**Lesson 7 : The Tamagotchi Battle Logic & Implementation**

**Lesson Summary**

* This lesson focuses on the contract that facilitates battles between two Tamagotchis.
* The battle contract can be in one of four states: Registration, Moves, Waiting, and GameIsOver.
* The Battle struct contains information about the players, the current state of the game, the current turn, the Tamagotchi store ID, the winner, and the number of steps taken so far.
* To participate in the game, users must allow their contracts to receive game-related messages like TmgAction::TmgInfo and StoreAction::GetAttributes.
* The register function obtains information about the Tamagotchi owner and its attributes from the shop, generates the Tamagotchi's power and energy randomly, and registers the Tamagotchi.
* The get\_owner and get\_attributes functions retrieve information about the Tamagotchi's owner and attributes from the store, respectively, while the get\_turn and generate\_power functions generate pseudo-random numbers to determine who starts the game and the Tamagotchi's power, respectively.

**Lesson Objectives**

By the end of the lesson, you will:

* Understand the contract that facilitates battles between two Tamagotchi.
* Understand the information contained in the Battle struct, such as player information, game state, winner, and number of steps taken so far.
* Comprehend the battle contract states and understand their functions
* Know the functions used to retrieve information from the store and generate pseudo-random numbers, such as get\_owner, get\_attributes, get\_turn, and generate\_power.
* Complete the simple realization of the battle between Tamagotchi

**Let’s get started!**

The battle contract can be in 3 states:

* *Registration*: the battle contract is waiting for the Tamagotchi owner to register them;
* *Move*: the Tamagotchi owners are making the move in turns;
* *Waiting*: after the Tamagotchi owners made their moves, the battle contract gives them time to equip their Tamagotchi.
* *GameIsOver*: the battle is over, it’s necessary to send a message *StartNewGame*.

| enum **BattleState** {  Registration,  Moves,  Waiting,  GameIsOver, }  impl Default for BattleState {  fn default() -> Self {  BattleState::Registration  } } |
| --- |

The battle program state:

| *#[derive(Default)]* pub struct Battle {  players: Vec<Player>,  state: BattleState,  current\_turn: u8,  tmg\_store\_id: ActorId,  winner: ActorId,  steps: u8,  } |
| --- |

Where the players are the following structs:

| **#[derive(Default)]** pub struct Player {  owner: ActorId,  tmg\_id: TamagotchiId,  energy: u16,  power: u16,  attributes: BTreeSet<AttributeId>, } |
| --- |

To participate, users must allow their contracts to receive game-related messages as below:

| TmgAction::TmgInfo |
| --- |

To which it will respond with information about tamagotchi owner:

| TmgEvent::Owner(ActorId) |
| --- |

We’ll also add the message

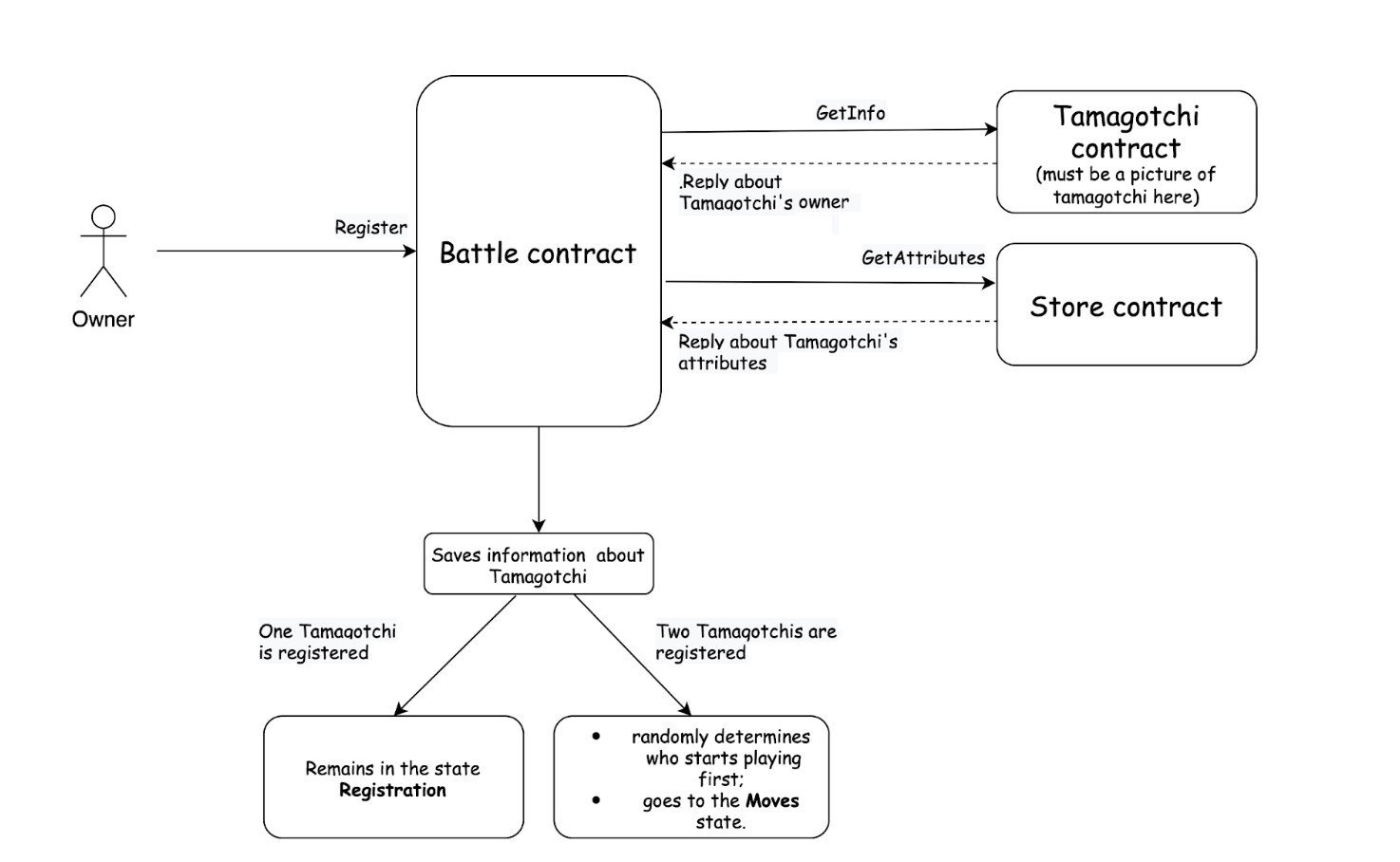
| StoreAction::GetAttributes |
| --- |

To the store contract that allows us to get attributes of Tamagotchi

| StoreEvent::Attributes {  attributes: BTreeSet<AttributeId> } |
| --- |

The register function:

It allows the registration of two Tamagotchi for battle.



1. Before you register a Tamagotchi, the Battle contract must receive the Tamagotchi’s owner and its attributes from the shop;
2. After receiving the details, the Battle contract randomly generates the Tamagorchi’s power and energy;
3. If one Tamagotchi is registered, it remains in the Registration state;
4. If two Tamagotchi are registered, the battle contract randomly determines who starts playing first and goes to the Moves state.

| async fn **register**(&mut self, tmg\_id: &TamagotchiId) {  assert\_eq!(  self.state,  BattleState::Registration,  "The game has already started"  );   let owner = get\_owner(tmg\_id).await;  let attributes = get\_attributes(&self.tmg\_store\_id, tmg\_id).await;  let power = generate\_power();  let power = MAX\_power - power;  let player = Player {  owner,  tmg\_id: \*tmg\_id,  energy,  power,  attributes,  };  self.players.push(player);  if self.players.len() == 2 {  self.current\_turn = get\_turn();  self.state = BattleState::Moves;  }  msg::reply(BattleEvent::Registered { tmg\_id: \*tmg\_id }, 0)  .expect("Error during a reply `BattleEvent::Registered");  } |
| --- |

The *get\_owner* function retrieves the Tamagotchi's owner from the Tamagotchi contract.

| pub async fn **get\_owner**(tmg\_id: &ActorId) -> ActorId {  let reply: TmgEvent = msg::send\_for\_reply\_as(\*tmg\_id, TmgAction::Owner, 0)  .expect("Error in sending a message `TmgAction::Owner")  .await  .expect("Unable to decode TmgEvent");  if let TmgEvent::Owner(owner) = reply {  owner  } else {  panic!("Wrong received message");  } } |
| --- |

Similarly, the *get\_attributes* function retrieves the Tamagotchi's attributes from the shop.

| async fn **get\_attributes**(tmg\_store\_id: &ActorId, tmg\_id: &TamagotchiId) -> BTreeSet<AttributeId> {  let reply: StoreEvent = msg::send\_for\_reply\_as(  \*tmg\_store\_id,  StoreAction::GetAttributes {  Tamagotchi\_id: \*tmg\_id,  },  0,  )  .expect("Error in sending a message `StoreAction::GetAttributes")  .await  .expect("Unable to decode `StoreEvent`");  if let StoreEvent::Attributes { attributes } = reply {  attributes  } else {  panic!("Wrong received message");  } } |
| --- |

To determine which player starts the game, we use the *get\_turn* function, which pseudorandomly selects the starting player :

| pub fn **get\_turn**() -> u8 {  let random\_input: [u8; 32] = array::from\_fn(|i| i as u8 + 1);  let (random, \_) = exec::random(random\_input).expect("Error in getting random number");  random[0] % 2 } |
| --- |

The *genetate\_power* function pseudorandomly generates a power value for the Tamagotchi:

| pub fn **genetate\_power**() -> u16 {  let random\_input: [u8; 32] = array::from\_fn(|i| i as u8 + 1);  let (random, \_) = exec::random(random\_input).expect("Error in getting random number");  let bytes: [u8; 2] = [random[0], random[1]];  let random\_power: u16 = u16::from\_be\_bytes(bytes) % MAX\_POWER;  if random\_power < MIN\_POWER {  return MAX\_POWER / 2;  }  random\_power } |
| --- |

There are also two constants, MAX\_POWER and MIN\_POWER, that define the upper and lower bounds of the Tamagotchi's power:

| const MAX\_POWER: u16 = 10\_000; const MIN\_POWER: u16 = 3\_000; |
| --- |

Next, as an example, we will define a very simple game mechanic:

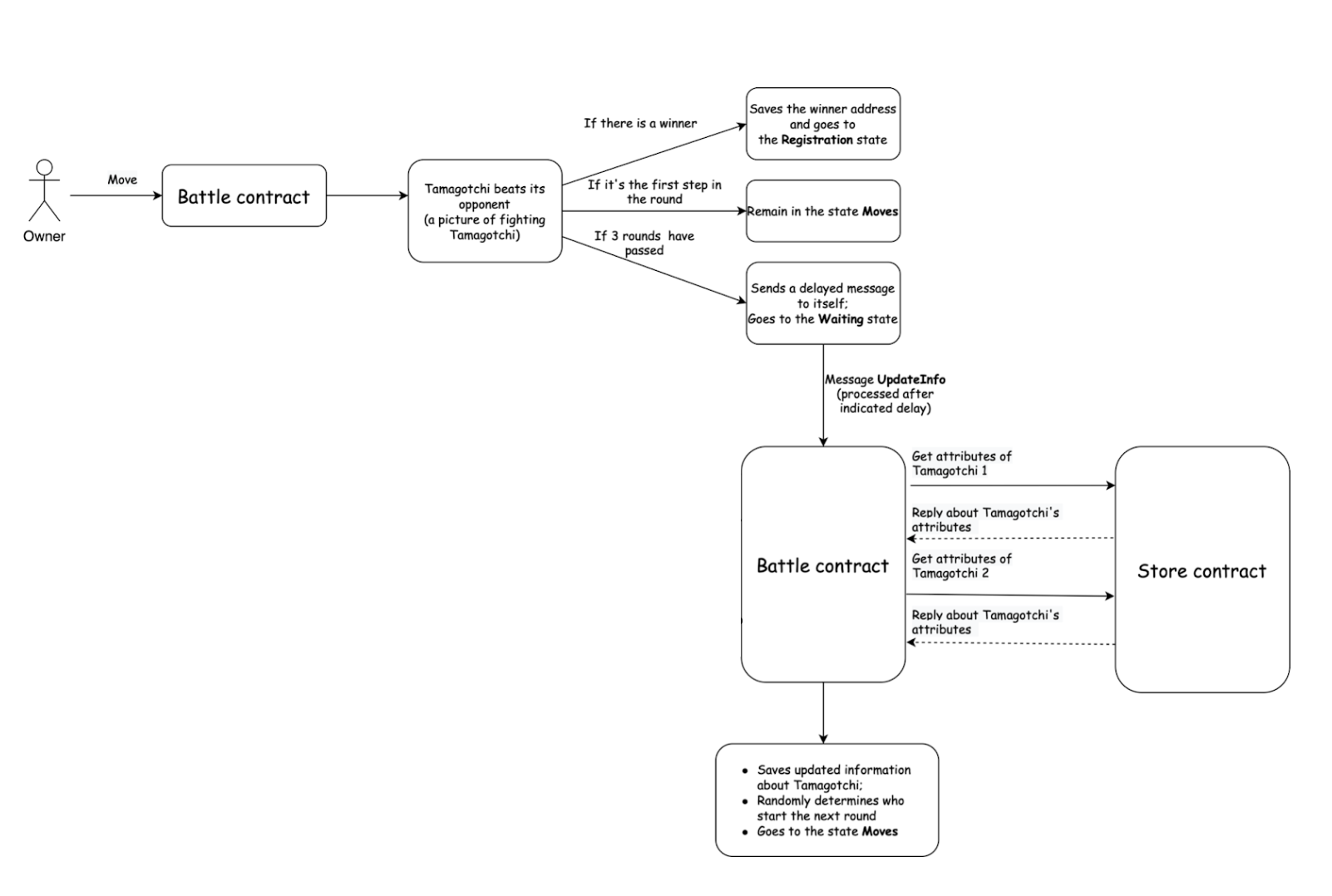
1. The Tamagotchi owner makes a move by simply sending a message *BattleAction::Move* to the battle contract. During that move, their Tamagotchi beats the opponent’s Tamagotchi. The opponent’s energy decreases by the force of its strike.
2. Now, there is only one attribute in the Tamagotchi store that can be used in the game - a sword. If the attacking Tamagotchi has a sword, the force of its strike is multiplied by SWORD\_POWER:

SWORD\_POWER \* power

Otherwise Tamagochi's strike power is simply their power.

In the future, you can expand the logic by adding fighting attributes to the store, as well as adding the movement of the Tamagotchi across the field.

1. If both players have made three moves, the game enters a waiting state, and the players can equip their Tamagotchis with items from the shop. After a set delay, the Tamagotchis' states are updated, and the next round begins.



| fn **make\_move**(&mut self) {  assert\_eq!(  self.state,  BattleState::Moves,  "The game is not in `Moves` state"  );  let turn = self.current\_turn as usize;  let next\_turn = (( turn + 1 ) % 2)as usize;  let player = self.players[turn].clone();  assert\_eq!(  player.owner,  msg::source(),  "You are not in the game or it is not your turn"  );  let mut opponent = self.players[next\_turn].clone();  let sword\_power = if player.attributes.contains(&SWORD\_ID) {  SWORD\_POWER  } else {  1  };   opponent.energy = opponent.energy.saturating\_sub(sword\_power \* player.power);  self.players[next\_turn] = opponent.clone();  *// check if opponent lost*  if opponent.energy == 0 {  self.players = Vec::new();  self.state = BattleState::GameIsOver;  self.winner = player.tmg\_id;  msg::reply(BattleEvent::GameIsOver, 0)  .expect("Error in sending a reply `BattleEvent::GameIsOver`");  return;  }  if self.steps <= MAX\_STEPS\_FOR\_ROUND {  self.steps += 1;  self.current\_turn = next\_turn as u8;  msg::reply(BattleEvent::MoveMade, 0)  .expect("Error in sending a reply `BattleEvent::MoveMade`");  } else {  self.state = BattleState::Waiting;  self.steps = 0;  msg::send\_with\_gas\_delayed(  exec::program\_id(),  BattleAction::UpdateInfo,  GAS\_AMOUNT,  0,  TIME\_FOR\_UPDATE,  )  .expect("Error in sending a delayed message `BattleAction::UpdateInfo`");  msg::reply(BattleEvent::GoToWaitingState, 0)  .expect("Error in sending a reply `BattleEvent::MoveMade`");  }  } |
| --- |

The **UpdateInfo** action just updates the changes in Tamagotchi states and starts the next round:

| async fn **update\_info**(&mut self) {  assert\_eq!(  msg::source(),  exec::program\_id(),  "Only contract itself can call that action"  );  assert\_eq!(  self.state,  BattleState::Waiting,  "The contract must be in `Waiting` state"  );   for i in 0..2 {  let player = &mut self.players[i];  let attributes = get\_attributes(&self.tmg\_store\_id, &player.tmg\_id).await;  player.attributes = attributes;  }  self.state = BattleState::Moves;  self.current\_turn = get\_turn();  msg::reply(BattleEvent::InfoUpdated, 0)  .expect("Error during a reply `BattleEvent::InfoUpdated");  } |
| --- |

So, we have finished the simple realization of the battle between Tamagotchi.

You can try to take part in the battle with your Tamagochi or create some Tamagotchi (using the previous lesson) and make them fight (Link to web application).

Now it’s time for your coursework.

**Coursework:**

* Add more fighting attributes to the store contract;
* Determine the power of these attributes. You can also add the opportunity to upgrade various attributes (That will also require extending the logic of the store contract);
* Allow the player to choose which weapon he will fight or defend during the turn;
* It is also possible to add the ability to Tamagotchi to move left or right;
* Be creative and think about other possible ways to make the game more interesting.