Appendix A. Categories of smart contract comment

This subsection presents our defined comment classification and corresponding simplified code examples based on OpenZeppelin smart contracts.

Appendix A.1. Functional

It focuses on describing the primary purpose and functionality of the code, emphasizing "what it does."

Appendix A.1.1. Business Logic

This type is used to explain the core business functionality of a function, i.e., the primary operational purpose of the code. For example, in a token contract, the business logic may involve token transfers, minting, or burning; in an NFT contract, it may cover NFT minting, transfers, or auction processes. Listing 4 shows an example of code and comment of this type.

```
// Moves `amount' tokens from the caller's account to `to'.
function transfer(address to, uint256 amount) public returns (bool) {
    _transfer(msg.sender, to, amount);
    return true;
}
```

Listing 4: An example of business logic code and comment from OpenZeppelin.

Appendix A.1.2. State Change

This type is used to label functions that modify the storage of a smart contract, i.e., operations that alter the contract's state (such as variables, data structures, or on-chain resources). For example, setter functions are commonly used to set a state variable in the contract, or state updates may occur through other logic, such as modifying account balances, updating contract permissions, or changing configuration parameters. Listing 5 provides an example of this type of code and comment.

```
// Sets `value' as the allowance of `spender' over the owner's tokens.
function _approve(address owner, address spender, uint256 value) internal{
    _approve(owner,spender,value, true);
}
```

Listing 5: An example of state change code and comment from OpenZeppelin.

Appendix A.1.3. Read Only

This type is typically used to read the state of a smart contract or perform computations, without having any persistent impact on on-chain data. Listing 6 shows an example of code and comment of this type.

```
// Returns the amount of tokens owned by `account'.
function balanceOf(address account) public view virtual returns (uint256) {
    return _balances[account];
}
```

Listing 6: An example of read only code and comment from OpenZeppelin.

Appendix A.1.4. Internal Helper

This type is used to indicate that the function is an internal tool function, designed to serve the internal logic of the smart contract and typically not exposed to external callers. These functions generally perform auxiliary operations, such as computations, validations, or data processing, with the goal of supporting the core business logic of the contract. Listing 7 shows an example of code and comment of this type.

```
// Destroys `amount' tokens from `account', reducing the total supply.
    Internal function without access restriction.
function _burn(address account, uint256 amount) internal virtual {
    if (account == address(0)){
        ....
    }
    _update(account, address(0), value);
}
```

Listing 7: An example of internal helper code and comment from OpenZeppelin.

Appendix A.2. Interface and Usability

It is used to specify the invocation method of a function, its access restrictions, and details of its inputs and outputs. The primary purpose is to help developers use the function correctly and securely. This category emphasizes descriptions of usage, focusing on "how to use it."

Appendix A.2.1. Access Control

It specifies the caller restrictions of a function, such as allowing only the contract owner (onlyOwner) or only administrators to execute it. Such comments are primarily used to indicate the security boundaries and access permissions of the function. Listing 8 shows an example of code and comment of this type.

```
// Leaves the contract without owner. It will not be possible to call `
    onlyOwner' functions anymore. Can only be called by the current owner.
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Listing 8: An example of access control code and comment from OpenZeppelin.

Appendix A.2.2. Invocation Attributes

It explains the function's invocation method or modifiers, such as payable (allows receiving Ether), or external (callable only externally). Listing 9 shows an example of code and comment of this type.

```
// Callable by any address and can receive Ether.
function cancelOwnershipHandover() public payable virtual {
    assembly {
        mstore(0x0c, _HANDOVER_SLOT_SEED)
        mstore(0x00, caller())
        sstore(keccak256(0x0c, 0x20), 0)
        ...
    }
}
```

Listing 9: An example of invocation attributes code and comment from OpenZeppelin.

Appendix A.2.3. Inputs/Outputs

This type explains a function's input parameters and return values, including their meaning, data format, and units. Listing 10 shows an example of code and comment of this type.

```
// Returns true if `account' is a minter.
function isMinter(address account) public view returns (bool) {
    return hasRole(MINTER_ROLE, account);
}
```

Listing 10: An example of inputs/outputs code and comment from OpenZeppelin.

Appendix A.3. Safety and Constraint

Its purpose is to alert developers to potential risks and safeguards, thereby reducing security vulnerabilities. This category focuses on "what to watch out for."

Appendix A.3.1. Constraints

It specifies the constraints on function invocation, such as value ranges (e.g., $amount > 0$), gas consumption requirements, or invocation order dependencies. Their primary purpose is to highlight the usage boundaries for developers, thereby preventing misuse that may cause execution failures or potential security vulnerabilities. Listing 11 shows an example of code and comment of this type.

```
// Transfers `amount' tokens to `recipient', the caller must have a balance
    of at least `amount'.
function transfer(address recipient, uint256 amount) public virtual override
     returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Listing 11: An example of constraints code and comment from OpenZeppelin.

Appendix A.3.2. Exceptions

It specifies the conditions under which a function execution may fail and the corresponding error messages, typically triggered by require or revert statements. They help developers understand when exceptions will be raised, thereby reducing invalid calls and logical errors during usage. Listing 12 shows an example of code and comment of this type.

```
// Throws if called by any account other than the owner.
modifier onlyOwner() {
    require(owner() == _msgSender(), "Ownable: caller is not the owner");
    _;
}
```

Listing 12: An example of exceptions code and comment from OpenZeppelin.

Appendix A.3.3. Deprecation

It indicates that a function or interface is deprecated and should no longer be used, as it may be removed in future versions. Their purpose is to alert developers to migrate to the recommended alternatives, ensuring long-term maintainability and security of the contract. Listing 13 shows an example of code and comment of this type.

```
// This function is deprecated and use Solidity 0.8.x built-in overflow
    checks instead.
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "SafeMath: addition overflow");
    return c;
}
```

Listing 13: An example of deprecation code and comment from OpenZeppelin.

Appendix A.3.4. Security Patterns

It describes the security design patterns implemented in a smart contract, such as Reentrancy Guard or Checks-Effects-Interactions. They help developers understand how the contract maintains robustness against potential attacks, ensuring the safe execution of critical operations. Listing 14 shows an example of code and comment of this type.

```
// Applying the nonReentrant modifier to functions ensures that there are no
    nested (reentrant) calls.
modifier nonReentrant() {
    require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
    _status = _ENTERED;
    _;
    _status = _NOT_ENTERED;
}
```

Listing 14: An example of security patterns code and comment from OpenZeppelin.

Appendix A.4. Execution and Interaction

It describes how a smart contract behaves within the blockchain execution environment and how it interacts with other contracts or external components. The main purpose is to clarify the contract's system-level behavior and collaboration, ensuring its correct operation in complex settings. This category focuses on "system-level behavior."

Appendix A.4.1. Fallback Fuction

The fallback function is triggered when the call data does not match any of the functions defined in the contract. It is commonly employed to handle unexpected calls, such as accepting unintended transactions, recording log information, or serving as a safeguard against potential malicious attacks. Listing 15 shows an example of code and comment of this type.

```
// This implementation logs the call details.
fallback() external payable {
    emit FallbackCalled(msg.sender, msg.value, msg.data);
}
```

Listing 15: An example of fallback fuction code and comment from OpenZeppelin.

Appendix A.4.2. Receive Function

The receive function is specifically designed to accept Ether transfers. It is automatically triggered when a transaction includes Ether and *msg.data* is empty. This function is typically employed to implement deposit mechanisms, reward distribution, or general fund reception within the contract. Listing 16 shows an example of code and comment of this type.

```
// Function to receive ETH that will be handled by the governor (disabled if
    executor is a third party contract)
receive() external payable virtual {
    if (_executor() != address(this)) {
        revert GovernorDisabledDeposit();
    }
}
```

Listing 16: An example of receive fuction code and comment from OpenZeppelin.

Appendix A.4.3. Cross-Contract Interaction

It describes the interactions between a function and external contracts or system components, such as invoking external contract functions, executing logic via delegatecall, or handling callbacks from oracles. The primary purpose is to clarify the contract's external dependencies and highlight potential risks (e.g., reentrancy, external call failures), thereby enhancing transparency and maintainability in contract design. Listing 17 shows an example of code and comment of this type.

```
// Function to receive ETH that will be handled by the governor (disabled if
    executor is a third party contract)
function _checkOnERC721Received(address from, address to, uint256 tokenId,
    bytes memory data) internal returns (bool) {
    if (to.isContract()) {
        try IERC721Receiver(to).onERC721Received(msg.sender, from, tokenId,
            data) returns (bytes4 retval) {
            return retval == IERC721Receiver.onERC721Received.selector;
        } catch (bytes memory reason) {
            if (reason.length == 0) {
                revert("ERC721: transfer to non ERC721Receiver implementer")
                    ;
            } else {
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    } else {
        return true;
    }
}
```

Listing 17: An example of cross-contract interaction code and comment from OpenZeppelin.

Appendix A.5. Maintenance

It is not directly related to the smart contract function's business logic but facilitate developers' understanding of the design rationale, regulatory context, or future evolution of the code. They also support debugging, upgrading, and long-term maintenance.

Appendix A.5.1. Metadata

It primarily provides background information about the contract, such as the protocol standards it follows, version compatibility requirements, and the overall design intent. Unlike comments that explain the internal logic of a specific function, these focus on the broader context of the contract, helping developers understand the design rationale and long-term maintenance goals. Listing 18 shows an example of code and comment of this type.

```
// An upgradeable ERC20 token contract designed for governance systems.
function initialize(string memory _name, string memory _symbol, uint8
    _decimals, uint256 _supply) public initializer {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        totalSupply = _supply;
    }
```

Listing 18: An example of metadata code and comment from OpenZeppelin.