**Lesson 6: Implementing the Program Factory for Multiple Escrow Smart Contracts**

**Lesson Summary**

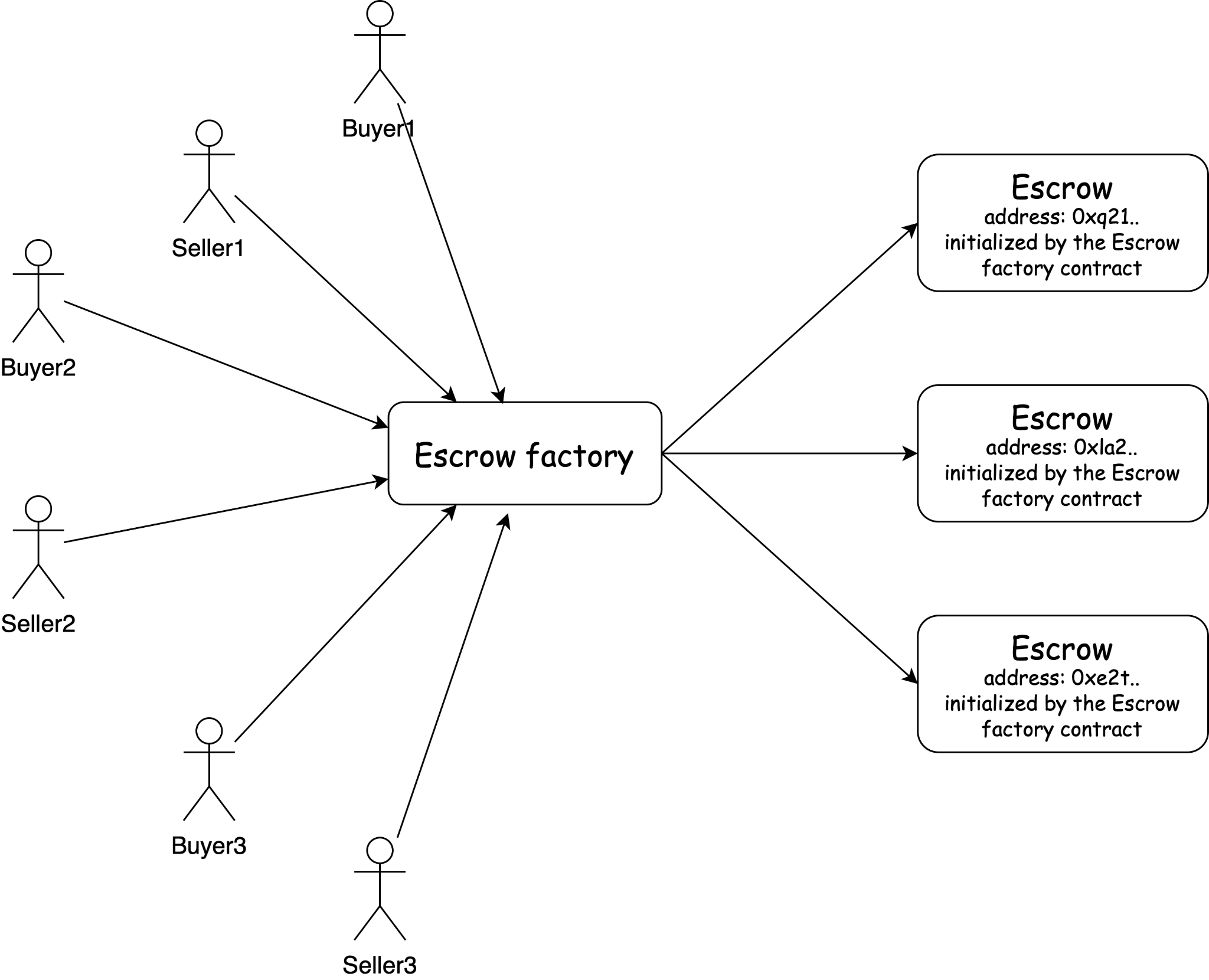
* In the previous **(2nd)** tutorial, we created an escrow smart contract that facilitated an agreement between two parties.
* However, the escrow contract became useless after the deal was over and had to be initialized again for new deals with new parties.
* In this tutorial, we’ll learn how to write an Escrow Factory smart contract that creates multiple instances of the escrow contract template from the previous tutorial.
* The Escrow Factory smart contract eliminates the need to upload and initialize the same escrow contract template for every new deal.
* Finally, we’ll test the Escrow Factory smart contract by deploying it to the blockchain and creating multiple escrow contract instances using the factory.

**Lesson Objectives**

By the end of this lesson, you should:

* Understand how to create a factory smart contract
* Explain how to initialize a new contract instance using a factory contract
* Demonstrate how to test a factory smart contract
* Understand the concept of a factory contract and how it can be used to deploy new instances of a contract.
* Understand how to interact with the Escrow Factory contract to create new instances of the Escrow contract for different parties.

**Let’s get started!**



**Coding Practice to Create Escrow Factory**

Our Escrow Factory will store the number of created escrow contracts, the mapping from the escrow id to its program address, and also the **CodeId** of the escrow smart contract. :

| **#![no\_std]** use gstd::{msg, prelude::\*, ActorId, CodeId}; pub type EscrowId = u64;  **#[derive(Default)]** pub struct **EscrowFactory** {  pub escrow\_number: EscrowId,  pub id\_to\_address: BTreeMap<EscrowId, ActorId>,  pub escrow\_code\_id: CodeId, } static mut ESCROW\_FACTORY: Option<EscrowFactory> = None;  **#[gstd::async\_main]** async fn **main**() {}  **#[no\_mangle]** unsafe extern "C" fn **init**() {  let escrow\_code\_id: CodeId = msg::load().expect("Unable to decode CodeId of the Escrow program");  let escrow\_factory = EscrowFactory {  escrow\_code\_id,  ..Default::default()  };  ESCROW\_FACTORY = Some(escrow\_factory); } |
| --- |

The **CodeId** is a hash of the escrow program uploaded into the chain. That hash will be used to create instances of escrow smart contracts.

Let's define the functionality of our loan factory program. It will deploy an escrow contract and send messages about deposit and delivery confirmation to the escrow.

| #[derive(Encode, Decode, TypeInfo)] pub enum **FactoryAction** {  CreateEscrow {  seller: ActorId,  buyer: ActorId,  price: u128,  },  Deposit(EscrowId),  ConfirmDelivery(EscrowId), }  #[derive(Encode, Decode, TypeInfo)] pub enum **FactoryEvent** {  EscrowCreated {  escrow\_id: EscrowId,  escrow\_address: ActorId,  },  Deposited(EscrowId),  DeliveryConfirmed(EscrowId), } |
| --- |

As you can see, the Escrow contract will interact with Buyer and Seller through Escrow Factory contract, meaning the Escrow Factory contract will send messages to the Escrow contract.

Firstly, we have to define an io crate for the Escrow contract. Then we’ll modify the structure of incoming messages and Escrow methods. Try to change it yourself and then compare it with the correct implementation (link).

After that, we’ll define Loan Factory methods and write the **handle** function:

| impl EscrowFactory {  async fn **create\_escrow**(&mut self, seller: &ActorId, buyer: &ActorId, price: u128) {}  async fn **deposit**(&self, escrow\_id: EscrowId) {}  async fn **confirm\_delivery**(&self, escrow\_id: EscrowId) {} } **#[gstd::async\_main]** async fn **main**() {  let action: FactoryAction = msg::load().expect("Unable to decode `FactoryAction`");  let factory = unsafe { ESCROW\_FACTORY.get\_or\_insert(Default::default()) };  match action {  FactoryAction::CreateEscrow {  seller,  buyer,  price,  } => factory.create\_escrow(&seller, &buyer, price).await,  FactoryAction::Deposit(escrow\_id) => factory.deposit(escrow\_id).await,  FactoryAction::ConfirmDelivery(escrow\_id) => factory.confirm\_delivery(escrow\_id).await,  } } |
| --- |

Let’s implement the **create\_escrow** function.

For the program deployment we should import **ProgramGenerator** from the **prog** module in **gstd** library:

| use gstd::{msg, prelude::\*, ActorId, prog::ProgramGenerator, CodeHash}; |
| --- |

To create a new contract instance, we will use the **create\_program\_with\_gas\_for\_reply** function. Here are the required parameters:

* The code hash of the uploaded program code
* Payload for initialization message
* Gas for the program creation (calculate in advance how much the initialization of the program loaded on the network requires)
* Value attached to the init message

| async fn **create\_escrow**(&mut self, seller: &ActorId, buyer: &ActorId, price: u128) {  let (address, \_) = ProgramGenerator::create\_program\_with\_gas\_for\_reply(  self.escrow\_code\_id,  InitEscrow {  seller: \*seller,  buyer: \*buyer,  price,  }  .encode(),  GAS\_FOR\_CREATION,  0,  )  .expect("Error during Escrow program initialization")  .await  .expect("Program was not initialized");  self.escrow\_number = self.escrow\_number.saturating\_add(1);  self.id\_to\_address.insert(self.escrow\_number, address);  msg::reply(  FactoryEvent::EscrowCreated {  escrow\_id: self.escrow\_number,  escrow\_address: address,  },  0,  )  .expect("Error during a reply `FactoryEvent::ProgramCreated`");  } |
| --- |

In our Escrow factory smart contract, we use asynchronous program creation to ensure the program is initialized without errors. Since the factory program waits for a reply, we add a reply message to the program initialization.

Other methods are implemented easily since all logic and all checks are included in the Escrow contract:

| async fn **deposit**(&self, escrow\_id: EscrowId) {  let escrow\_address = self.get\_escrow\_address(escrow\_id);  send\_message(&escrow\_address, EscrowAction::Deposit(msg::source())).await;  msg::reply(FactoryEvent::Deposited(escrow\_id), 0)  .expect("Error during a reply `FactoryEvent::Deposited`");  }  async fn **confirm\_delivery**(&self, escrow\_id: EscrowId) {  let escrow\_address = self.get\_escrow\_address(escrow\_id);  send\_message(&escrow\_address, EscrowAction::ConfirmDelivery(msg::source())).await;  msg::reply(FactoryEvent::DeliveryConfirmed(escrow\_id), 0)  .expect("Error during a reply `FactoryEvent::DeliveryConfirmed`");  }   fn **get\_escrow\_address**(&self, escrow\_id: EscrowId) -> ActorId {  \*self  .id\_to\_address  .get(&escrow\_id)  .expect("The escrow with indicated id does not exist")  } |
| --- |

We move the **msg::send\_for\_reply\_as** to a separate function to send messages to the Escrow program for better readability.

| async fn send\_message(escrow\_address: &ActorId, escrow\_payload: EscrowAction) {  msg::send\_for\_reply\_as::<\_, EscrowEvent>(\*escrow\_address, escrow\_payload, msg::value())  .expect("Error during a sending message to a Escrow program")  .await  .expect("Unable to decode EscrowEvent"); } |
| --- |

With the factory loan contract finished, we’ll now test our factory contract.

**Testing the Escrow Factory Functionality**

Before testing the Escrow Factory smart contract, we need to set up the environment. Here's how:

* Upload the code of the Escrow contract:

| use gtest::{Program, System};  **#[test]** fn init\_escrow\_factory() {  **let** **system** = System::new();  let escrow\_code\_id = system.submit\_code("./escrow/target/wasm32-unknown-unknown/release/escrow.opt.wasm");  let escrow\_factory = Program::current(&system);  let res = escrow\_factory.send(100, escrow\_code\_id);  assert!(!res.main\_failed());  assert!(res.log().is\_empty()); } |
| --- |

Continue to test the contract as you learnt in previous lessons.

**Assignment**:

* Finish tests for the Escrow factory;
* Write the contract that will create Tamagotchi from your contract template.

