Using Non-Transferable Tokens as Objects on a Decentralised Smart Contract Platform in a Game Theory Simulation

Ozzak Jure Matić
118743625

CS4501 - Final Year Project
BSc Computer Science
University College Cork

Supervisor: Dr. James Doherty
2nd Reader: Dr. John Herbert
April 2023

Table Of Content

Abstract

The objective of this investigative research project is to showcase the feasibility of using Non-Transferable Tokens (NTTs) as Objects within an object-oriented programming framework, specifically for interacting with Smart Contracts on a decentralised platform like Ethereum. This innovative approach holds potential for creating decentralised organisations where tokens could serve as authentication, identification tools, or any abstract dObject. To validate this concept, a series of simple game-theoretical scenarios will be simulated, in which agents utilising tokens as objects to interact with smart contracts will demonstrate the viability of this approach.

Extended Abstract

Defining Relevant Concepts and Technologies

Game theory

Game theory finds relevance in numerous domains, encompassing economics, political science, psychology, biology, computer science, and engineering. As a mathematical and economic discipline, game theory investigates the conduct of agents, known as "players," in scenarios where one agent's decision outcome relies on other agents' choices. In these circumstances, agents possess varying objectives and may have partial or flawed knowledge about one another. Game theory aims to establish models that encapsulate the strategic interplay among agents and scrutinise the potential consequences of such interactions.

Ethereum Smart Contract

Smart Contract is an unfortunately named technology that isn't particularly smart nor is it a contract. Simple and more accurate definition of a “Smart Contract” technology was given by [1.] Antonopoulos (2018) *“the term smart contract refers to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol - ie. on the decentralised Ethereum world computer.”* (p.127). Other characteristics to keep in mind when talking about Ethereum Smart Contracts are characteristics shared with all blockchain protocols, like transparency (smart contracts on Ethereum are transparent, meaning that anyone can view the code and the transactions on the public blockchain), trustlessness (smart contracts on Ethereum are trustless, meaning that parties do not need to trust each other as the contract is enforced automatically by the Ethereum network) and self-execution (smart contracts on Ethereum are self-executing, meaning they automatically execute the terms of the contract without the need for intermediaries).

Non-Fungible Tokens (NFTs)

Non-Fungible Tokens (NFTs) are distinct digital assets representing ownership or authentication of specific items, such as artwork or collectibles. These tokens reside on a blockchain, ensuring transparency, security, and immutability. Programmatically, NFTs consist of data and/or code blocks stored on the blockchain. The quantity of data storable depends on the "block gas limit," a parameter dictating the maximum computational capacity for a single Ethereum Virtual Machine (EVM) block. Each NFT contract operation necessitates a certain amount of gas payment to the network for computation. Consequently, the data quantity stored in an NFT relies on the contract

code's efficiency and the required gas for each operation. Generally, it is advisable to store only crucial data in an NFT and utilise external storage solutions for larger data volumes.

Non-Transferable Tokens (NTTs)

Non-Transferable Tokens (NTTs) are a type of Non-Fungible Tokens (NFTs) that cannot be transferred from one account to another, hence the name “non-transferable”. They can be used to represent ownership or proof of participation in a particular event. There are also attempts to use them as proof of identity. All of the characteristics of NFTs are valid for NTTs as well but since they cannot be transferred they are less flexible than other types of tokens. Nonetheless, this limitation presents distinctive opportunities that this research seeks to leverage.

Object Oriented Programming (OOP)

Object Oriented Programming (OOP) is a common modern programming paradigm, and in this context, software design methodologies that emphasise the representation of the world as a set of interacting objects rather than as a sequence of actions to be taken. As stated by [2.] Lippman, S. B., & Lajoie, J. (2003), *"Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties [attributes] and behaviours [methods] are bundled into individual objects."* (p. 51).

dObject and dClass

dObject is a novel term introduced in this paper, referring to a Non-Transferable Token that serves as an object in Object-Oriented Programming (OOP). The study aims to demonstrate the applicability of key OOP concepts, such as abstraction, inheritance, encapsulation, and polymorphism, in a decentralised manner using NTTs. The paper will investigate the feasibility of implementing these concepts and explore potential solutions and best practices for using this framework. Each dObject will be created by the dClass describing the properties of the dObject.

Blockchain Security Model

In order to analyse the use cases for dObjects, it is necessary to understand the security model and advantages of blockchain networks. Main security features of blockchain are:

- Immutability: Once data is recorded on the blockchain, it cannot be altered or deleted, making it tamper-proof.

- Decentralisation: The blockchain operates as a decentralised system, eliminating the need for a central authority to govern the network.
- Transparency: All transactions on the blockchain are visible to all participants in the network, ensuring transparency and accountability.
- Security through cryptography: The blockchain uses advanced cryptographic techniques to store data and transactions on the network. Guaranteeing that data on the blockchain is secure and cannot be hacked or tampered with.
- Fault tolerance: The blockchain technology is designed to be fault-tolerant, which means that the system can continue to function even if some nodes on the network fail or go offline. This feature makes the blockchain technology a highly resilient and secure system.

Blockchain Advantages

The advantages of blockchain systems are best described by Andreas Antonopoulos as “The 5 Pillars of Open Blockchains” in his 2019 talk at Google Campus in Seoul, South Korea. The pillars in question being:

1. Open: The system is transparent, allowing anyone to participate in the network and access its data.
2. Public (Permissionless): Users do not need permission or authorisation to join the network, which enables anyone to engage with the system.
3. Neutral: The network operates impartially, treating all users equitably without bias or discrimination.
4. Borderless: The network is not limited by geographical or political borders, and it is accessible to anyone with an internet connection.
5. Censorship Resistant: The network is designed to resist censorship, ensuring that transactions cannot be censored or reversed.

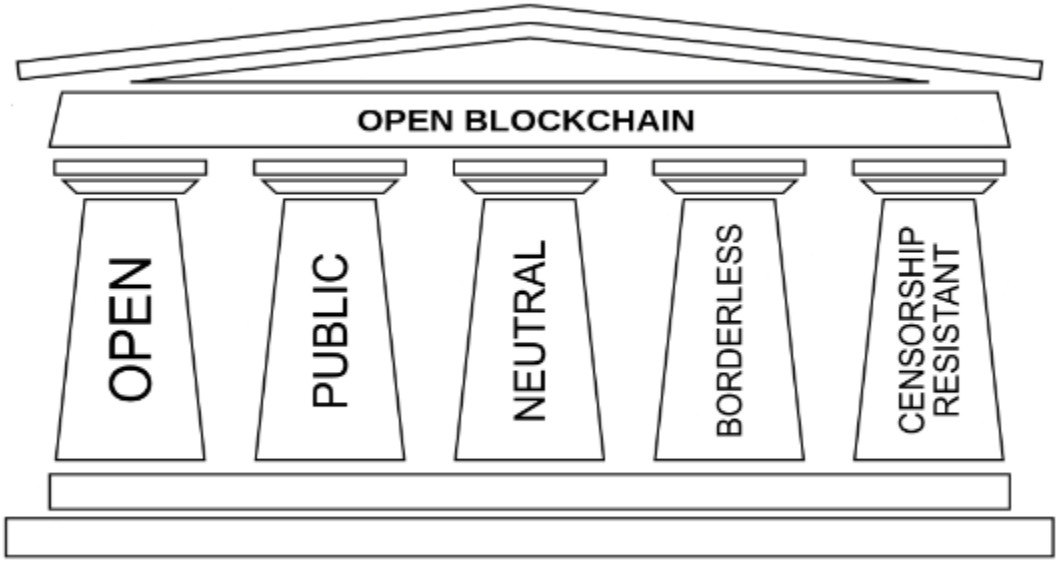


Figure 1. - Five pillars of Open Blockchain

Possible Use Case for dObjects

In conventional object-oriented programming, objects offer an organised and comprehensible method for arranging and structuring code, thereby enhancing its readability, maintainability, and extensibility. Objects can represent intricate data structures like lists, trees, and graphs, while also encapsulating behaviour and functionality to enable modular and adaptable code reuse. Furthermore, objects can model real-world entities, such as users, products, or orders, simplifying system behaviour comprehension and facilitating the development of more resilient and scalable software applications.

Aside from the typical advantages and applications of Object-Oriented Programming (OOP), dObjects are expected to capitalise on the Blockchain security model and Blockchain advantages listed in the previous section. Consequently, the essential issue to explore is pinpointing practical use cases that would benefit from having portions of their code or data stored and protected as dObjects in a decentralised fashion on a public blockchain.

Within the context of decentralised smart contract platforms like Ethereum, dObjects will be represented by Non-Transferable Tokens (NTTs), which can serve as a mechanism for authentication, identification, or integrity validation. The rationale behind this approach stems from the organised and well-structured representation and handling of data and functionality that objects provide, while NTTs offer a method for confirming ownership and control of digital assets securely and in a decentralised way. Potential use cases span across various domains, such as finance, gaming, supply chain management, and digital identity, to name a few.

In a recent publication, "Decentralised Society: Finding Web3's Soul" (May 10, 2022), authored by E. Glen Weyl, Puja Ohlhaver, and Vitalik Buterin, a number of use cases for a technology analogous to dObjects have been proposed. The authors utilised Non-Transferable Tokens (NTTs) which they referred to as "soulbound tokens" (SBTs) in their implementation. For instance, they discussed the following use cases:

> *"Permissioning access to privately or publicly controlled resources (e.g., homes, cars, museums, parks, and virtual equivalents). Transferable NFTs fail to capture this use case well because often access rights are conditional and non-transferable: if I trust you to enter my backyard and use it as recreational space, that does not imply that I trust you to sub-license that permission to someone else." (p.10)*
>
> *"Data Cooperatives where SBTs grant data access to researchers, while instantiating members' rights to grant access (perhaps by quadratic vote) and bargain for economic rights to discoveries and intellectual property born out of research." (p.10)*
>
> *"Experiments with local currencies with rules that make them more valuable to hold and spent by Souls who live in a particular region or are part of a particular community." (p.11)*

> *"Experiments in market design, such as Harberger taxation and SALSA (self-assessed licenses sold at auction), where holders of an asset post a self-assessed price at which anyone else can buy the asset from them, and must periodically pay a tax proportional to the self-assessed price to maintain control. SBTs could be used to create more nuanced versions of SALSA—for example, where rights of participation are approved by the community to minimize strategic behavior from within or outside the community." (p.11)*
>
> *"Experiments in democratic mechanism design such as quadratic voting. Holders of SBTs representing membership in a community could quadratically vote on parameters such as incentives and tax rates. Ultimately, "markets" and "politics" are not separate design spaces; SBTs can be a major part of a technological stack that enables the entire space between the two categories to be explored. Provision of public goods through quadratic funding is another such intersection." (p.11)*

It is worth noting that all of the concepts presented can be implemented through the use of dObjects. Additionally, considering that dObjects are as flexible and versatile as objects in OOP are, use of dObjects can allow for further expansion of these concepts. In summary, the utilisation of dObjects to interact with smart contracts can lead to the creation of decentralised organisations that offer greater security, transparency, and reliability compared to conventional centralised systems.

Using Game Theory as proof of concept

Game Theory can be defined as the study of strategically interdependent behaviour, which becomes relevant in the context of blockchain protocols as the actions of node A can impact the outcomes of node B and actions of node B impact the outcomes of node A. This is why Game Theory principles are commonly utilised in blockchain protocols to secure the network, validate blocks, reward miners, punish misbehaving nodes, mint new coins, etc.

From Bitcoin to more modern and complex protocols, on the lowest level of abstraction blockchains can be viewed as games where nodes follow the set of game rules, to create the decentralised system that is cryptocurrency. In the case of the Bitcoin protocol the game is a simple "enter the new transactions into a ledger" game. Every full node has its own copy of the ledger, and every node listens on the network for new blocks that adhere to game rules. Specialised nodes called "miners" create new blocks and validate the incoming transactions, rejecting the ones that do not conform to the rules, for this service mining nodes are rewarded with newly minted coins approximately every 10 minutes. To ensure the proper behaviour and adherence to the protocol rules, mining nodes themselves are required to validate their entries to the blockchain by submitting "proof of work" in the form of SHA256 hashes. Thus proving they spent some amount of energy (money on electricity) to create their new block

proposal. In case two different mining nodes submit two conflicting blocks the “full node client” will wait until one of the chains in the “soft fork” grows significantly longer and then discard the shorter chain, as described by the “longer chain rule”.

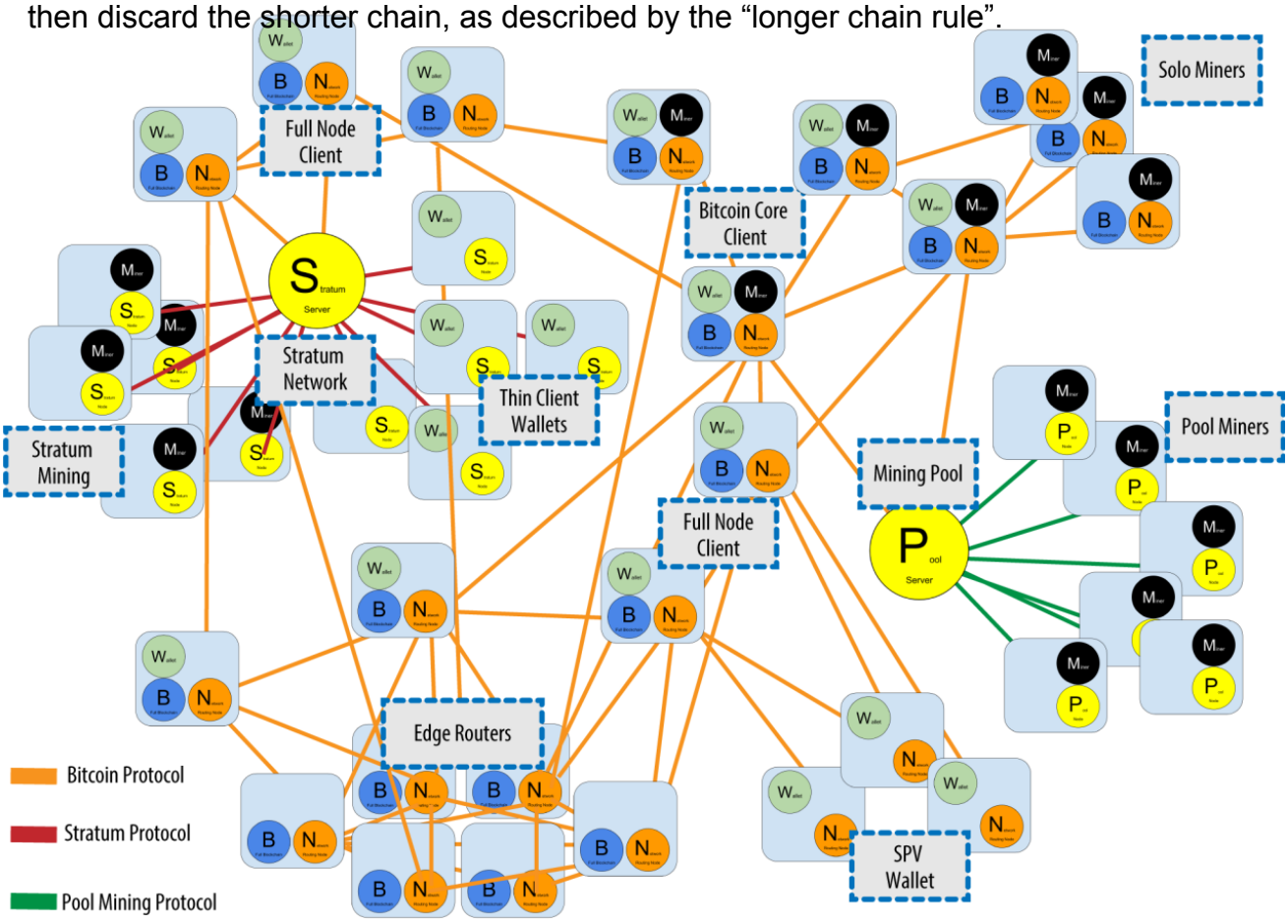


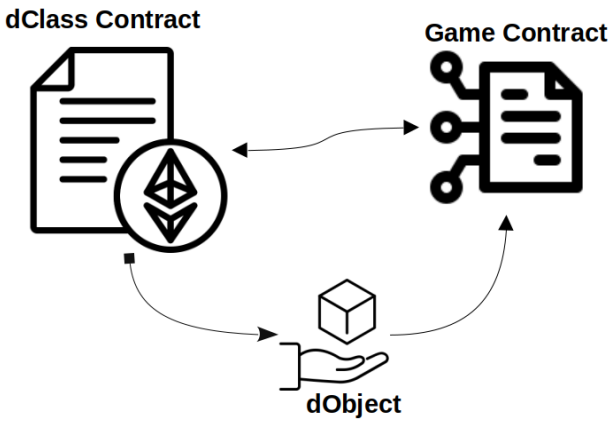
Figure 2. - Diagram of the Bitcoin network protocols with different node types and gateways

The depiction of the Bitcoin protocol provided is a simplified and incomplete representation, as demonstrated by the accompanying [Figure 2]. Nonetheless, it is notable that the consensus rules governing the Bitcoin network can be expressed using the language of Game Theory mathematics. In short, to build truly decentralised systems, the use of Game Theory is essential. Proper coordination and cooperation of network participants can only be ensured if all incentives align appropriately. Therefore, Game Theory serves as a perfect tool for describing the "Rules Without Rulers" that govern blockchain systems.

Hence, the purpose of this paper is to demonstrate, through simple Game Theory models, a proof of concept for dObjects. Specifically, classic static and dynamic games of complete information will be simulated. It is important to note that this experiment is not intended to create a fully functional decentralised protocol, but rather to illustrate that the logical building blocks of Game Theory can be used in designing such systems.

Simulation Structure

The simulations will be deployed as Smart Contracts on a local instance of Ethereum blockchain. Each set of Smart Contracts will include a dClass Contract that will be used for creating dObject Tokens, as well as one Game Contract. The dClass Contract will serve as a constructor for the dObject Tokens and will aim to validate the implementation of fundamental OOP principles such as inheritance, abstraction, and polymorphism. The Game Contracts will be utilised to integrate Game Theory logic and validate the central premise of this paper. [Figure 3.] depicts creation of the dObject, agent interaction with the Game Contract using dObject and the communication between the Game Contract and the dClass Contract to validate the dObject and interact with the data stored within it.



• Figure 3. - Simulation Deployment

Non-Transferable Tokens as dObjects

In this section, the concept of Non-Transferable Tokens (NTTs) as dObjects is explored, focusing on the Solidity programming language framework and various contract structures that can be employed for achieving the desired functionality. The evolution of Ethereum Request for Comments (ERC) standards and their relevance to NTTs are discussed. Furthermore, the intricacies of the Solidity programming language are examined, comparing and contrasting different approaches for creating abstract objects within this framework, while addressing security concerns and potential solutions for the challenges associated with NTTs.

Solidity Programming Language Framework

Solidity is a high-level programming language for developing smart contracts. As described by [1.] Antonopoulos (2018) ***"****Solidity was created [...] as a language explicitly for writing smart contracts with features to directly support execution in the decentralised environment of the Ethereum world computer. The resulting attributes are quite general, and so it has ended up being used for coding smart contracts on several other blockchain platforms. [...] Solidity is now developed and maintained as an independent project on GitHub. The main product of the Solidity project is the Solidity compiler, solc, which converts programs written in the Solidity language to EVM [Ethereum Virtual Machine] bytecode. The project also manages the important application binary interface (ABI) standard for Ethereum smart contracts."* (p131)

Solidity syntax is inspired by JavaScript and C++ and it supports inheritance, libraries and it is statically typed. Contracts in solidity are similar to classes in OOP but are not perfectly analogous with conventional classes and objects. This will be discussed in more detail in subsequent sections, as it affects how the central thesis of the paper will be solved.

Non-Transferable Token Contract Structure

In the context of Ethereum, ERC (Ethereum Request for Comments) refers to a set of standards that are proposed to improve the functionality of the Ethereum blockchain. ERC standards define the rules for the creation and management of different types of tokens on the Ethereum blockchain, including fungible tokens (such as ERC-20), non-fungible tokens (such as ERC-721), and many others. ERC standards are developed through a process of community feedback and collaboration, with developers and users of the Ethereum blockchain providing input and suggestions for improvement. Once a standard is proposed, it is published as an Ethereum Improvement Proposal (EIP) and undergoes a review process before it is implemented.

There are a couple of ERC proposals currently being undertaken by the Ethereum developer community that would fit the requirements of this project. Such as [4.] Tim Daubenschütz's (2022) "ERC-4973: Account-bound Tokens" and [5.] (2022) "ERC-5192: Minimal Soulbound NFTs", or [6.] "ERC-4671: Non-Tradable Tokens Standard" by authors Omar Aflak, Pol-Malo Le Bris and Marvin Martin (2022) and [7.] "ERC-6454: Minimalistic Non-Transferable NFTs" by Authors Bruno Škvorc, Francesco Sullo, Steven Pineda, Stevan Bogosavljevic, Jan Turk (2023).

Unfortunately, as it can be noted, all of the proposals for the Non-Transferable Token standard have been initiated in the last year, and are currently in the development phase. This is a very new topic in the cryptocurrency space. Therefore, there is no existing standard for NTTs that could be used in this project. As a result, a solution had to be developed by using a workaround and overriding the ERC-721 standard, typically utilised for conventional Non-Fungible Tokens.

The function shown in [Figure 4.] works by enforcing the "override" of the standard token transfer function, and employing the 'require' error handling method to raise an exception when either the sending 'or' receiving address is not a *null* value. This condition is impossible to satisfy effectively rendering the token non-transferable in accordance with the intended design.

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 tokenId,
    uint256 batchSize)
    internal override(ERC721, ERC721Enumerable) {
    require(from == address(0) || to == address(0),
        "ERROR: You can't send NonTransferableTokens :P");
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

Figure 4. - Override function needed to mint Non-Transferable Tokens

dClass Contract Structure

Upon researching the intricacies of the Solidity programming language and the Ethereum Virtual Machine, it has been determined that there are two possible mechanisms for creating abstract Objects. The first approach involves utilising Parent-Child contracts, while the second approach entails crafting a user-defined data type 'struct' and tying abstract attributes and methods to it. Each of these approaches presents its own set of benefits and drawbacks, which will be explored in the following section.

Parent-Child Contract Objects

The Solidity programming language incorporates many object-oriented programming (OOP) terms and abstract concepts, with the primary goal to minimise the data stored on the Ethereum Network. To maximise code reusability for commonly

implemented contract structures and libraries, Solidity employs network-wide, multilevel hierarchical inheritance. Encapsulation and polymorphism can also be effectively utilised within this framework. Consequently, each Solidity contract can be considered a class in the OOP paradigm. However, Solidity diverges from traditional OOP in the way it handles class instance states. In classical OOP, the data comprising a class is stored for each instantiation, meaning that each class instance (e.g., object1 and object2) has its own allocated memory state. In contrast, each Solidity contract running in the program has a single state (singleton class). This means that each Solidity contract functions as a class with only one instance state, and each new instance requires the definition of another contract (class). In other words, all data fields within a contract can be compared to static members in a real OOP language like Java.

Hence, creating a new Object (new class instance with its own state) is in most cases prohibitively expensive on the Ethereum network, as the user needs to pay the network for storing the entire class definition, not just state variables. This design choice is intentional with the goal of reducing the amount of data stored on the Network. While this approach is currently employed in some cases, this paper will not delve further into it, as ample research has already been conducted on the topic.

Data Type 'struct' Objects

The second approach for creating Objects in Solidity involves utilising the user-defined data type 'struct', which allows developers to create complex data structures by combining multiple variables of different data types under a single name. It is important to note that 'struct' alone falls short at emulating a true object, as there is no straightforward way for embedding methods (functions) into it. Consequently, it is at best considered an object-like structure rather than a fully fledged one. On the other hand the 'struct' type can further be stored in the Solidity 'mapping' data structure, which allows developers to store key-value pairs similar to dictionaries in other programming languages. This addresses the previously discussed issue of high Ethereum network fees associated with Parent-Child Objects. It is also important to note that mappings are virtually initialised, meaning that every possible key already exists and is mapped to a default value (usually zero). And Solidity does not keep track of the keys inserted into mappings, so there is no way to iterate over the keys or determine their count. To maintain a record of keys in a mapping, developers must employ a separate data structure, such as an array or a linked list.

```
struct Object {
    // Attributes
    string stringVar;
    uint uintVar;
}
```

Figure 5. - Simple Struct

In order to further expand on the functionality of 'struct' data type to better resemble a proper Object, two techniques were identified and employed during this study. The first approach involves using boolean type as a function flag. This enables the execution of specific code based on the value of the flag, as demonstrated in the

following example: iff (obj.functionFlag == true) then {execute Foo()}. While this process is effective, its capabilities are somewhat limited.

```
struct Object {
    // Attributes
    string stringVar;
    uint uintVar;
    bool boolVar;
    // Methods
    bool functionFlag;
    bytes4 functionSelector;
}
```

Figure 6. - Struct Object with Methods

The preferred technique discovered was to utilise the Function Selector feature of the Ethereum Virtual Machine (EVM). In Solidity a Function Selector is a unique identifier that represents a specific function within a smart contract. Function selectors are generated from the function's signature, which consists of a function's name and its argument types. The selector is created by taking the first four bytes of the Keccak-256 hash of the function signature. Function selectors are an essential part of Ethereum's contract ABI (Application Binary Interface), which outlines the process of calling functions in a smart contract, as well as encoding and decoding data for contract interaction. When a user submits a transaction to a smart contract, the transaction data includes the function selector along with the encoded arguments for the function. The Ethereum Virtual Machine (EVM) uses the function selector to determine which function to execute within the smart contract.

```
// Computing Function Selector of function "foo"
bytes4 functionSelector = bytes4(keccak256("foo(uint,bool)"));

// Encode the arguments with the previously computed function selector
bytes memory data = abi.encodeWithSelector(functionSelector, 10, true);

// Call the function "foo" using the low-level `call` function
(bool success, bytes memory returnData) = targetContract.call(data);
```

Figure 7. - computing and calling Function Selector of function "foo"

This low-level operation enables the storage of any arbitrary function call inside of the 'struct' object, thereby creating a fully-fledged OOP object. In combination with Parent-Child functionality of Solidity, developers can take advantage of both approaches while minimising the downsides of either. From now on, this approach will be referred to as the hybrid approach. For example inheritance, encapsulation and polymorphism can be taken from the Parent-Child framework, while the object instance method state can be stored in a 'bytes4' data type within a 'struct' object inside a 'mapping' data structure, as illustrated in [Figure 7.a].

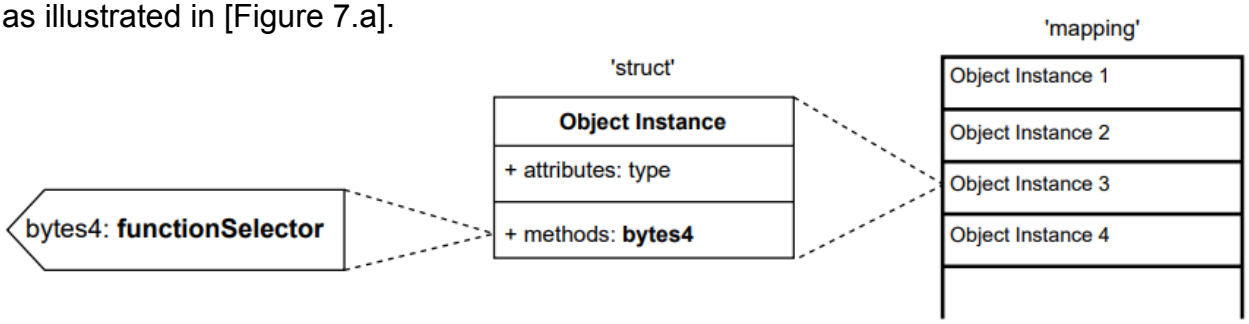


Figure 7.a - diagram of 'mapping' storage of the dObject instance

Security Concerns

There are indeed some security concerns related to Non-Transferable Tokens (NTTs) when it comes to the potential loss of private keys and wallet access. If NTTs represent important objects or data, such as an ID object or passport object, losing access to the wallet holding the dObject would have significant consequences. By default and design, the blockchain security model makes recovery of such wallets extremely difficult, if not completely impossible.

Hence, something must be done during the development of the dClass Smart Contract, as a precaution for cases like this. As discussed in the [15.] Dinhobl and Riedl's Ethereum Improvement Proposal (2022) "ERC-5883: Token Transfer by Social Recovery"', it could be possible to recover the lost tokens by the means of communal vote. Organised by the peer community, of holders of the same dClass token, such that when a sufficient number of peers votes "Yes" the token can be transferred from the locked account to a new account. This approach introduces an additional layer of security and recovery, relying on a trusted social network to safeguard assets in case of lost access or compromised accounts.

Overall, it is crucial to consider these security concerns and potential solutions when designing and implementing Non-Transferable Tokens, especially when they represent important and sensitive data and objects. Such a system would require a careful balance between security and recovery, ensuring that the network of peers is trusted and reliable while maintaining the decentralisation and security features of blockchain technology.

Game Theory

Game Theory Simulation Overview

In this section, the central hypothesis of the paper will be evaluated by employing three well researched and widely accepted Game Theory games. As illustrated in [Figure 8.] the games selected are, in order: The Centipede game, The Double Ultimatum game, and The Stag Hunt game.

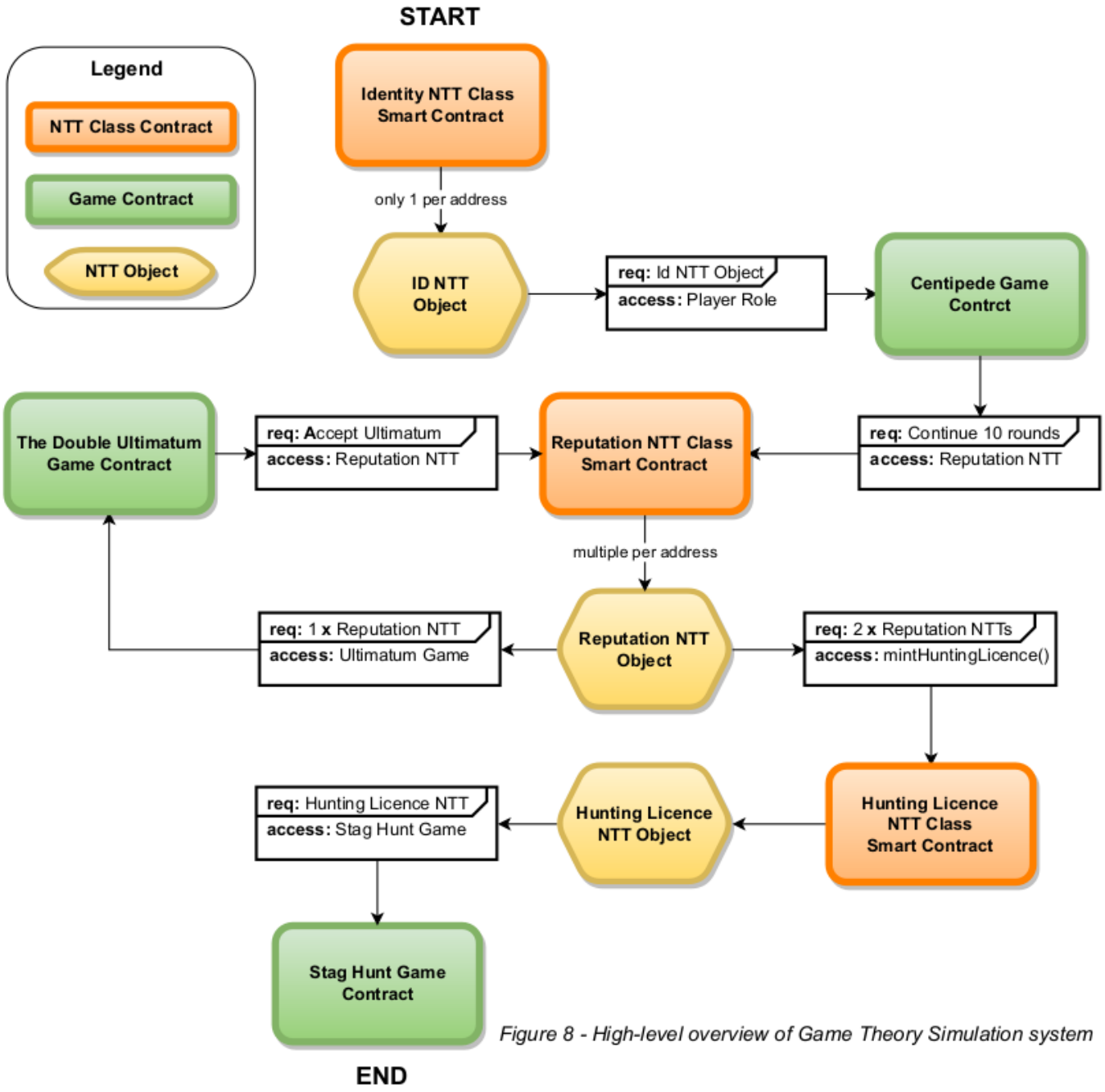


Figure 8 - High-level overview of Game Theory Simulation system

These games were specifically chosen due to their inherent Nash Equilibrium characteristics, and the alterations in Nash Equilibrium that occur as the game incentives are slightly shifted with the introduction of Non-Transferable Token (NTT) objects as rewards for completing the games in a mutually beneficial way. Each game will be thoroughly examined in the following sections; however, it is worth noting that the ultimate objective of the entire simulation is to choose the *{Stag}* option in the last game in the sequence, the Stag Hunt game. In a typical Stag Hunt game this would be a highly risky move as it could result in a payoff of zero for the player. Hence the player must have complete trust in their counterpart. The two preceding games serve to demonstrate the ability to play cooperatively and to foster trust between players, enabling them to confidently select the “high risk and high payoff” option of *{Stag}*, knowing that their counterpart will make the same choice.

The entire system is attempting to demonstrate how a decentralised cooperation of any kind involving an arbitrary number of individuals, can be organised and incentivised with the use of Non-Transferable Objects at the core of the organisation's structure.

The Centipede Game

The first game in the series, The Centipede Game, originally proposed by [8.] Rosenthal in his 1981. paper “Games of perfect information, predatory pricing and the chain-store paradox.”, is a sequential game of complete information that involves two players who alternate taking turns for 100 rounds, with each turn providing an opportunity to either *{Continue}* the game or *{End}* the game taking the current pot of payoffs. The Centipede Game is often used to study cooperation, trust, and iterative decision-making. It highlights the tension between short-term gains and long-term cooperation that can lead to higher payoffs for both players. As demonstrated in [Figure 9.] the game is typically presented as a series of nodes forming a path resembling a centipede. It is noteworthy that the implemented version of the game has only ten rounds for efficiency reasons, but this does not affect the logic of the game.

Figure 9. - Centipede Game logic structure and payoffs

With each passing round the pot increases in value in such a way that it first reduces for the player choosing *{continue}* and it only increases again if the other player chooses *{continue}* as well. Following the formula, current player payoff equals round number plus one and next player payoff equals round number minus one. Until the last round where in case the last player honours the cooperative trend the payoff for both players is equal to the round number and both get Reputation NTT token dObject.

In formal language: if $\mathscr{G}$ is the Centipede game then $\mathscr{G}(R, p_0, p_1)$, where $R, p_0, p_1 \in \mathbb{N}$, R is the maximum number of rounds, and p_0 and p_1 are the payoffs for player A (*Alice*) and player B (*Bob*), respectively. Players A and B alternate starting with player A, and may on each turn play a move $M \in \{End, Continue\}$. The game ends when *{End}* is played for the first time, otherwise upon R moves, if *{End}* is never played. If the game ends at round $r \in \{1, \ldots, R - 1\}$ and payoff $p \in \{p_0, p_1\}$ with player $e \in \{Alice, Bob\}$ making the final move. Then the outcome of the game is: player e gets $p = r+1$ and player e^* gets $p^* = r-1$. The asterisk (*) in this case denotes the other player. If the game *{Continues}* until round R then both players get $p_0, p_1 = R$ + *Reputation NTT* token.

In a purely rational framework for classic Centipede Game, the Nash Equilibrium solution suggests that the first player should "take" the pot immediately, as both players would try to maximise their individual payoffs. This can be proven by using the backwards induction method, starting from the end and taking only round number 10 into account Bob's best decision is to *{End}* the game, the same is valid for round number 9, Alice's best decision is to *{End}* the game, this can be repeated all the way to the start where Alice *{Ends}* the game in round 1. However, experimental results often deviate from this prediction, with players passing the pot multiple times before someone decides to take it.

This deviation is attributed to factors like altruism, fairness, trust, and reciprocity that are not fully captured by the traditional rational model, as discussed in the [16.](1995) Colin and Thaler's paper "Anomalies: Ultimatums, Dictators and Manners". The Centipede Game is therefore an interesting example of how human behaviour may not always align with purely rational decision-making in game theory. On top of experimental results keeping the players in game for longer, this version of the game has another incentive for them to see the game until the end. This being the fact that both get a "Reputation NTT" object, which allows them to play the next game in the sequence, thereby shifting the Nash Equilibrium from ending the game in round 1 to finishing the game in a cooperative manner.

The Double Ultimatum Game

The Double Ultimatum game is a variation of the classic Ultimatum Game, designed for two players who must make two sets of decisions instead of one. In the original Ultimatum game, first proposed in the [9.] (1961) John Harsanyi's paper "On the rationality postulates underlying the theory of cooperative games", Alice proposes a division of a fixed sum of money between herself and Bob. Bob can then either accept the proposed offer or reject it. If the responder accepts, both players receive the proposed shares; if the responder rejects, neither player receives anything. We can express this formally as: for the classic Ultimatum game the offer $x \in \{0, 100\}$, (p_0, p_1) where $x, p_0, p_1 \in \mathbb{N}$, and payoffs $p_0, p_1 = (x, 100 - x)$. The rational subgame-perfect equilibrium dictates if $p_0, p_1 = (99, 1)$, player 2 (Bob) will accept the offer because $1 > 0$, hence Bob will accept any offer $p_1 > 0$. This paints a very stark and merciless solution to the game.

Contrary to theoretical expectations, real world experiments reveal that players in the role of Alice tend to offer substantial shares to those in the role of Bob, usually ranging between 25% and 50% of the total value. Additionally, when player Bob is offered a small amount, often less than 20-25% of the total value, they are likely to reject these offers. This indicates that participants in the role of Bob exhibit a certain level of "spite" when presented with low shares, as they would rather receive nothing than allow player Alice to obtain a large portion of the total value. There are also differences in outcome of the game depending on the country of origin of the participants, indicating that there are cultural factors and the sense of "fairness" to consider. For a summary of the experimental findings and theories that explain this behaviour, refer to [10.] Camerer's "Strategizing in the Brain" (2003).

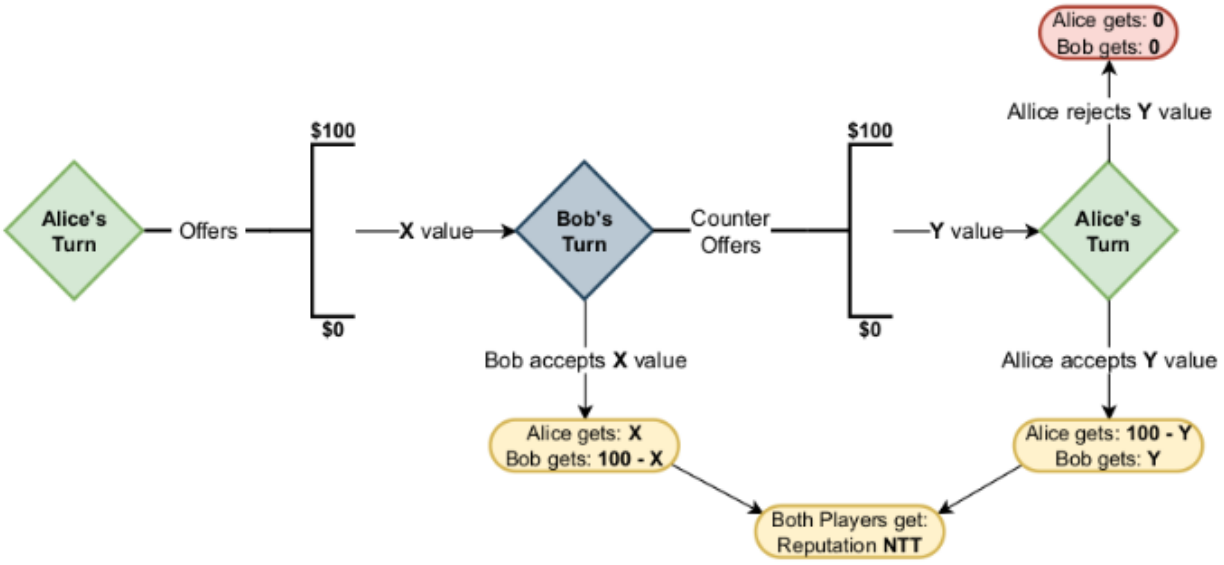


Figure 10. - The Double Ultimatum game logic structure and payoffs

In the Double Ultimatum game shown in [Figure 10.] and first described in the [11.](1972) "Bargaining theory" by Stahl, the process is repeated twice, with the players switching roles between the first and second rounds. A backwards induction solution shows that the outcome is the same as with the original game with the difference being that the second player has the upper hand. So the subgame-perfect equilibrium dictates if player Bob offers p_0, p_1 = *(1, 99),* player Alice will accept it.

In the modified version of the game presented in this study, an additional reward for accepting the offer is the Reputation NTT dObject. Following the outlined logic this would mean that the player Alice would even accept p_0, p_1 = *(0+NTT, 100+NTT),* as a payoff of 0+NTT is preferable to the payoff of 0.

However, as experimental results have shown, there is an aspect of “fairness” that must be taken into account. Therefore, it is possible that players, recognizing the Reputation NTT reward as an indication of fairness in their opponent, will not settle for anything less than the most fair outcome possible. This could lead to game outcomes approaching an even p_0, p_1 = *(50+NTT, 50+NTT)* split. While the experimental component is beyond the scope of this purely theoretical investigative paper, it would be intriguing to design a series of experiments based on this hypothesis. The pot of money could be further realised by making the Game Smart Contract distribute actual ETH coins with tangible value.

The Stag Hunt Game

The Stag Hunt game, also known as the Assurance game, is often attributed to Jean-Jacques Rousseau, who described the game in his book [12.] “A Discourse on Inequality” (1754). However it is noteworthy that the game was not introduced in a scientific paper but rather in Rousseau’s book where he discussed the game as a philosophical concept to explain social cooperation. The formal game theoretic analysis of the Stag Hunt game came later, with the development of game theory, in papers like the [13.](1988) “A general theory of equilibrium selection in games” by Harsanyi and Selten. The game is usually presented as a scenario where two hunters set out to capture prey. They can choose between hunting two hares or one stag in their hunting range. However, they can only bring the equipment necessary to catch one type of animal. They must decide which animal to hunt without observing the other hunter's choice, as they simultaneously select their equipment. The stag provides more meat than the combined hares, but both hunters

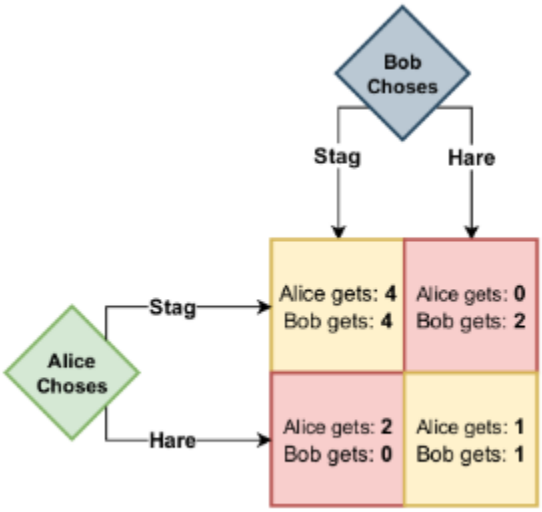


Figure 11. - The Stag Hunt game logic structure

must collaborate in capturing the stag. In contrast, if any hunter chooses to hunt hares, they can capture their prey independently. Interestingly, if one hunter pursues the stag and the other hunts hares, the hare hunter captures both hares, while the stag hunter returns empty-handed. This information can be condensed into a matrix representation as shown in [Figure 11.]

Formally, a stag hunt is a game featuring two pure strategy Nash equilibria, one that is risk-dominant, and another that is payoff-dominant, associated with a higher-payoff outcome that requires cooperation among players. In the matrix presented the [Stag, Stag] strategy represents the payoff-dominant Nash equilibrium, yielding the highest payoff for both players if they cooperate. The [Hare, Hare] strategy represents the risk dominant strategy, where both players choose a safer option that guarantees a lower payoff without relying on the other player's cooperation.

The game represents a situation where players must choose between pursuing individual interests with a guaranteed lower payoff or collaborating to achieve a higher collective payoff. The stag hunt game exemplifies the tension between individual security and collective welfare and highlights the importance of trust, communication, and coordination in achieving optimal outcomes in strategic situations.

In his 1990 paper [14.] "Nash Equilibria are not Self-Enforcing," Robert Aumann proposed a modification to the Stag Hunt game, allowing players to communicate before the game begins. At first glance, it might seem that the optimal [Stag, Stag] strategy would be guaranteed. However, let's consider a scenario where player Bob calls player Alice and claims they will hunt a stag, suggesting that player Alice should do the same. Surprisingly, there is a reason for player Bob to lie in this situation. If Bob had already decided to hunt a rabbit, their payoffs depend on Alice's decision. If Alice hunts a rabbit, Bob receives a payoff of one; if Alice hunts a stag, Bob's payoff is two. In this case, player Bob benefits from deceiving player Alice into hunting a stag. This uncertainty arises because Bob would prefer Alice to hunt a stag regardless of their own choice. Consequently, communication between the players doesn't necessarily reveal their true intentions, leaving room for deception and potentially resulting in the less favourable [Hare, Hare] outcome. This concept is an intriguing aspect of the Stag Hunt game to contemplate

Sequence of Games as a Simulation of a Decentralised System

The three games presented were specifically selected, and arranged in this specific sequence, to demonstrate the feasibility of using Non-Transferable Token dObjects as incentives to achieve cooperation in a decentralised system. In the original version, the first game in the sequence, the Centipede game, never advances past round one. The second game, the Double Ultimatum game, results in a highly uncooperative and greedy solution. And in the final game of the sequence, cooperation cannot be guaranteed even with the pre-game communication.

Figure 8.a - sequence of Games

For the Centipede game, incentives were slightly adjusted by rewarding an Reputation NTT dObject in the last round. Making the last player (Bob) in the last round (Round 10) choose between the payoff of $p = 11$ or payoff of $p = 10+NTT$. Given the difference of only 1 point for the opportunity to participate in another game, it seems likely that the Nash equilibrium will shift towards concluding the game in a cooperative manner.

The incentives in the second game, the Double Ultimatum game, are somewhat more ambiguous. On one hand, the strict logic suggests that the outcome will be even less cooperative than in the original game with payoff being $p_0, p_1 = (0+NTT, 100+NTT)$. However, as aforementioned experiments have shown ([10.] Camerer and [16.]Camerer and Thaler), the practical outcomes might differ significantly from what is expected. As previously stated, it is possible that players, perceiving the Reputation NTT as a sign of fairness in their opponent, will not accept anything less than the fairest outcome possible. This could lead to game outcomes approaching an even split of $p_0, p_1 = (50+NTT, 50+NTT)$. Further research would be needed to resolve this dilemma.

Lastly, the Stag Hunt game was specifically chosen for its long history of serving as an example for theorising about cooperation in society. As such, it was an ideal candidate to attempt demonstrating the possibility of guaranteeing a cooperative [Stag, Stag] outcome, even without the need for pre-game communication. To mint the Hunting Licence NTT the player had to show their ability to cooperate and had to prove it through their collection of Reputation NTT tokens. Uncooperative players cannot reach this point in the simulation. This would enable players to confidently play high-payoff (Stag) strategy, knowing that their counterpart is likely to do the same.

Game Contract Structure

To implement the entire system and play the games, only three Smart Contracts need to be deployed: [Appendix B] *2_CentipedeGame.sol*, *4_DoubleUltimatum.sol* and the *6_StagHunt.sol*. This was accomplished by developing the contracts in pairs, where each game has its corresponding NTT minting Smart Contract pair that acts as an dClass for the dObject, and from which the Game Contract inherits all of the necessary functionalities. This was made possible via the Parent-Child inheritance native to the Solidity programming language. And in synergy with the 'struct' dObjects described in the previous section of this paper the hybrid approach to Object Oriented Programming was achieved. Where the Parent-Child inheritance provides the description of the dClass and the 'struct' object provides the state of individual dObject instances.

IdentityNTT and CentipedeGame Contract pair

The first pair of contracts in the sequence can be found in files *1_IdentityNTT.sol* and *2_CentipedeGame.sol*, provided in the [Appendix B] at the end of the paper. It is highly recommended to open [Appendix B] in a separate window, allowing for the convenient reference to the code discussed in this section; for practicality a link is provided here: *Appendix B - Code*.

1_IdentityNTT.sol contract is acting as the dClass for the Identity Non-Transferable dObject. The *constructor()* function at the lines 12-18 is initialising the Non-Transferable Token, that is then associated with the 'struct' data type in the minting function *mintIDNTT()* at lines 54-66. This transforms the Non-Transferable Token into a

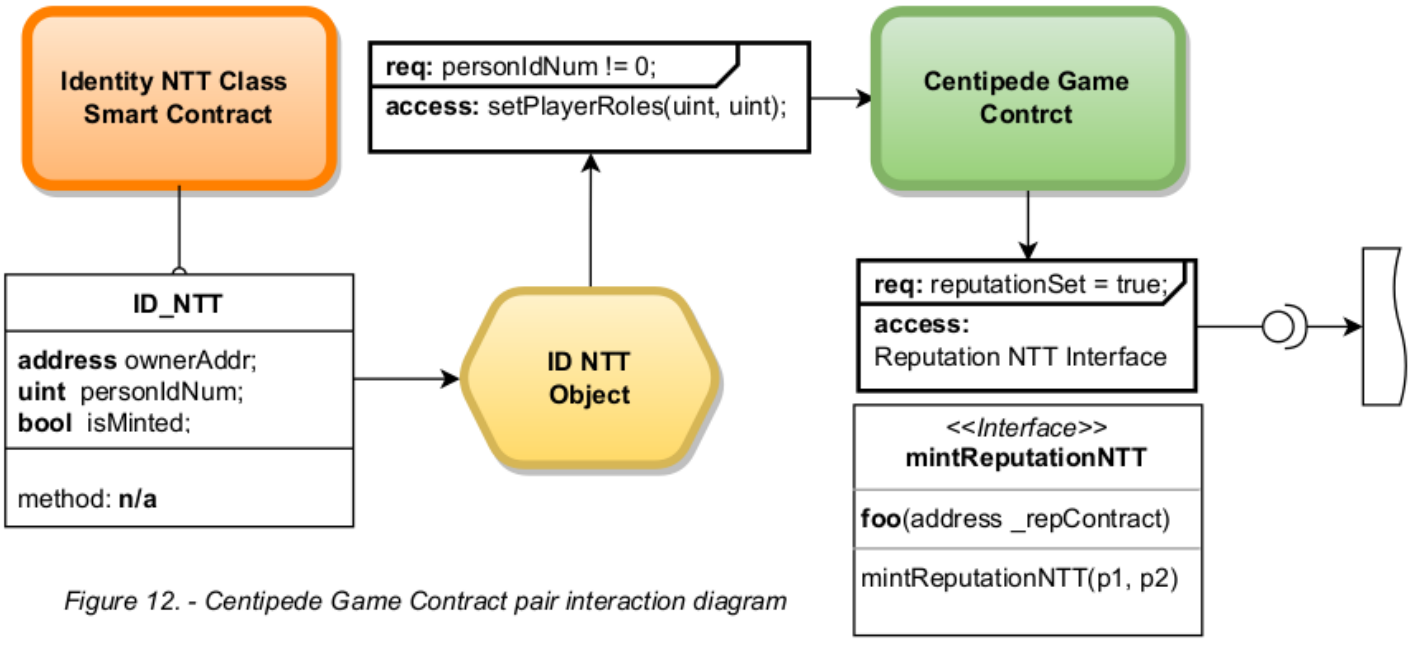


Figure 12. - Centipede Game Contract pair interaction diagram

Non-Transferable Object. The ‘struct’ data type, named *idObject{}*, is defined at the lines 34-41, and represents the simplest possible form of the ‘struct’ object, containing only three object attributes and no object methods. At line 44 mapping of the *idObjects* can be found, providing the most efficient way to conserve Ethereum network memory and reduce the ETH fees, while still being easily referenced and accessible in further code. User interaction with the main *1_IdentityNTT.sol* contract functions is depicted in [Figure 12.a].

Figure 12.a - main Identity NTT class functions

The *2_CentipedeGame.sol* contract [Appendix B], at line 8, inherits all of its counterparts functions and can access the *idObject{}* mapping. As depicted in [Figure 12.] user needs to mint the idObject to access the setPlayerRolesCent*()* function, defined at the lines 43-56, and gain access to the game functions. Game logic is straightforward with only two main functions, *continueGame()* and *stopGame()* lines 59-81, alongside a few supporting functions. User interaction with the main *2_CentipedeGame.sol* contract function can be seen in [Figure 12.b].

Upon successfully completing ten rounds, the user will gain access to the interface functionality connecting this set of contracts to the next set of contracts. The interface function is defined at the lines 11-14 and 18-27, and as shown in [Figure 12.] it allows the user to mint the next dObject, *ReputationNTT*.

Figure 12.b - user interaction with the Centipede Game Contract functions

ReputationNTT and DoubleUltimatum Contract pair

Access to *3_ReputationNTT.sol* file [Appendix B] functionality and its Reputation NTT dObject is only possible through an interface contract call from the previous game, the Centipede Game. This requirement is achieved by implementing a modifier function *onlyAuthorizedContracts()*, lines 63-67. The mapping of *authorizedContracts* at line 55 is the database containing the addresses of only two Game contracts that are permitted to mint Reputation NTT dObjects, as rewards for cooperatively completing the games. Consequently, *mintReputationNTT()* function, lines 69-94 mints two tokens simultaneously, one for each game participant.

The data type 'struct' object, named *reputationObject{}* is defined at the lines 34-41 and in addition to attributes, it has one method of type *bytes4.* As discussed in the previous section of this paper, the low-level EVM function selector is stored at this memory location. The function selector is computed at line 47, and the first 4 bytes of the *keccak256* hash can be observed in [Figure 13.a] as an output of the *getObject()* function (*bytes4: 0x5315c3b8*). This is then utilised to call the interface of the subsequent set of Smart Contracts, which will be discussed in subsequent section. As with the previous contract, *3_ReputationNTT.sol* is inherited by its counterpart Game Contract, the Double Ultimatum game.

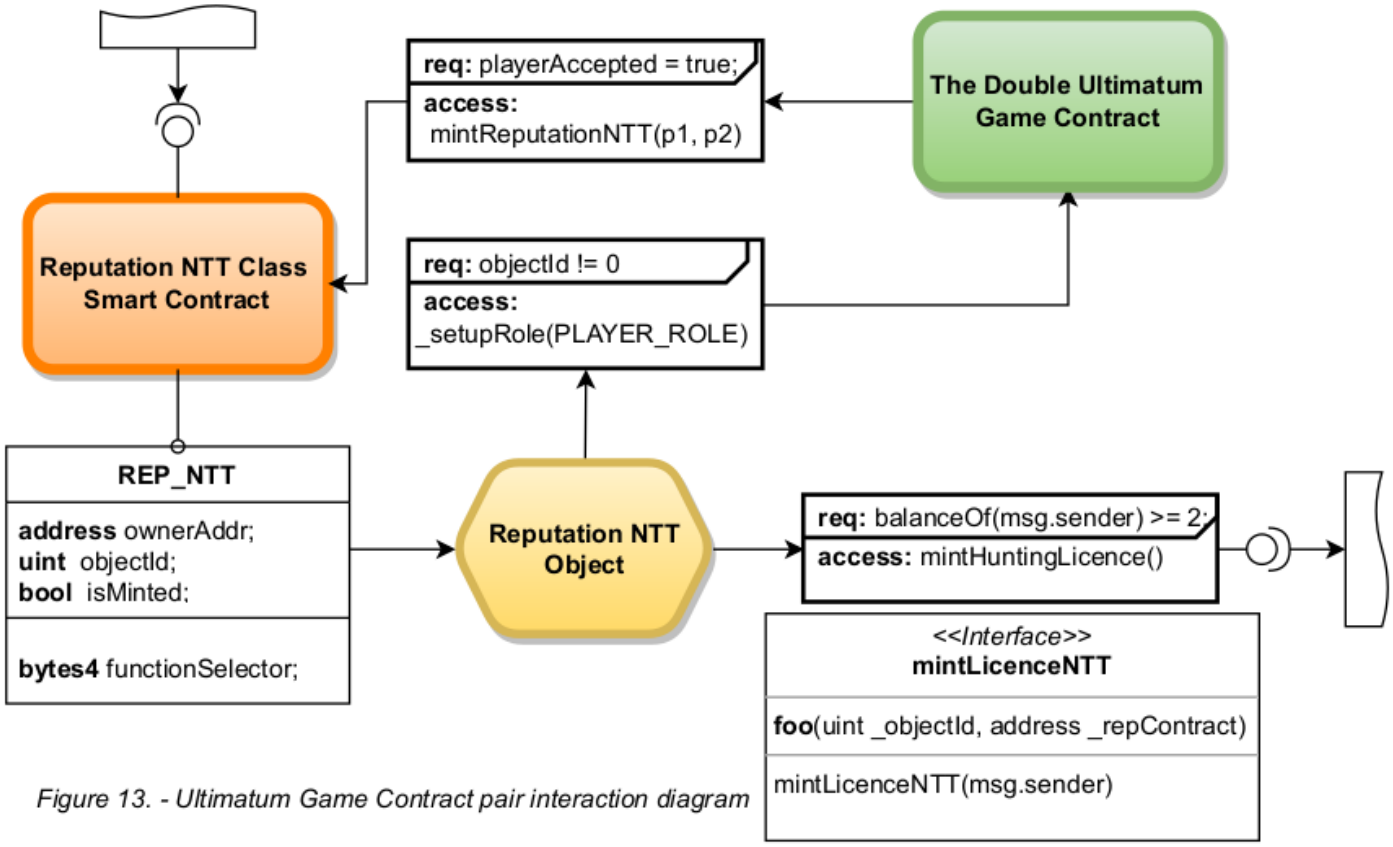


Figure 13. - Ultimatum Game Contract pair interaction diagram

As demonstrated in [Figure 13.] the functionality of *4_DoubleUltimatum.sol* contract [Appendix B] is accessible when the user owns at least one Reputation NTT token. This prerequisite enables the *setPlayerRolesUlt()* function, at lines 59-72 to be called. The game can then begin, as described in the previous section of this paper, with players setting their demands for dividing the pool of money, line 47. The game functions are extremely simple, with the majority of the game logic being in the *gameResults()* function at lines 123-139. This same function, if the game is successfully resolved, calls the *getRewardReputationTokens()* function, defined at the lines 38-40, which automatically mints additional two Reputation NTT dObjects, one for each player.

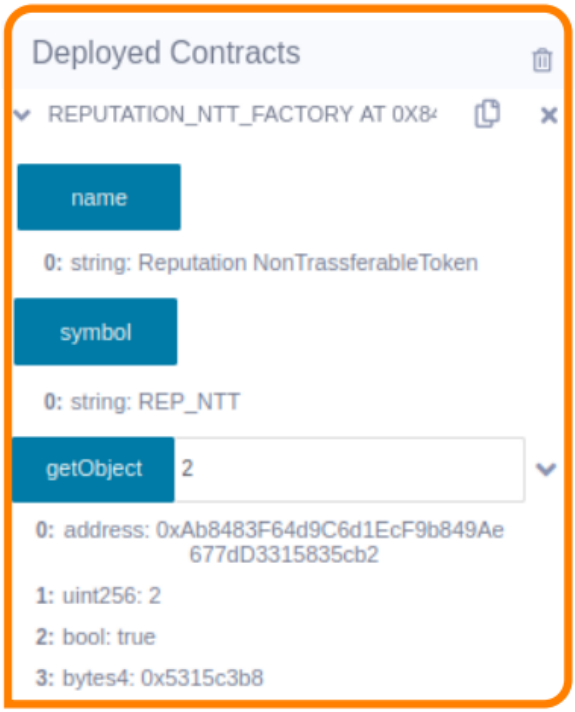


Figure 13.a - main Reputation NTT class functions

At this point, players will each own two Reputation NTTs, and as illustrated in [Figure 13.] this will allow them to make an interface call to the subsequent set of Smart Contracts in the sequence. The interface is established at the lines 10-12, and it is invoked at the lines 17-25. It is noteworthy that the call is using the low-level Ethereum Virtual Machine (EVM) function selector, at line 22, which is obtained from the Reputation NTT dObject.

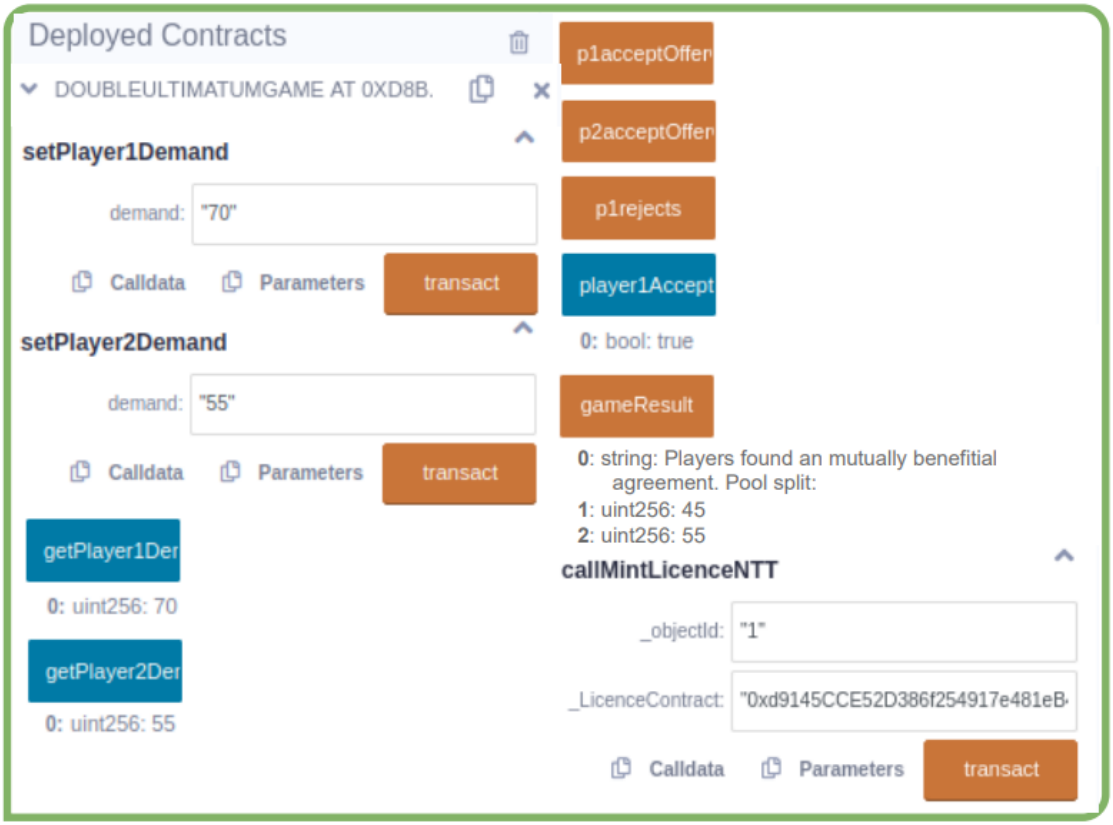


Figure 13.b - user interaction with the Ultimatum Game Contract functions

LicenceNTT and StagHunt Contract pair

Regarding the *5_LicenceNTT.sol* Smart Contract [Appendix B], to avoid repetition of previously discussed code elements, it can be stated that everything valid for the prior two dClass minting Contracts is also applicable here.

With the main difference being the kind of dObject produced. For the *licenceObject{}*, the 'struct' object has three attributes, same as the others, and one boolean value serving as a function flag: *joinHuntFlag;* at line 40. The second difference is the presence of the *setFunctionFlagToTrue()* function at the lines 47-50. This function is not strictly necessary, the token could have been initiated with the flag already set to *true*, but it was added regardless to demonstrate the option for users to toggle the flag on and off. As illustrated in [Figure 14.] the *joinHuntFlag = true* is needed to access the Hunter Role.

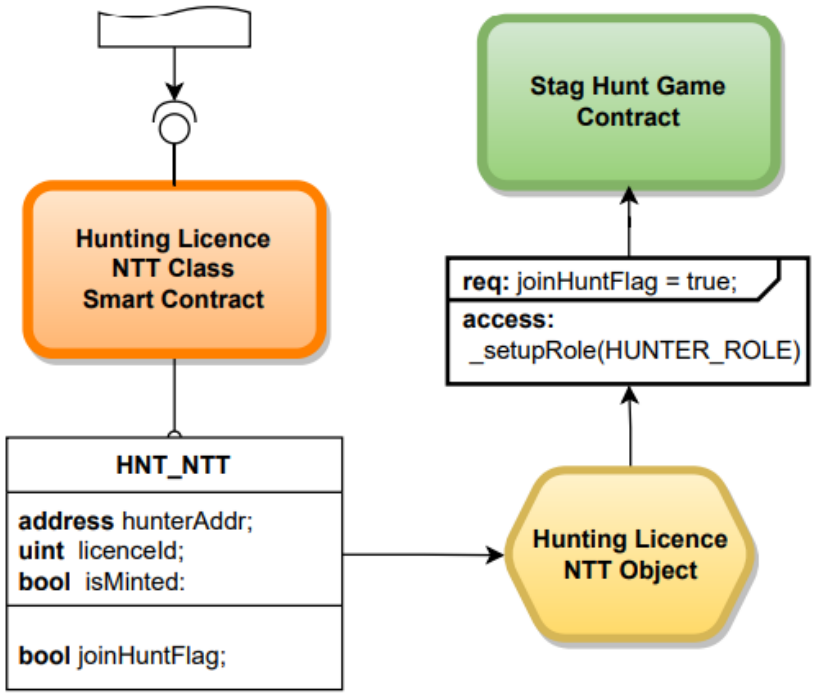


Figure 14. - Stag Hunt Game Contract pair interaction diagram

The *6_StagHunt.sol* Smart Contract [Appendix B] contains the game logic for the Stag Hunt game described in the Game Theory section of the paper. The game commences when the *setPlayerRolesHunt()* function, found at lines 24-37 is invoked. The game functions are once again very simple and straightforward, with the primary function being *gameResults()* lines 65-77, that simultaneously calculates payoffs for both players, and as shown in [Figure 14.b] returns a 'string' notification about the results and 'uint' payoffs.

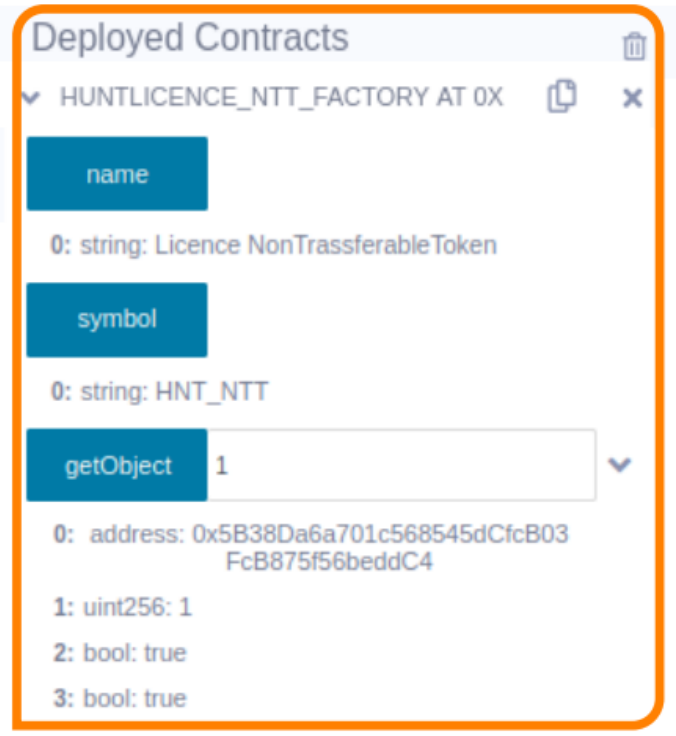


Figure 14.a - main Licence NTT class functions

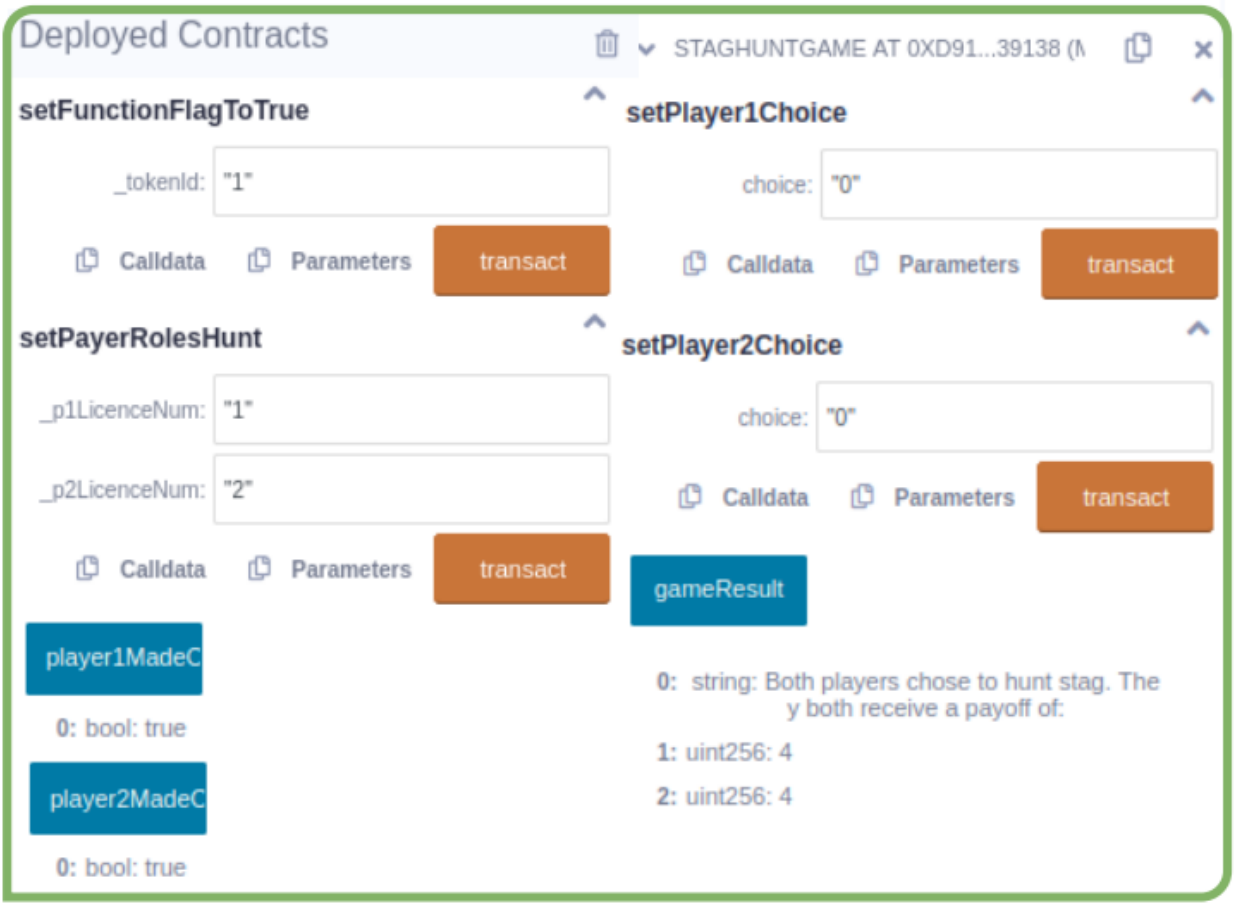


Figure 14.b - user interaction with the Stag Hunt Game Contract functions

Conclusion

Simulation Results Discussion

From a technical standpoint, the entire simulation system works as intended. The hybrid approach to Object Oriented Programming, by utilisation of Parent-Child approach where it was appropriate, and utilisation of ‘struct’ approach to store the instance state proved the benefits of such hybrid framework conceptualisation.

Furthermore the simulation also utilises three distinct sorts of ‘struct’ dObjects, thus validating the central thesis of the paper. Namely ‘struct’ *idObject{}*, with only attributes, ‘struct’ *reputationObject{},* with a function selector method and ‘struct’ *licenceObject{}*, with a function flag method. This simple proof of concept shows possible real world usage of NTTs as abstract dOjects.

Further research could involve developing a front-end user interface and organising test players to gather data on the practical application of game-theoretical incentives. An additional layer of incentives could be explored by assigning tangible value to the payoffs for each game, such as using ETH coins. For instance, in the Double Ultimatum game, the currently hardcoded value *uint poolOfMoney = 100* could easily be modified to distribute ETH instead. However, this falls beyond the scope of the current paper and must be left for future studies.

Non-Transferable Tokens as dObjects Discussion

In conclusion, the dObject Oriented Framework presented in this paper successfully demonstrates the potential and flexibility of Non-Transferable Tokens as dObjects in various use cases outlined in the extended abstract. Through the utilisation of unique properties of dObjects, a novel approach to constructing decentralised systems that provide improved security, transparency, and reliability in comparison to conventional centralised models has been showcased. This paper has shown, through the specific series of Game Theory games, and the change in incentives caused by the utilisation of NTT dObjects, the value of such approach in the construction of decentralised organisations across multiple industries.

The domain of Non-Transferable Tokens is still in its infancy, and the trajectory of development could benefit from a concept like dObject. The creation of specialised development tools, such as libraries tailored to this framework, could further enhance its utility, as some workarounds implemented in the code to achieve the paper’s objectives seem too cumbersome and impractical for large scale adoption. Libraries could be developed not only for the Solidity programming language but also for classical OOP

languages like Java and C++, allowing developers to designate significant objects for blockchain storage in the form of dObjects. This would facilitate the integration of Web3 into conventional software development.

Incorporating dObjects into different domains, such as finance, gaming, supply chain management, and digital identity, enables the exploration of new possibilities for creating decentralised applications that effectively address the unique challenges and requirements of each sector. The implementation of dObjects in permissioning access, cooperatives, local currencies, market design experiments, and democratic mechanism design exemplifies the versatility and adaptability of this framework. The dObject Oriented Framework provides a solid foundation for future development and research in the field of decentralised systems. This will ultimately lead to more accessible, efficient, and user-friendly decentralised applications that cater to a wide range of needs and use cases.

It should be noted that the Ethereum blockchain may not be the ideal platform for implementing this framework. Although Ethereum was chosen for this project due to its status as the oldest, most recognizable, and extensively documented smart contract platform, its primary focus is on financial applications and settlements, which are well-served by the current Solidity framework. Meanwhile, dObjects, as a broad and abstract concept, may find a better-suited blockchain network for integration. When initiating a new project, it is essential to consider the compatibility of the chosen blockchain platform with the project's scope and requirements.

In summary, the utilisation of dObject Oriented Programming (dOOP) in conjunction with Smart Contracts has the potential to revolutionise the way we design and interact with decentralised organisations. The potential implementation of dObjects in the various use cases discussed in the extended abstract serves as a testament to the power and versatility of this novel approach to building decentralised systems, setting the stage for further exploration, experimentation, and innovation in the rapidly evolving world of Blockchain technology.

* * *

Acknowledgments

I am deeply grateful to my sister, Dora Lucija Ward, for her steadfast love and support throughout my four years of college. Her financial assistance, encouragement, and inspiration have been invaluable, and I cannot imagine how I would have managed this journey without her by my side.

I am profoundly grateful to my best friend in Cork, Oliver Frank Cunnington, who generously lent me his high-quality laptop for a year when he noticed I was struggling with my outdated PC. Without it, writing this paper would have been incredibly challenging. I also appreciate his companionship throughout our college years, his assistance with my studies, and the time he dedicated to proofreading my paper.

I extend my sincere gratitude to Felix Hieser, a professional Solidity Security Auditor at Byterocket GmbH (https://byterocket.com/). Despite being a complete stranger in a Discord chat room, he generously took the time to answer my questions. His guidance on which Solidity development framework to use and his insights on the subtle nuances of the EVM compiler proved invaluable to me.

I further extend my gratitude to Damien Cunningham, a former colleague, for generously taking the time to proofread my paper and provide valuable feedback on my use of the English language. His expertise and attention to detail have greatly contributed to the quality of this work.

Lastly, but certainly not least, I am sincerely grateful to Dr. James Doherty, my project supervisor, for approving my project proposal and providing steadfast support for my idea. While I may not have been the best at maintaining regular contact during this project, I truly enjoyed working on a topic I am passionate about, and I am profoundly appreciative of the opportunity.

Declaration of Originality

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Date: 23.04.2023

Signed: Ozzak Jure Matić

Appendix A - Citations

[1.] Antonopoulos, Andreas M. “Mastering Ethereum: Building Smart Contracts and DApps” O’Reilly, (2018). https://github.com/ethereumbook/ethereumbook

[2.] Lippman, S. B., & Lajoie, J. “C++ Primer (4th ed.)”. Boston, MA: Addison-Wesley. (2003) https://picture.iczhiku.com/resource/eetop/SHkThjUojASZoCxm.pdf

[3.] Weyl, Eric Glen, Puja Ohlhaver, and Vitalik Buterin. "Decentralized Society: Finding Web3's Soul." SSRN, May 10, 2022. https://ssrn.com/abstract=4105763 or http://dx.doi.org/10.2139/ssrn.4105763

[4.] Tim Daubenschütz (@TimDaub), "ERC-4973: Account-bound Tokens [DRAFT]," *Ethereum Improvement Proposals*, no. 4973, April 2022. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-4973

[5.] Tim Daubenschütz (@TimDaub), Anders (@0xanders), "ERC-5192: Minimal Soulbound NFTs," *Ethereum Improvement Proposals*, no. 5192, July 2022. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-5192

[6.] Omar Aflak (@omaraflak), Pol-Malo Le Bris, Marvin Martin (@MarvinMartin24), "ERC-4671: Non-Tradable Tokens Standard [DRAFT]," *Ethereum Improvement Proposals*, no. 4671, January 2022. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-4671

[7.] Bruno Škvorc (@Swader), Francesco Sullo (@sullof), Steven Pineda (@steven2308), Stevan Bogosavljevic (@stevyhacker), Jan Turk (@ThunderDeliverer), "ERC-6454: Minimalistic Non-Transferable NFTs [DRAFT]," *Ethereum Improvement Proposals*, no. 6454, January 2023. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-6454

[8.] Rosenthal, R. W. “Games of perfect information, predatory pricing and the chain-store paradox.” (1981). Journal of Economic Theory, 25(1), 92-100. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a7552bd3302f6dee7c0b74dea6fc486f794db0b8

[9.] Harsanyi, J. C. (1961). On the rationality postulates underlying the theory of cooperative games. Journal of Veterinary Dentistry, 5(2), 15–18. https://doi.org/10.1177/089875640201900102

[10.] Colin F. Camerer ,Strategizing in the Brain". Science 300, 1673-1675(2003). https://www.science.org/doi/full/10.1126/science.1086215

[11.] Stahl, Ingolf. "Bargaining theory (stockholm school of economics)." Stockholm, Sweden (1972). https://www.ssc.wisc.edu/~dquint/econ522%202007/econ%20522%20lecture%204%20(property%20law).doc

[12.] Rousseau, Jean-Jacques. (1754). A Discourse on Inequality. Marc-Michel https://books.google.hr/books?hl=en&lr=&id=n0tdG2qZFJUC&oi=fnd&pg=PA1&dq=A+Discourse+on+Inequality&ots=q-vmK8bxsU&sig=Nz8Ss-gL944eB2qp-Z9DcfdWK3E&redir_esc=y#v=onepage&q=A%20Discourse%20on%20Inequality&f=false

[13.] Harsanyi, J. C., & Selten, R. (1988). A general theory of equilibrium selection in games. MIT Press. https://ideas.repec.org/b/mtp/titles/0262582384.html

[14.] Aumann, Robert. "Nash equilibria are not self-enforcing, in ‘‘Economic Decision-Making: Games, Econometrics and Optimization''(JJ Gabszewicz, J.-F. Richard, and LA Wolsey, Eds.)." (1990) https://dial.uclouvain.be/pr/boreal/object/boreal:75915

[15.] Erhard Dinhobl (@mrqc), Kevin Riedl (@wsdt), "ERC-5883: Token Transfer by Social Recovery [DRAFT]," *Ethereum Improvement Proposals*, no. 5883, July 2022. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-5883

[16.] Camerer, Colin F., and Richard H. Thaler. 1995. "Anomalies: Ultimatums, Dictators and Manners." Journal of Economic Perspectives, 9 (2): 209-219. https://www.aeaweb.org/articles?id=10.1257/jep.9.2.209

Figure Sources

Figure *. - any figure not specified in this section is edited by the paper author

Figure 2. - Antonopoulos, Andreas M. Mastering Bitcoin: Programming The Open Blockchain. O'Reilly 2nd Edition (2017) page 191

Appendix B - Code

- Project Github repository:
    - https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation
- @openzeppelin library:
    - website: https://www.openzeppelin.com/
    - Github repository: https://github.com/openzeppelin

- **Table of Content:**
    - *1_IdentityNTT.sol*
    - *2_CentipedeGame.sol*
    - *3_ReputationNTT.sol*
    - *4_DoubleUltimatum.sol*
    - *5_LicenceNTT.sol*
    - *6_StagHunt.sol*

1_IdentityNTT.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/1_IdentityNTT.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.9;
03
04 // @openzeppelin libraries
05 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
06 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
07 import "@openzeppelin/contracts/utils/Counters.sol";
08
09 contract Identity_NTT_Factory is ERC721, ERC721Enumerable {
10
11     // ========== Lifecycle Methods ==========
12     address nttContractAddress;
13     constructor() ERC721("Identity NonTrassferableToken", "ID_NTT") {
14         // Start Token counter
15         _tokenIdCounter.increment();
16         // Store contract address
17         nttContractAddress = address(this);
18     }
19
20     // Override transfer function to make the token Non-Transferable
21     function _beforeTokenTransfer(
22         address from,
23         address to,
24         uint256 tokenId,
25         uint256 batchSize)
26         internal override(ERC721, ERC721Enumerable) {
27         require(from == address(0) || to == address(0),
28                 "ERROR: You can't send NonTransferableTokens :P");
29         super._beforeTokenTransfer(from, to, tokenId, batchSize);
30     }
31
32     // ========== NonTransferable Object ==========
33     // struct type that will store the state of the Identity NTT Object
34     struct idObject {
35         // Object Attributes
36         address ownerAddr;
37         uint personIdNum;
38         bool isMinted;
39         // Object Methods
40             // n/a
41     }
42
43     // mapping of the objects
44     mapping(uint => idObject) mapOfObjects;
```

```
45
46      // ========== Property variables ==========
47      // using counter to give each token an ID
48      using Counters for Counters.Counter;
49      Counters.Counter private _tokenIdCounter;
50
51      // uint256 public MINT_PRICE = 100 gwei; // 0.0000001 ETH
52
53      // ========== Minting Functions ==========
54      function mintIdNTT() public {
55          require(balanceOf(msg.sender) < 1,
56                  "Only 1 ID token per address allowed");
57          // require(msg.value >= MINT_PRICE, "Not enough ether sent");
58          uint256 tokenId = _tokenIdCounter.current();
59          _tokenIdCounter.increment();
60          _safeMint(msg.sender, tokenId);
61
62          // link the token to the struct Object
63          idObject storage _idObject = mapOfObjects[tokenId];
64          _idObject.ownerAddr = msg.sender;
65          _idObject.personIdNum = tokenId;
66          _idObject.isMinted = true;
67      }
68
69      //  ========== Getter Functions ==========
70      function getObject(uint _tokenId) view public returns (address, uint, bool) {
71          idObject memory _idObject = mapOfObjects[_tokenId];
72          return (_idObject.ownerAddr, _idObject.personIdNum, _idObject.isMinted);
73      }
74
75      //  ========== Other Functions ==========
76      // The following functions are overrides required by Solidity
77      // Helping the interoperability of ERC721 standard
78      function supportsInterface(bytes4 interfaceId)
79          public view virtual override(ERC721, ERC721Enumerable) returns (bool) {
80          return super.supportsInterface(interfaceId);
81      }
82  }
```

2_CentipedeGame.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/2_CentipedeGame.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.0;
03
04 // import @openzeppelin libraries
05 // inheriting the IdentityNTT minting contract
06 import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
07 import "@openzeppelin/contracts/access/AccessControl.sol";
08 import "./1_IdentityNTT.sol";
09
10 // creating interface pointing to the next game's NTT minting function
11 interface Ireputation {
12     function mintReputationNTT(address player1, address player2) external;
13     function addAuthorizedContract(address _callingContractAddress) external;
14 }
15
16 contract CentipedeGame is AccessControl, Identity_NTT_Factory {
17
18     // Interface function calls to Reputation NTT Factory Contract
19     function callMintReputationNTT(address _reputationContract) external {
20         require(reputationSet,
21             "You can't call Reputation NTT Minting Contract. Finish the game.");
22         // add this contract address to authorized contracts
23         Ireputation(_reputationContract).addAuthorizedContract(address(this));
24         // mint Reputation Token for both players
25         Ireputation(_reputationContract).mintReputationNTT(player1, player2);
26         reputationSet = false;
27     }
28
29     // ========== Variables ==============
30     bytes32 public constant PLAYER_ROLE = keccak256("PLAYER_ROLE");
31
32     address public player1;
33     address public player2;
34     uint public player1payoff;
35     uint public player2payoff;
36
37     uint public roundsPlayed;
38     bool public isGameEnded;
39     bool public reputationSet;
40
```

```
41      // Function to set player roles for the Centipede game
42      // Require ID NTT Token to get access to player roles
43      function setPlayerRolesCent(uint _player1idNum, uint _player2idNum) public {
44          idObject memory _idObject1 = mapOfObjects[_player1idNum];
45          idObject memory _idObject2 = mapOfObjects[_player2idNum];
46
47          require(_idObject1.personIdNum != 0, "Player 1 has to have an ID Token");
48          require(_idObject2.personIdNum != 0, "Player 2 has to have an ID Token");
49          require(_player1idNum != _player2idNum,
                                        "Players cannot have the same ID Token");
50
51          _setupRole(PLAYER_ROLE, ownerOf(_player1idNum));
52          _setupRole(PLAYER_ROLE, ownerOf(_player2idNum));
53
54          player1 = ownerOf(_player1idNum);
55          player2 = ownerOf(_player2idNum);
56      }
57
58      // ============ Game Logic ============
59      function continueGame() public onlyRole(PLAYER_ROLE) {
60          roundsPlayed++;
61          require(!isGameEnded, "The game has already ended");
62
63          if (roundsPlayed < 10) {
64              require(msg.sender == getLastPlayer(),
65                  "The same player cannot play twice in a row");
66              calculateCurrentPayoffs();
67          } else {
68              player1payoff = roundsPlayed;
69              player2playoff = roundsPlayed;
70              isGameEnded = true;
71              reputationSet = true;
72          }
73      }
74
75      function stopGame() public onlyRole(PLAYER_ROLE) {
76          require(!isGameEnded, "The game has already ended");
77
78          roundsPlayed++;
79          isGameEnded = true;
80          calculateCurrentPayoffs();
81      }
82
```

```
83      function readCurrentPayoffs() public view returns (uint, uint) {
84          return (player1payoff, player2payoff);
85      }
86
87      function calculateCurrentPayoffs() private {
88          if (msg.sender == player1) {
89              player1payoff = roundsPlayed + 1;
90              player2payoff = roundsPlayed - 1;
91          } else {
92              player1payoff = roundsPlayed - 1;
93              player2 playoff = roundsPlayed + 1;
94          }
95      }
96
97      function getLastPlayer() public view returns (address) {
98          if (roundsPlayed % 2 == 1) {
99              return player1;
100          } else {
101              return player2;
102          }
103      }
104
105      //  ========== Other Functions ==========
106      // The following functions are overrides required by Solidity
107      // Helping the interoperability of ERC721 standard
108      function supportsInterface(bytes4 interfaceId)
109      public view override(AccessControl, Identity_NTT_Factory) returns (bool) {
110          return super.supportsInterface(interfaceId);
111      }
112  }
```

3_ReputationNTT.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/3_ReputationNTT.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.9;
03
04 // @openzeppelin libraries
05 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
06 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
07 import "@openzeppelin/contracts/utils/Counters.sol";
08
09 contract Reputation_NTT_Factory is ERC721, ERC721Enumerable {
10
11     // ========== Lifecycle Methods ==========
12     address repNttContractAddress;
13     constructor() ERC721("Reputation NonTrassferableToken", "REP_NTT") {
14         // Start Token counter
15         _tokenIdCounter.increment();
16         // Store contract address
17         repNttContractAddress = address(this);
18     }
19
20     // Override transfer function to make the token Non-Transferable
21     function _beforeTokenTransfer(
22         address from,
23         address to,
24         uint256 tokenId,
25         uint256 batchSize)
26         internal override(ERC721, ERC721Enumerable) {
27         require(from == address(0) || to == address(0),
28                 "ERROR: You can't send NonTransferableTokens :P");
29         super._beforeTokenTransfer(from, to, tokenId, batchSize);
30     }
31
32     // ========== NonTransferable Object ==========
33     // struct type that will store the state of the Reputation NTT Object
34     struct reputationObject {
35         // Object Attributes
36         address ownerAddr;
37         uint objectId;
38         bool isMinted;
39         // Object Methods
40         bytes4 functionSelector;
41     }
42
43     // mapping of the objects
44     mapping(uint => reputationObject) mapOfRepObjects;
```

```
45
46      // computing bytes4 Function Selector for function mintLicenceNTT
47      bytes4 private constant MINT_LICENCE_NTT_SELECTOR =
                                        bytes4(keccak256("mintLicenceNTT(address)"));
48
49      // ========== Property variables ==========
50      // using counter to give each token an ID
51      using Counters for Counters.Counter;
52      Counters.Counter private _tokenIdCounter;
53
54      // ========== Minting Functions ==========
55      mapping(address => bool) public authorizedContracts;
56
57      // Function to add an authorized contract
58      function addAuthorizedContract(address _contractAddress) external {
59          authorizedContracts[_contractAddress] = true;
60      }
61
62      // Modifier to check if the caller is an authorized contract
63      modifier onlyAuthorizedContracts() {
64          require(authorizedContracts[msg.sender],
65              "Caller is not an authorized contract");
66          _;
67      }
68
69      function mintReputationNTT(address player1, address player2)
70                                                  external onlyAuthorizedContracts {
71          // mint token for player 1
72          uint256 tokenId1 = _tokenIdCounter.current();
73          _tokenIdCounter.increment();
74          _safeMint(player1, tokenId1);
75
76          // link the token for player 1 to the struct Object
77          reputationObject storage _repObject = mapOfRepObjects[tokenId1];
78          _repObject.ownerAddr = player1;
79          _repObject.objectId = tokenId1;
80          _repObject.isMinted = true;
81          _repObject.functionSelector = MINT_LICENCE_NTT_SELECTOR;
82
83          // mint token for player 2
84          uint256 tokenId2 = _tokenIdCounter.current();
85          _tokenIdCounter.increment();
86          _safeMint(player2, tokenId2);
87
```

```
88          // link the token for player 2 to the struct Object
89          reputationObject storage _repObject2 = mapOfRepObjects[tokenId2];
90          _repObject2.ownerAddr = player2;
91          _repObject2.objectId = tokenId2;
92          _repObject2.isMinted = true;
93          _repObject2.functionSelector = MINT_LICENCE_NTT_SELECTOR;
94      }
95
96      //  ========== Getter Functions ==========
97      function getObject(uint _tokenId) view public
                                              returns (address, uint, bool, bytes4) {
98          reputationObject memory _repObject = mapOfRepObjects[_tokenId];
99          return (_repObject.ownerAddr, _repObject.objectId, _repObject.isMinted,
100                 _repObject.functionSelector);
101     }
102
103     function getMyBalance() public view returns (uint256) {
104         return balanceOf(msg.sender);
105     }
106
107     //  ========== Other Functions ==========
108     // The following functions are overrides required by Solidity
109     // Helping the interoperability of ERC721 standard
110     function supportsInterface(bytes4 interfaceId)
111         public view virtual override(ERC721, ERC721Enumerable) returns (bool) {
112         return super.supportsInterface(interfaceId);
113     }
114 }
```

4_DoubleUltimatum.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/4_DoubleUltimatum.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.0;
03
04 // import @openzeppelin libraries
05 // inheriting the ReputationNTT minting contract
06 import "@openzeppelin/contracts/access/AccessControl.sol";
07 import "./3_ReputationNTT.sol";
08
09 // creating interface pointing to the next game's NTT minting function
10 interface IhuntLicence {
11     function mintLicenceNTT(address hunter) external;
12 }
13
14 contract DoubleUltimatumGame is AccessControl, Reputation_NTT_Factory {
15
16     // Interface function call to Reputation NTT Factory Contract
17     function callMintLicenceNTT(uint _objectId,address _LicenceContract) external {
18         require(balanceOf(msg.sender) >= 2,
19             "You need 2 Reputation Tokens to call Hunting Licence NTT Contract.");
20         reputationObject memory _repObject = mapOfRepObjects[_objectId];
21         // low level call using functionSelector
22         bytes memory data =
                       abi.encodeWithSelector(_repObject.functionSelector, msg.sender);
23         (bool success, ) = _LicenceContract.call(data);
24         require(success, "Function call failed");
25     }
26
27     // variables needed for the constructor initialization
28     address public contractAddress;
29     Reputation_NTT_Factory reputationNTT;
30     // constructor to initialize the Reputation_NTT_Factory at the same address
31     constructor() {
32         contractAddress = address(this);
33         reputationNTT = Reputation_NTT_Factory(contractAddress);
34         authorizedContracts[contractAddress] = true;
35     }
36
37     // minting Reputation Tokens
38     function getRewardReputationTokens() public {
39         reputationNTT.mintReputationNTT(player1, player2);
40     }
41
```

```
42      // ========== Variables =============
43      bytes32 constant PLAYER_ROLE = keccak256("PLAYER_ROLE");
44
45      address public player1;
46      address public player2;
47      uint public poolOfMoney = 100;
48
49      uint public player1Demand;
50      uint public player2Demand;
51      bool public player1Accepted;
52      bool public player1Rejected;
53      bool public player2Accepted;
54      bool player1ProposedAnOffer;
55      bool player2ProposedAnOffer;
56
57      // Function to set player roles for the Ultimatum game
58      // require 1 Reputation NTT Token per player to get access to player roles
59      function setPlayerRolesUlt(uint _p1repIdNum, uint _p2repIdNum) public {
60          reputationObject memory _repObject1 = mapOfRepObjects[_p1repIdNum];
61          reputationObject memory _repObject2 = mapOfRepObjects[_p2repIdNum];
62
63          require(_repObject1.objectId != 0,
                                        "Player 1 has to have 1 Reputation Token");
64          require(_repObject2.objectId != 0,
                                        "Player 2 has to have 1 Reputation Token");
65          require(_p1repIdNum != _p2repIdNum,
                                        "Players cannot have the same Token");
66
67          _setupRole(PLAYER_ROLE, ownerOf(_p1repIdNum));
68          _setupRole(PLAYER_ROLE, ownerOf(_p2repIdNum));
69
70          player1 = ownerOf(_p1repIdNum);
71          player2 = ownerOf(_p2repIdNum);
72      }
73
```

```
74      // ============ Game Logic OK ============
75      function getPlayer1Demand() public view returns (uint) {
76          return player1Demand;
77      }
78
79      function getPlayer2Demand() public view returns (uint) {
80          return player2Demand;
81      }
82
83      function setPlayer1Demand(uint demand) public onlyRole(PLAYER_ROLE) {
84          require(msg.sender == player1, "Only Player 1 can set his demand");
85          require(demand <= poolOfMoney, "Player 1 demand exceeds pool of money");
86          require(demand >= 0, "Demand has to be positive int");
87
88          player1Demand = demand;
89          player1ProposedAnOffer = true;
90      }
91
92      function setPlayer2Demand(uint demand) public onlyRole(PLAYER_ROLE) {
93          require(msg.sender == player2, "Only Player 2 can set his demand");
94          require(demand <= poolOfMoney, "Player 2 demand exceeds pool of money");
95          require(demand >= 0, "Demand has to be positive int");
96          require(player1ProposedAnOffer, "Player 1 didn't propose an offer yet");
97
98          player2Demand = demand;
99          player2ProposedAnOffer = true;
100     }
101
102     function p2acceptOfferOfPlayer1() public onlyRole(PLAYER_ROLE) {
103         require(player1ProposedAnOffer, "Player 1 didn't propose an offer yet");
104         require(msg.sender == player2, "Only Player 2 can accept");
105
106         player2Accepted = true;
107     }
108
109     function p1acceptOfferOfPlayer2() public onlyRole(PLAYER_ROLE) {
110         require(player2ProposedAnOffer, "Player 2 didn't propose an offer yet");
111         require(msg.sender == player1, "Only Player 1 can accept");
112
113         player1Accepted = true;
114     }
115
```

```
116     function p1rejects() public onlyRole(PLAYER_ROLE) {
117         require(player2ProposedAnOffer, "Player 2 didn't propose an offer yet");
118         require(msg.sender == player1, "Only Player 1 can accept");
119 
120         player1Rejected = true;
121     }
122 
123     function gameResult() public onlyRole(PLAYER_ROLE)
                                    returns (string memory, uint, uint){
124         require(player1Accepted || player2Accepted || player1Rejected,
                                    "At least one player has to accept");
125 
126         if (player1Rejected) {
127             return ("No agreement was made. Players get:", 0, 0);
128         } else if (player1Accepted) {
129             getRewardReputationTokens();
130             uint poolSplit;
131             poolSplit = poolOfMoney - player2Demand;
132             return ("Players found an mutually benefitial agreement. Pool split:",
                                                            poolSplit, player2Demand);
133         } else {
134             getRewardReputationTokens();
135             uint poolSplit;
136             poolSplit = poolOfMoney - player1Demand;
137             return ("Players found an mutually benefitial agreement. Pool split:",
                                                            player1Demand, poolSplit);
138         }
139     }
140 
141     //  ========== Other Functions ==========
142     // The following functions are overrides required by Solidity
143     // Helping the interoperability of ERC721 standard
144     function supportsInterface(bytes4 interfaceId)
145         public view virtual override(AccessControl, Reputation_NTT_Factory)
                                                            returns (bool) {
146         return super.supportsInterface(interfaceId);
147     }
148 }

101010 // All of the 1s and 0s are going where they are supposed to be :)
```

5_LicenceNTT.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/5_LicenceNTT.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.9;
03
04 // @openzeppelin libraries
05 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
06 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
07 import "@openzeppelin/contracts/utils/Counters.sol";
08
09 contract HuntLicence_NTT_Factory is ERC721, ERC721Enumerable {
10
11     // ========== Lifecycle Methods ==========
12     address nttContractAddress;
13     constructor() ERC721("Licence NonTrassferableToken", "HNT_NTT") {
14         // Start Token counter
15         _tokenIdCounter.increment();
16         // Store contract address
17         nttContractAddress = address(this);
18     }
19
20     // Override transfer function to make the token Non-Transferable
21     function _beforeTokenTransfer(
22         address from,
23         address to,
24         uint256 tokenId,
25         uint256 batchSize)
26         internal override(ERC721, ERC721Enumerable) {
27         require(from == address(0) || to == address(0),
28                 "ERROR: You can't send NonTransferableTokens :P");
29         super._beforeTokenTransfer(from, to, tokenId, batchSize);
30     }
31
32     // ========== NonTransferable Object ==========
33     // struct type that will store the state of the Licence NTT Object
34     struct licenceObject {
35         // Object Attributes
36         address hunterAddr;
37         uint licenceID;
38         bool isMinted;
39         // Object Methods
40         bool joinHuntFlag;
41     }
42
43     // mapping of the objects
44     mapping(uint => licenceObject) mapOfLicObjects;
```

```
46      // set Object Function flag to True to access Hunting Licence NTT
47      function setFunctionFlagToTrue(uint _tokenId) public {
48          licenceObject storage _licenceObject = mapOfLicObjects[_tokenId];
49          _licenceObject.joinHuntFlag = true;
50      }
51
52      // ========== Property variables ==========
53      // using counter to give each token an ID
54      using Counters for Counters.Counter;
55      Counters.Counter private _tokenIdCounter;
56
57      // ========== Minting Functions ==========
58      function mintLicenceNTT(address hunter) external {
59          uint256 tokenId = _tokenIdCounter.current();
60          _tokenIdCounter.increment();
61          _safeMint(hunter, tokenId);
62
63          // link the token to the struct Object
64          licenceObject storage _licenceObject = mapOfLicObjects[tokenId];
65          _licenceObject.hunterAddr = hunter;
66          _licenceObject.licenceID = tokenId;
67          _licenceObject.isMinted = true;
68          _licenceObject.joinHuntFlag = false;
70      }
71
72      //  ========== Getter Functions ==========
73      function getObject(uint _tokenId) view public
                                              returns (address, uint, bool, bool) {
74          licenceObject memory _licenceObject = mapOfLicObjects[_tokenId];
75          return (_licenceObject.hunterAddr,
76                  _licenceObject.licenceID,
77                  _licenceObject.isMinted,
78                  _licenceObject.joinHuntFlag);
79      }
81      function getMyBalance() public view returns (uint256) {
82          return balanceOf(msg.sender);
83      }
84
85      //  ========== Other Functions ==========
86      // The following functions are overrides required by Solidity
87      // Helping the interoperability of ERC721 standard
88      function supportsInterface(bytes4 interfaceId)
89          public view virtual override(ERC721, ERC721Enumerable) returns (bool) {
90          return super.supportsInterface(interfaceId);
91      }
92  }
```

6_StagHunt.sol

https://github.com/juju515/NTTs-as-Objects-in-a-Game-Theory-simulation/blob/master/src/6_StagHunt.sol

```
01 // SPDX-License-Identifier: MIT
02 pragma solidity ^0.8.0;
03
04 // import @openzeppelin libraries
05 // inheriting the LicenceNTT minting contract
06 import "@openzeppelin/contracts/access/AccessControl.sol";
07 import "./5_LicenceNTT.sol";
08
09 contract StagHuntGame is AccessControl, HuntLicence_NTT_Factory {
10
11     // ========== Variables =============
12     bytes32 constant HUNTER_ROLE = keccak256("HUNTER_ROLE");
13
14     address public player1;
15     address public player2;
16
17     uint player1Choice;
18     uint player2Choice;
19     bool public player1MadeChoice;
20     bool public player2MadeChoice;
21
22     // Function to set Hunter roles for the Stag Hunt game
23     // Require Hunting Licence NTT Token to get access to Hunter role
24     function setPayerRolesHunt(uint _p1LicenceNum, uint _p2LicenceNum) public {
25         licenceObject memory _licenceObject1 = mapOfLicObjects[_p1LicenceNum];
26         licenceObject memory _licenceObject2 = mapOfLicObjects[_p2LicenceNum];
27
28         require(_licenceObject1.joinHuntFlag, "Player 1 has to have a licence");
29         require(_licenceObject2.joinHuntFlag, "Player 2 has to have a licence");
30         require(_p1LicenceNum != _p2LicenceNum,
                                        "Players cannot have the same address");
31
32         _setupRole(HUNTER_ROLE, ownerOf(_p1LicenceNum));
33         _setupRole(HUNTER_ROLE, ownerOf(_p2LicenceNum));
34
35         player1 = ownerOf(_p1LicenceNum);
36         player2 = ownerOf(_p2LicenceNum);
37     }
38
39     // ========== Game Logic ==========
40     function getPlayer1Choice() public view returns (uint) {
41         return player1Choice;
42     }
43
```

```
44      function getPlayer2Choice() public view returns (uint) {
45          return player2Choice;
46      }
47
48      function setPlayer1Choice(uint choice) public onlyRole(HUNTER_ROLE) {
49          require(msg.sender == player1, "Only Player 1 can make a choice");
50          require(choice == 0 || choice == 1, "Invalid choice");
51
52          player1Choice = choice;
53          player1MadeChoice = true;
54      }
55
56      function setPlayer2Choice(uint choice) public onlyRole(HUNTER_ROLE) {
57          require(msg.sender == player2, "Only Player 2 can make a choice");
58          require(choice == 0 || choice == 1, "Invalid choice");
59          require(player1MadeChoice, "Player 1 has not made a choice yet");
60
61          player2Choice = choice;
62          player2MadeChoice = true;
63      }
64
65      function gameResult() public onlyRole(HUNTER_ROLE) view
                                              returns (string memory, uint, uint){
66          require(player1MadeChoice && player2MadeChoice,
                                              "Both players must make a choice");
67
68          if (player1Choice == 0 && player2Choice == 0) {
69              return ("Both players chose to hunt stag.
                                              They both receive a payoff of:", 4, 4);
70          } else if (player1Choice == 1 && player2Choice == 1) {
71              return ("Both players chose to hunt hare.
                                              They both receive a payoff of:", 1, 1);
72          } else if (player1Choice == 0 && player2Choice == 1) {
73              return ("Player 1 hunted stag, but Player 2 hunted hare.
                                              Players receive a payoff of:", 0, 2);
74          } else {
75              return ("Player 1 hunted hare, but Player 2 hunted stag.
                                              Players receive a payoff of:", 2, 0);
76          }
77      }
78
```

```
79      //  ========== Other Functions ==========
80      // The following functions are overrides required by Solidity
81      // Helping the interoperability of ERC721 standard
82      function supportsInterface(bytes4 interfaceId)
83          public view virtual override(AccessControl, HuntLicence_NTT_Factory)
                                                                  returns (bool) {
84          return super.supportsInterface(interfaceId);
85      }
86  }
```