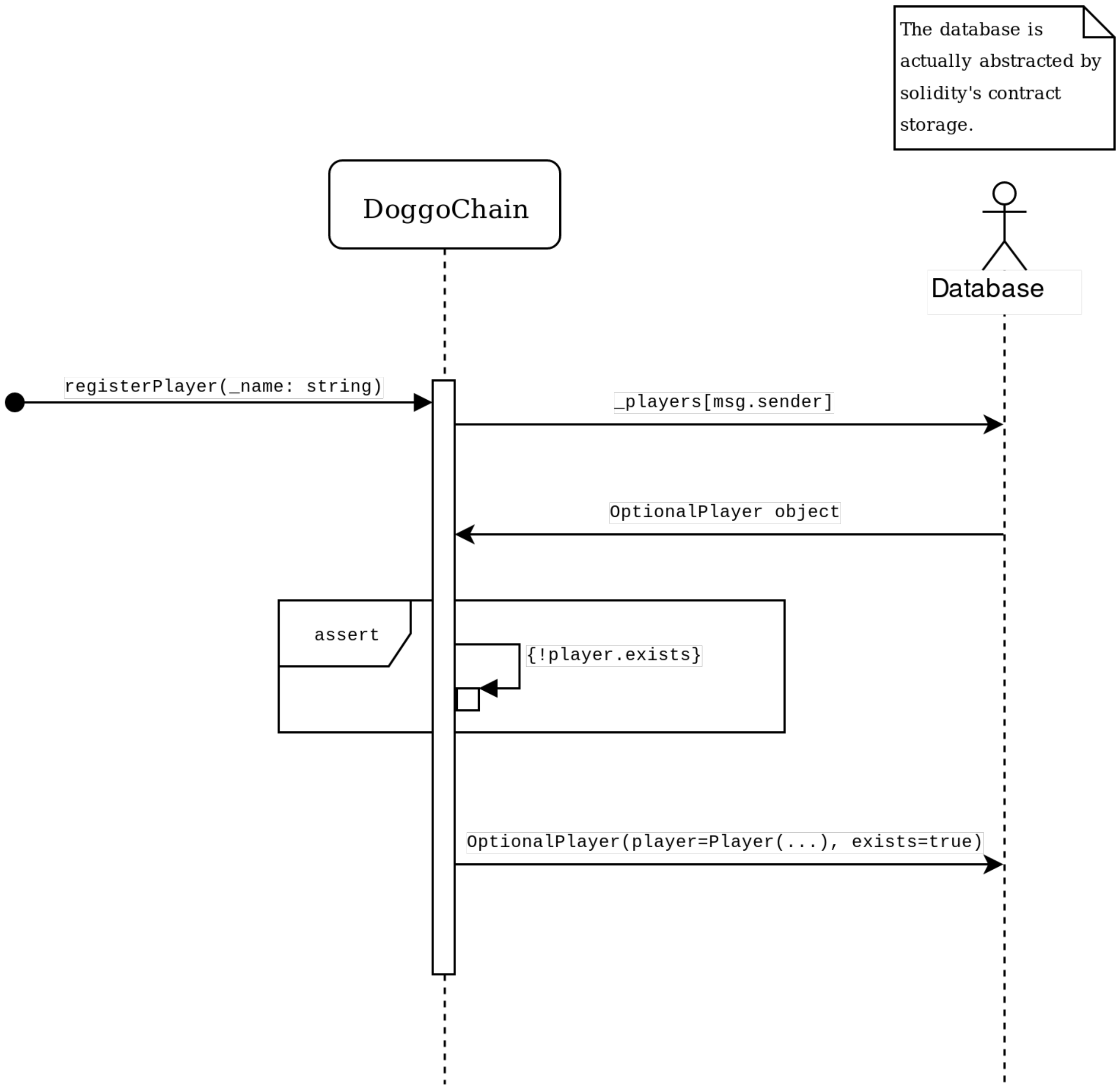
The database is actually abstracted by solidity's contract storage.
DoggoChain
Database
registerPlayer(_name: string)
_players[msg.sender]
OptionalPlayer object
assert
{!player.exists}
OptionalPlayer(player=Player(...), exists=true)

Figure 2: Sequence Diagram for `DoggoChain.registerPlayer` function.

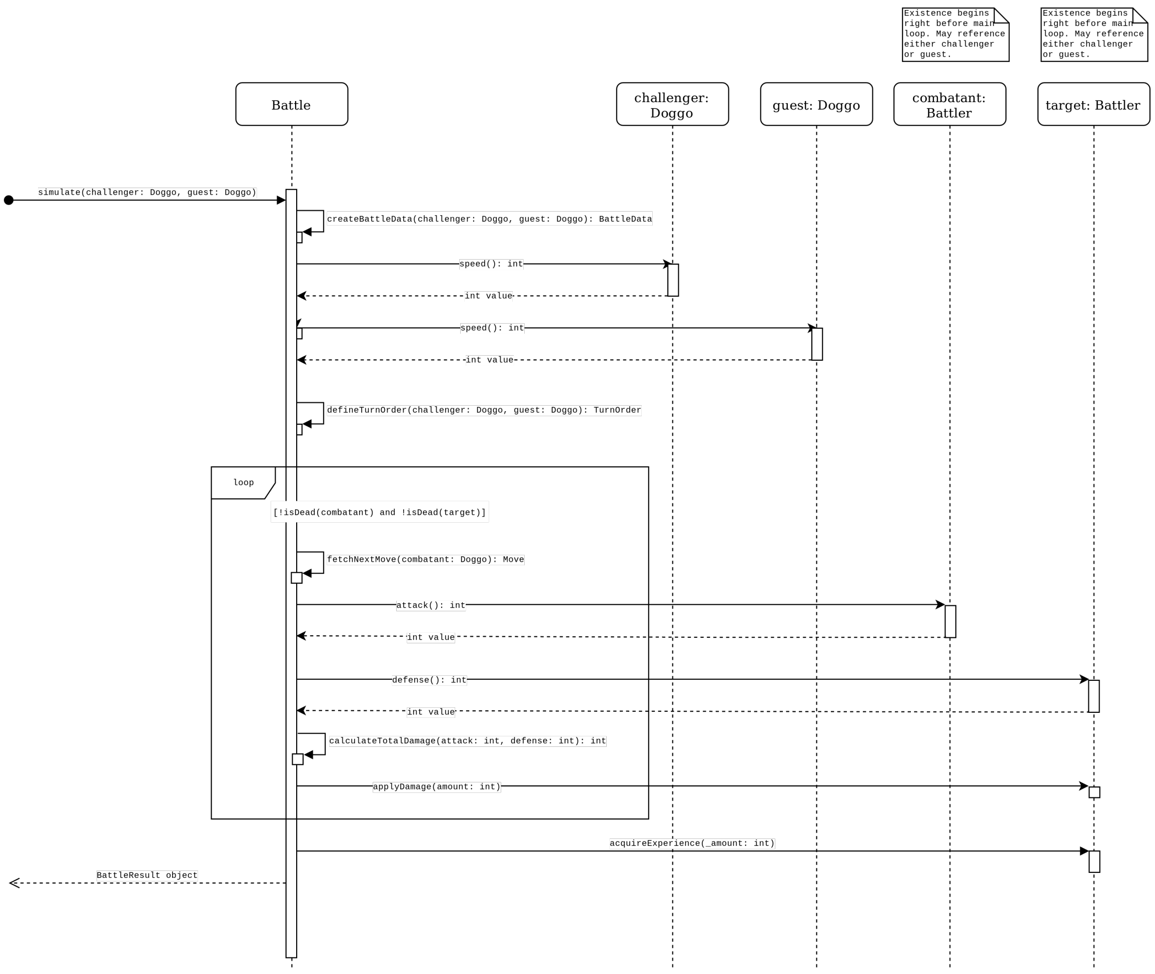
Existence begins right before main loop. May reference either challenger or guest.
Existence begins right before main loop. May reference either challenger or guest.
Battle
challenger: Doggo
guest: Doggo
combatant: Battler
target: Battler
simulate(challenger: Doggo, guest: Doggo)
createBattleData(challenger: Doggo, guest: Doggo): BattleData
speed(): int
int value
speed(): int
int value
defineTurnOrder(challenger: Doggo, guest: Doggo): TurnOrder
loop
[!isDead(combatant) and !isDead(target)]
fetchNextMove(combatant: Doggo): Move
attack(): int
int value
defense(): int
int value
calculateTotalDamage(attack: int, defense: int): int
applyDamage(amount: int)
acquireExperience(_amount: int)
BattleResult object

Figure 3: Sequence Diagram for `Battle.simulate` function.