Assignment 3 – Smart contract analysis

COS30049 – Computing Technology Innovation Project

Group 1.5

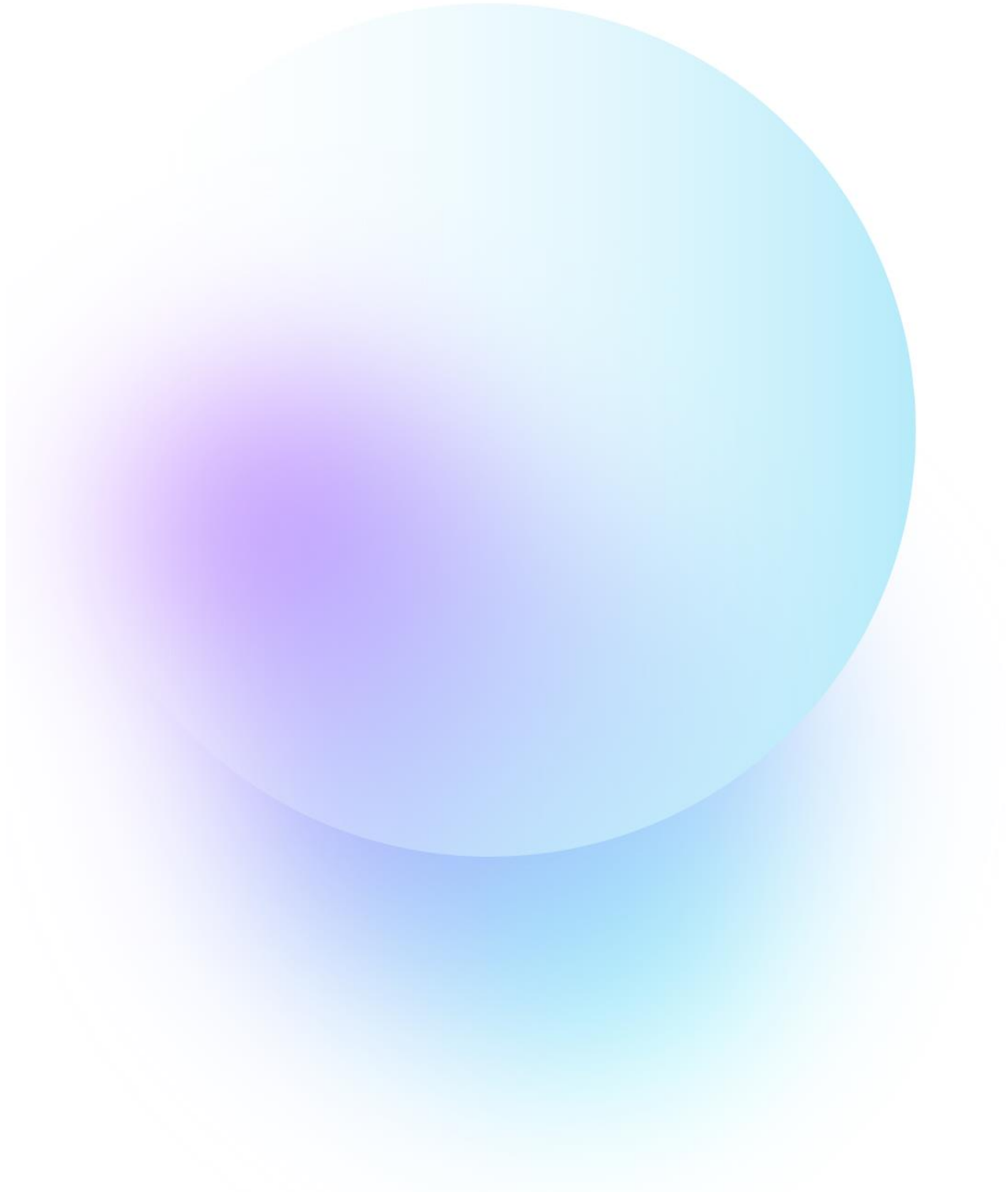

Table of Contents

Project information

Project name and description

Our project, Dripple, aims to give consumers access to a decentralized marketplace for virtual fashion transactions. Users can easily and securely create, trade, and collect NFTs with others on the platform. To achieve this, the team has undertaken the development of a full-stack decentralized web application which works in conjunction with a private blockchain system. The trades on the platform are governed by a smart contract which implements the ERC 721 standard for NFTs. Key functionalities of the website include:

- **Authenticating users, profile viewing and management:** Each user on the platform will be given a unique account that they can use to trade with other users. In their account, the user can see their wallet balance, the NFTs they have collected and traded, as well as their transaction history.
- **Sending and purchasing coins for trading:** Users can purchase coins for trading on the platform or send coins to other users. These transactions will be recorded into their transaction histories.
- **Minting and uploading NFTs to the market:** Users can securely create new assets on the platform and upload them to the market for trading.
- **Browsing the marketplace and purchasing items:** Users are provided with a friendly user interface to search for and buy specific assets of interest on the marketplace.

While some of the above functionalities can be achieved without smart contracts, others necessitate a smart contract on the blockchain. The contract used by our project will address three functionalities: ***minting NFTs, purchasing NFTs***, and ***viewing collected NFTs***.

Team information

Our team consists of 6 people, which are:

- **Ta Quang Tung**: Tung is our team leader who is responsible for developing, testing and deploying smart contracts, designing the system architecture,

designing the database, and creating the API. In this assignment, he works on the ABI illustration section.

- **Nguyen Quang Huy**: Huy writes the frontend and backend of the marketplace page. In this assignment, he works on the call graph section.
- **Phan Sy Tuan**: Tuan writes the header and footer components and backend for minting and uploading NFTs. In this assignment, he works with Sang on the inheritance graph discussion.
- **Tran Hoang Hai Anh**: Hai Anh writes the frontend of the item information page and backend for purchasing NFTs. In this assignment, he works on the project introduction section.
- **Vu Xuan Sang**: Sang writes the frontend for the profile page and backend for retrieving and uploading NFTs from the database and buying coins. In this assignment, he works with Tuan on the inheritance graph discussion.
- **Nguyen My Hanh**: Hanh writes the frontend for the user profile page and backend for retrieving the transaction history from the database. In this assignment, she works on the executive summary section.

Version information

Smart contract version

Our smart contract was written in Solidity version 0.5.16.

Dependencies version

Our smart contract inherits from the ERC 721 contract version 0.5.16 from OpenZeppelin. It is a standard for non-fungible tokens (NFTs) specifying the rules and interfaces for the creation and management of NFTs on the Ethereum blockchain (Open Zeppelin 2024).

Executive summary

Process overview and tools used

Our process for developing the project could be divided into two stages. The first stage is understanding the requirements and designing the front-end for the website. We first designed the UI for the website in Figma, then translated the design into a functional static website with ***React*** and ***TailwindCSS***.

The second stage is designing the backend and private blockchain system. For this, we had to design a ***MySQL*** database to store user and asset metadata, an API with ***Express*** to facillitate communication between the front-end and back-end, and a smart contract written in ***Solidity*** to handle the creation and trading of NFTs. We decided that our smart contract would inherit from the ERC 721 contract from OpenZeppelin. This contract implements the ERC 721 standard, which is a standard for working with NFTs. Our smart contract allows users to mint new NFTs, purchase NFTs, and view NFTs in their collection. We then used ***Truffle*** to compile and deploy our contract to the private blockchain network, which is run on a local ***Geth*** node.

Major findings

- The ERC721 standard ensures that each NFT has a unique identifier and the ownership of these tokens can be transferred securely and transparently between parties. This forms the establishments of ownership history for digital assets (Open Zeppelin 2024).
- ERC721 smart contracts eliminate the need for central agreements, such as marketplaces for the transfer and ownership verification (Open Zeppelin 2024).
- ERC721 tokens are compatible with various decentralized applications and platforms, enabling integration of the extended Ethereum ecosystem (Open Zeppelin 2024).

Smart contract requirements

To fulfil the requirements of the project, the smart contract needs to support the following functionalities:

- Allows users to mint NFTs. This helps grow the NFT collection on the platform and give users more items to trade. The minting function must ensure that the requested NFT has not been minted before. It should also charge the minter a small amount of money to prevent people from spamming mint requests and overloading the system.
- Allows users to buy NFTs from other users. This operation is central to the platform as it is a decentralized trading website. The function should transfer the ownership of the NFT to the buyer and send coins to the seller.
- Allows users to view the NFTs they own. Users need a way to see the assets in their collection to check that their trades have taken place successfully. The function should return a list of token IDs in the user's collection given their account address.

ABI illustration

Code structure

The source code for our smart contract is structured as follows:

- The first two lines specify the SPDX license and Solidity version. In our case, the license is MIT and the version is 0.5.16.
- Below the above declarations, we defined our smart contract, which inherits from ERC 721. We established this inheritance with the keyword `is`.
- Inside the smart contract body, we defined two state variables, a uint256 and a string-to-boolean mapping, to auto-generate token IDs and check the uniqueness of each token. We also defined a parameterless and bodyless constructor which calls the constructor of the base ERC 721 contract. This is followed by 3 custom functions, the details of which are discussed below.

Since our smart contract is quite small in size, it is relatively easy to read. We have also provided detailed comments in the source code to explain the functionality of each function.

ABI discussion

The compiled smart contract is stored on the blockchain as byte code. To interact with the functions of the smart contract, the outside caller must know what their names and expected inputs and outputs are. The ABI (Application Binary Interface) helps achieve this goal. It helps translate names and arguments specified by high-level languages (such as JavaScript using Web3.js) into byte code that the compiled smart contract can work with. It also helps convert responses from smart contract functions into the values expected in higher-level languages (Sen 2023).

The ABI for the smart contract of our project was generated with Truffle, an asset pipeline for Ethereum. After writing our Solidity source files, we ran `truffle compile` to generate a JSON file containing information about our contract, including its ABI. The ABI is an array of JSON objects each corresponding to a public function or event defined in the smart contract or its parent contracts. Our smart contract inherits from the ERC 721 contract from OpenZeppelin, which defines a standard for working with NFTs. Since

ERC 721 already comes with most of the features needed by our platform, we only had to write 3 custom functions.

As the ERC 721 contract from which we inherit contains many public functions, the ABI generated for our contract is quite large. The following section will ignore the functions from ERC 721 and only focus on our custom functions. The details of ERC 721 functions can be found in its official documentation.

safeMint function

ABI:

```
{
  "constant": false,
  "inputs": [
    {
      "internalType": "string",
      "name": "drippleAsset",
      "type": "string"
    }
  ],
  "name": "safeMint",
  "outputs": [
    {
      "internalType": "uint256",
      "name": "",
      "type": "uint256"
    }
  ],
  "payable": true,
  "stateMutability": "payable",
  "type": "function"
}
```

Function type: payable (the function receives Ether and can read/update the blockchain state.)

Inputs:

- `drippleAsset` - A string containing unique information about the NFT asset to mint. This string **MUST** be unique for each asset.

Outputs:

- A `uint256` representing the token ID of the newly minted token.

Functionality: Safely mints an NFT based on the supplied string and assigns the caller as its owner. The operation costs 0.5 coins and cannot be performed without this fee.

purchaseNFT function

ABI:

```
{
  "constant": false,
  "inputs": [
    {
      "internalType": "address payable",
      "name": "from",
      "type": "address"
    },
    {
      "internalType": "uint256",
      "name": "tokenId",
      "type": "uint256"
    }
  ],
  "name": "purchaseNFT",
  "outputs": [],
  "payable": true,
  "stateMutability": "payable",
  "type": "function"
}
```

Function type: payable (the function receives Ether and can read/update the blockchain state.)

Inputs:

- `from` - The address that you want to buy the NFT from. This address MUST NOT be the caller and MUST own the NFT.
- `tokenId` - A `uint256` representing the ID of the token you want to buy. This token MUST belong to the from address.

Outputs: None

Functionality: Transfers the ownership of the NFT with the `tokenId` to the function caller, then transfers the Ether amount passed into the function by the caller to the original owner.

getOwnedAssets function

ABI:

```
{
  "constant": true,
  "inputs": [
    {
      "internalType": "address",
      "name": "user",
      "type": "address"
    }
  ],
  "name": "getOwnedAssets",
  "outputs": [
    {
      "internalType": "uint256[]",
      "name": "",
      "type": "uint256[]"
    }
  ],
  "payable": false,
  "stateMutability": "view",
  "type": "function"
}
```

Function type: view (it can read but cannot modify the blockchain's state.)

Inputs:

- `user` - The address of the user whose NFT collection you want to read.

Outputs:

- An array of `uint256` values each representing a token in the user's collection.

Functionality: Returns an array of token IDs owned by the given user address.

Interacting with functions

After obtaining the ABI for the smart contract, we can call the smart contract functions like this from ***Web3.js:***

```
const web3 = new Web3(BLOCKCHAIN_ADDRESS);
const contract = new web3.eth.Contract(ABI, CONTRACT_ADDRESS);

const itemToMint = "My NFT";
contract.methods.safeMint(itemToMint).send({
    from: accountAddress,
    value: web3.utils.toWei(0.5, "ether"),
    gas: web3.utils.toHex(200000),
    gasPrice: "1000000"
});
```

The code above first creates a Web3 instance to connect to the blockchain at `BLOCKCHAIN_ADRESS`. It then creates a Contract object given the ABI and the address of the deployed contract. The smart contract functions are available through `contract.methods`. They are called with the same name and parameters as specified in the ABI (in this case the `safeMint` function is called with the string parameter `itemToMint`). To execute this function on the blockchain, the `send` function is called with additional information such as the address of the caller as well as the value of the transaction. If the function is marked as `payable` in the smart contract, the Ether can be sent through the `value` field.

Call graph discussion

Our Dripple smart contract extends the ERC721 contract, which is part of the OpenZeppelin library. This library provides implementations of standards like ERC20 and ERC721 which are widely used in the Ethereum ecosystem. The functions `_mint`, `safeTransferFrom`, `balanceOf`, and `tokenOfOwnerByIndex` are all part of this library (Open Zeppelin 2024).

All the logic has been implemented in the call chart diagram below. More specifically, the `safeMint` function calls the `_mint` function to create a new token. The `purchaseNFT` function calls `safeTransferFrom` to transfer a token from one address to another. The `getOwnedAssets` function calls `balanceOf` to get the number of tokens owned by an address, and `tokenOfOwnerByIndex` to get the specific tokens owned by that address.

The interrelationships between these functions form the core features of the contract. The `safeMint` function allows users to create new tokens, the `purchaseNFT` function allows users to buy existing tokens, and the `getOwnedAssets` function allows users to see which tokens they own. These functions work together to allow users to create, buy, and keep track of tokens, which are the fundamental operations of our system.

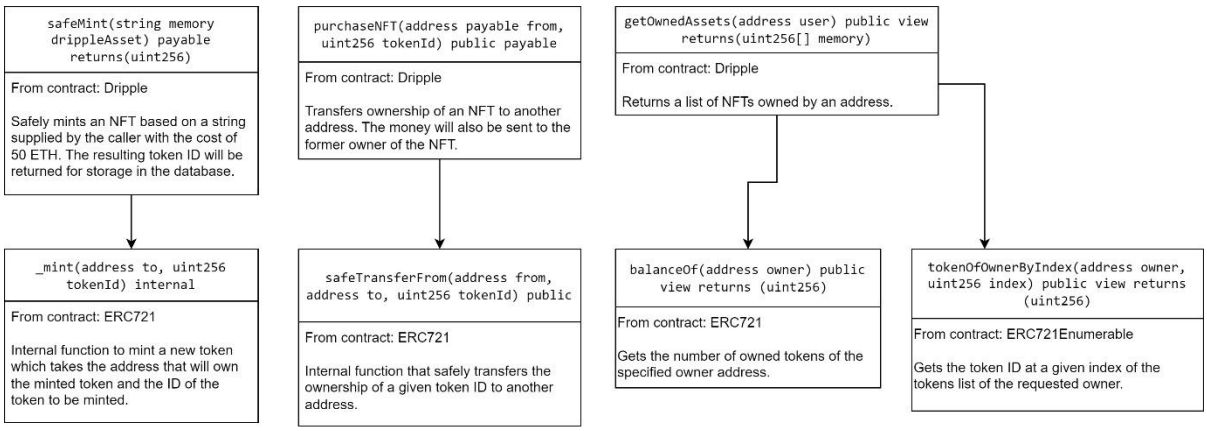

Fig. 1: The call graph for our smart contract functions.

Inheritance graph discussion

The following is the inheritance graph showing the relationship between our smart contract and the contracts of the ERC 721 standard:

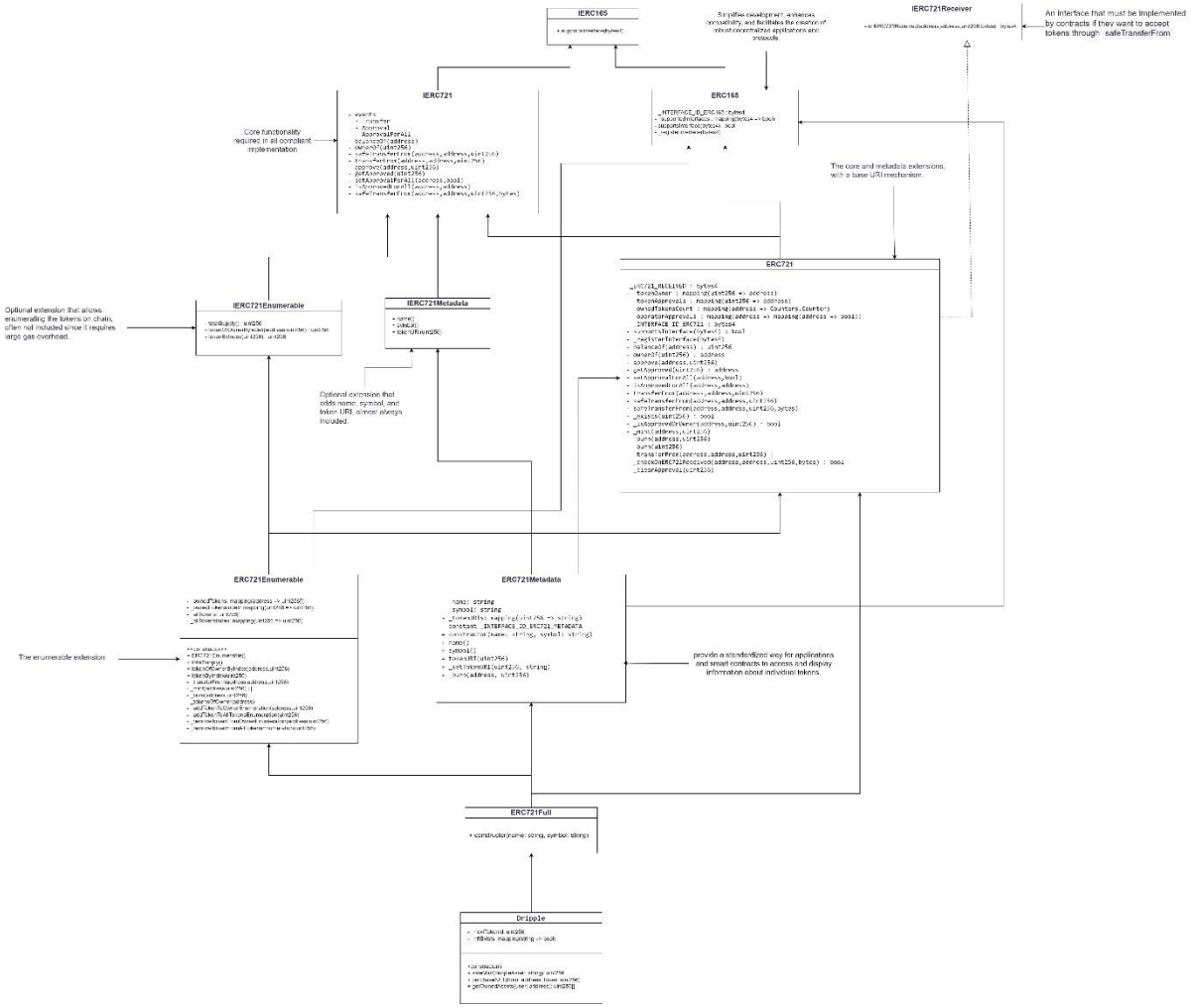

Fig. 2: The inheritance graph.

Inheritance relationships

Solidity allows one contract to inherit from one or more parent contracts to extend their functionality. This allows the child contract to reuse the logic defined in its parents without having to write extra code. This is useful when the contract requires industry-standard functionalities as it can simply inherit from contracts that are written and maintained by professionals. In our case, the smart contract extends the ERC 721 contract from OpenZeppelin. The inheritance graph shows that our smart contract (at the bottom) inherits from ERC721Full, which in turn inherits from ERC721Enumerable, ERC721Metadata, and the core ERC721 contract. This gives our contract access to the

full suite of functions needed to work with NFTs such as minting, transferring, burning, and enumerating. The parent contracts of Dripple themselves inherit from other contracts and interfaces, such as IERC721, IERC721Enumerable, and ERC165.

Code reusability

As mentioned in the previous section, our custom smart contract can reuse the logic of parent contracts without writing additional code, which allows us to quickly extend the logic of parent contracts. For example, we have extended the base _mint function (provided by parent contract ERC 721) into the safeMint function in our smart contract to charge a minting cost of 0.5 ETH and ensuring that the requested asset is unique. Without reusing _mint, we would have to handle the minting logic ourselves, which would have taken time and potentially caused security issues as we are not experts in blockchain.

Function and state variable overriding

Our Dripple contract does not override anything from its parents. However, the inheritance graph shows multiple cases of overriding among Dripple's parent contracts. An example of this is the contract ERC721Enumerable, which inherits from the interface contract IERC721Enumerable. IERC721Enumerable specifies functions with only signatures and no bodies. The contracts that inherit from this interface (i.e. ERC721Enumerable) need to override these functions and implement the bodies. Different children of IERC721Enumerable can provide different implementations of the functions to adapt to different use cases.

Function visibility and access control

Several of Dripple's parent contracts have functions that can only be called internally. An example of this is the _mint function of ERC721. It is marked as internal in the source code, and as such it can only be called from within that contract or its children. Dripple makes use of this function in safeMint, which is a public function that extends the functionality of _mint to charge a minting fee and ensure uniqueness.

Our Dripple contract has not altered the visibility of its parent contracts' functions. All of their public functions can be invoked from the outside. However, for our project, it is safe to leave these functions public because they do not critically impact our features.

Modifier application

Modifiers are pieces of code that can run before or after a function is executed, usually for the purpose of input validation and access control (The Solidity Authors 2023). The Dripple contract and its parents have not made use of any modifiers. However, some of Dripple's functions have checks at the start of the function body that could be extracted into modifiers. An example of this is the check at the beginning of the purchaseNFT function. This check ensures that the caller of the function (the buyer) is different from the seller because it would not make sense for a buyer to buy NFTs from themselves. The modifier could be written as follows:

```
modifier buyerNotSeller(address _buyer, address _seller) {
    require (_buyer != _seller, "You cannot purchase an NFT from yourself!");
    _;
}
```

References

Open Zeppelin 2024, *ERC721*, Open Zeppelin, viewed 06 April 2024, <https://docs.openzeppelin.com/contracts/4.x/erc721>

Open Zeppelin 2024, *ERC721*, Open Zeppelin, viewed 06 April 2024, <https://docs.openzeppelin.com/contracts/4.x/api/token/erc721>

Sen, S 2023, *What is an ABI?*, QuickNode, viewed 06 April 2024, <https://www.quicknode.com/guides/ethereum-development/smart-contracts/what-is-an-abi>

The Solidity Authors 2023, *Contracts*, Solidity, viewed 06 April 2024, <https://docs.soliditylang.org/en/latest/contracts.html#function-modifiers>