Smart Auction
Dutch and Vickrey

Andrea Bruno

Department of Computer Science
University of Pisa
`a.bruno25@studenti.unipi.it`

1 Introduction

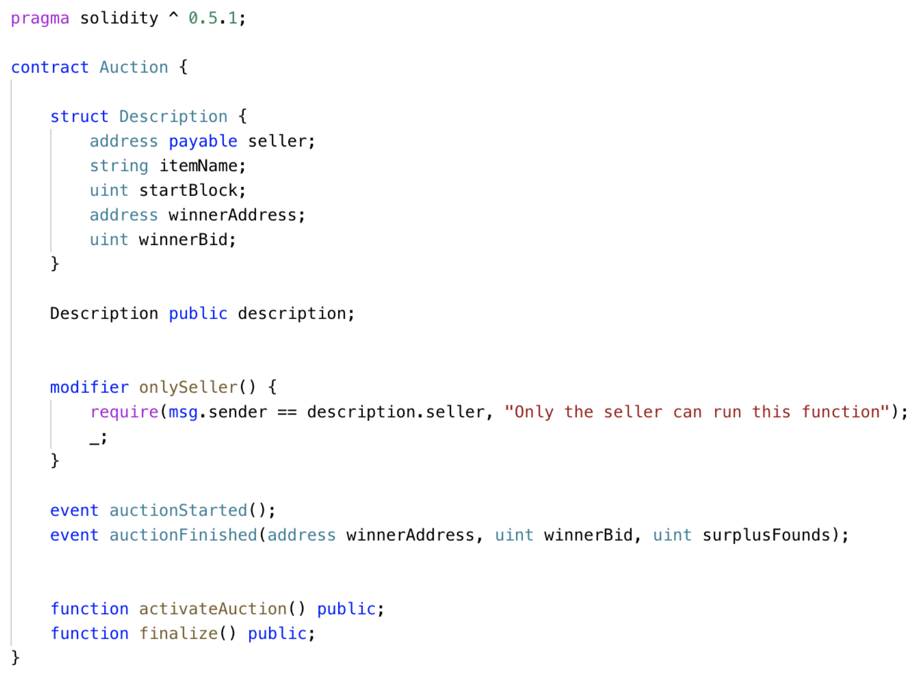
We were asked to implement two types of auction systems using the Ethereum blockchain. The first auction system is called Vickery whereas the second one (chosen by me) is called Dutch. Both systems share a so called "grace period" that is a time window of 5 minutes, in which the auction exists but is not active. In this implementation of the system, the common features of the two auctions are placed into an abstract contract called `Auction.sol`, whereas the specifics operations are left to the "sub-contracts" `DutchAuction.sol` and `VickreyAuction.sol`.

2 Auction.sol

To define a skeleton of shared functionalities, I chose to exploit the *Template-Method pattern*. This pattern requires a base class defining the shared code and some subclasses who will implement specific logics.

As you can see from Figure 1, in the Auction contract there is a public `struct` called *Description* that simulate the price-tag of the item that will be sold. This way, the basic info will be placed on the blockchain, hence people can watch the item without consuming gas, like a real shop.

As far as the "grace period" concerned, this functionality has been described within the `activateAuction` method, but since it is an abstract method, the implementation is delegated to the subclasses who will implement the Auction class. There is also a `finalize` method that allow the contract to be closed and confirmed adequately. Another fundamental part is the `onlySeller` modifier. This latter, when declared alongside a function, guarantee us that only authorized people (in this case the seller) can run that function. Moreover to communicate the begin and the end of the auction, two events have been declared.



```
pragma solidity ^ 0.5.1;

contract Auction {

    struct Description {
        address payable seller;
        string itemName;
        uint startBlock;
        address winnerAddress;
        uint winnerBid;
    }

    Description public description;


    modifier onlySeller() {
        require(msg.sender == description.seller, "Only the seller can run this function");
        _;
    }

    event auctionStarted();
    event auctionFinished(address winnerAddress, uint winnerBid, uint surplusFounds);


    function activateAuction() public;
    function finalize() public;
}
```

Fig. 1. Source code of *Auction.sol*

3 DutchAuction.sol

As specified in the requirements, the Dutch auction consists of an initial phase in which the price of the good is set to a very high price and afterwards, is gradually lowered by following a precise strategy. The auction finishes when someone is willing to pay the price of the good.

In this implementation of the Dutch auction, the contract extends the abstract contract `Auction` in order to follow the *Template-Method pattern* correctly and additionally, it uses a contract of class `Strategy` to implement the logic of devaluation.

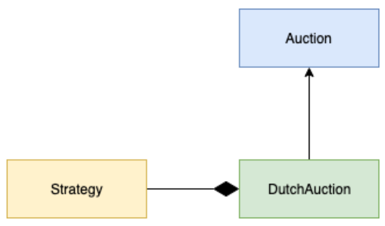


Fig. 2. UML view of *DutchAuction*

3.1 Strategy.sol [1]

We were asked to develop multiple method to compute the decrease of the price. In this work, this requirement has been developed exploiting an abstract contract (Figure 3) containing only one abstract method called `getPrice`.

```
contract Strategy {
    function getPrice(uint _actualPrice, uint _deltaBlocks) public view returns(uint);
}
```

Fig. 3. Source code of *Strategy* abstract class

Afterwards, we create three sub-contracts implementing the `Strategy` contract, in order to let the user choice his own favorite strategy. In particular, `NormalStrategy` decreases in a linear way, then `SlowStrategy` decreases twice slower than `NormalStrategy` and finally, `FastStrategy` decreases twice faster than `NormalStrategy`.

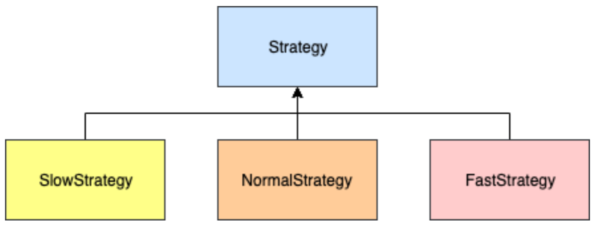


Fig. 4. UML view of the multiple *Strategy*

[1] Since every Dutch auction requires a strategy, we first describe the Strategy contract and afterwards the DutchAuction implementation.

3.2 Status

Although in the Dutch auction there are only three explicit phases, for security reasons we added a new phase called *Validation*. Since there could be the case in which two people make simultaneously a bid, as Figure 5 shows, before closing the contract, we verify that at least 12 blocks [1] have been confirmed. This restriction is essential to ensure that all miners own the same status of the contract and thus the same winner.

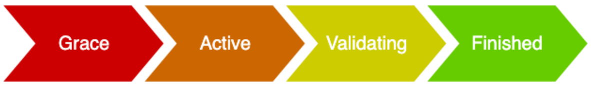


Fig. 5. Status of the *Dutch* auction

3.3 Constructor

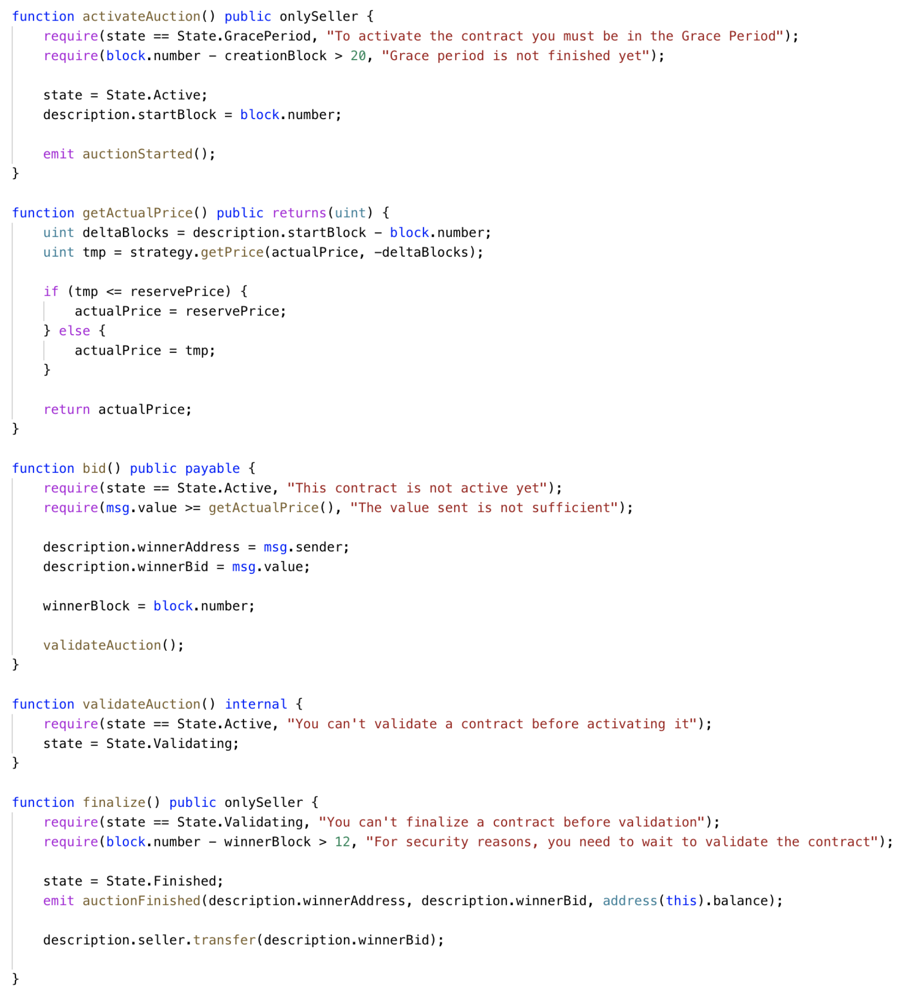
To create a new Dutch auction you need to insert the name of the item that you are willing to sold, both a reserve price and an initial price and lastly the address of a Strategy contract.

```
constructor(
    string memory _itemName,
    uint _reservePrice,
    uint _initialPrice,
    Strategy _strategy
) public {
    description.seller = msg.sender;
    description.itemName = _itemName;
    state = State.GracePeriod;

    reservePrice = _reservePrice;
    initialPrice = _initialPrice;
    actualPrice = _initialPrice;

    strategy = _strategy;

    creationBlock = block.number;
}
```

Fig. 6. Source code of the *DutchAuction* constructor

Once deployed the contract correctly, its own status is set to *Grace Period* . Note that in this implementation, the role of auctioneer does not exist and all critical functionalities are left to the seller of the item.

3.4 Functions



```
function activateAuction() public onlySeller {
    require(state == State.GracePeriod, "To activate the contract you must be in the Grace Period");
    require(block.number - creationBlock > 20, "Grace period is not finished yet");

    state = State.Active;
    description.startBlock = block.number;

    emit auctionStarted();
}

function getActualPrice() public returns(uint) {
    uint deltaBlocks = description.startBlock - block.number;
    uint tmp = strategy.getPrice(actualPrice, -deltaBlocks);

    if (tmp <= reservePrice) {
        actualPrice = reservePrice;
    } else {
        actualPrice = tmp;
    }

    return actualPrice;
}

function bid() public payable {
    require(state == State.Active, "This contract is not active yet");
    require(msg.value >= getActualPrice(), "The value sent is not sufficient");

    description.winnerAddress = msg.sender;
    description.winnerBid = msg.value;

    winnerBlock = block.number;

    validateAuction();
}

function validateAuction() internal {
    require(state == State.Active, "You can't validate a contract before activating it");
    state = State.Validating;
}

function finalize() public onlySeller {
    require(state == State.Validating, "You can't finalize a contract before validation");
    require(block.number - winnerBlock > 12, "For security reasons, you need to wait to validate the contract");

    state = State.Finished;
    emit auctionFinished(description.winnerAddress, description.winnerBid, address(this).balance);

    description.seller.transfer(description.winnerBid);

}
```

Fig. 7. Source code of the *DutchAuction* functions

To activate the auction, the *Grace Period* of 5 minutes should finish. According to `etherscan`[2] every 15 seconds a new block is confirmed, hence:

$$5\,min = 300\,sec \implies \frac{300\,sec}{15\,sec/block} = 20\,blocks$$

By waiting 20 blocks we can (more or less) ensure the 5 minutes asked by the requirements.

Another interesting aspect is the `getActualPrice` function. Since the price drops over time, it must be updated, thus a public `uint` value on the blockchain is not sufficient. For this reason, people interested in knowing the price, should pay some gas to update the price.

Now let's analyze the `bid` function. At the beginning of the function, we require that the `msg.value` (in other words, the money sent) are greater or equal the `actualPrice`. Right after this control, it is immediately written on the blockchain both the address of the user that made the bid, and the value of ether that he sent to the contract. Afterwards, due to security reasons related to simultaneous bids (as previously discussed in 3.2), the contract passes in a *Validating* status. From now on, the seller can finalize the contract and receive the ether he's owed.

3.5 Gas Evaluation

Function	**Transaction Cost**	**Execution Cost**	**Caller**
constructor	1067764	787072	Seller
`activateAuction()`	63359	42087	Seller
`getActualPrice()`	29677	8405	All
`bid()`	95839	74567	All
`finalize()`	37983	16711	Seller
Total	1294622	928842	

Table 1. Gas consumption of *DutchAuction* contract

4 VickreyAuction.sol

This kind of auction involves an initial phase in which all participants submit their bid in a sealed envelope. Afterwards, all these envelopes are opened and the person who did the highest bid, wins the auction by paying the second highest price. Due to possible fraud, people willing to submit their bid, must sent a deposit. This measure should discourage people in such unfair practices.

Similarly to the `DutchAuction`, the `VickreyAuction` as well extends the “super-contract” `Auction`, hence it inherits all the fields of its “father”. Furthermore, in this way, the *Template-Method pattern* is respected.

4.1 Phases

The `VickreyAuction` starts with a *Grace Period* of 5 minutes in the same way of the `DutchAuction`. The second phase called *Commitment*, is the period in which people can submit their bids. The novelty is the *Withdrawal* phase. Basically, during this phase the bidders can withdrawn their bids and get only half of their deposit back. Then we have the *Opening* phase, that is the period in which all the bids are opened, and lastly the closing of the contract.

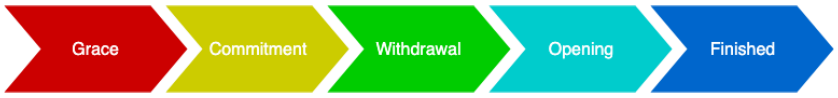


Fig. 8. Phases of the *Vickrey* auction

4.2 Constructor

To create a Vickrey auction, few parameters are needed. In particular, the lengths of all the phases are required, as well as both the reserve price and the minimum deposit. Note from Figure 9 that in this implementation, the minimum deposit must be greater or equal to the reserve price.

```
constructor(
    string memory _itemName,
    uint _reservePrice,
    uint _min_deposit,
    uint _commitment_len,
    uint _withdrawal_len,
    uint _opening_len
) public {
    require(_reservePrice > 0, "Reserve price should be greater than zero.");
    require(_min_deposit >= _reservePrice, "The deposit should be greater than the reserve price");
    require(_commitment_len > 0, "The lenght of commitment should be greater than zero.");
    require(_withdrawal_len > 0, "The lenght of withdrawal should be greater than zero.");
    require(_opening_len > 0, "The lenght of opening should be greater than zero.");
```

Fig. 9. Source code of *VickreyAuction* constructor

4.3 GenerateBid.sol

To help people in making bids, we create a contract called `GenerateBid`. As you can see from Figure 10, there is a function called `generateBid`, that given an integer value in input (the desired bid), it returns a 32-bit nonce obtained from the timestamp, and also the hash of the tuple $< bid , nonce >$. Of course, it will be the responsibility of the person who did the bid, not to reveal neither his bid nor his nonce.

```
pragma solidity ^ 0.5.1;

contract GenerateBid {

    struct BidHelper {
        uint value;
        bytes32 nonce;
        bytes32 hash;
    }

    BidHelper public bid;

    function generateBid(uint _bidValue) public {
        bid.value = _bidValue;
        bid.nonce = keccak256(abi.encodePacked(block.timestamp));
        bid.hash = keccak256(abi.encodePacked(_bidValue, bid.nonce));
    }
}
```

Fig. 10. Source code of *GenerateBid* contract

4.4 Modifiers

Before going deeper in the function implementations, we need first to understand the modifiers declared inside the contract. This tool (the modifier), allow to execute some code before and after a function. In particular, the code of the function will be executed when the `_;` symbols will occur.

By exploiting the potentialities of modifiers, we defined a way to control whether the actual phase is the one desired.

```
modifier duringCommitment {
    require(phase == Phase.Commitment, "Commitment phase not started yet");
    require((block.number - startPhaseBlock) <= commitment_len, "Commitment phase is ended");
    _;
}

modifier duringWithdrawal {
    require(phase == Phase.Withdrawal, "Withdrawal phase not started yet");
    require((block.number - startPhaseBlock) <= withdrawal_len, "Withdrawal phase is ended");
    _;
}

modifier duringOpening {
    require(phase == Phase.Opening, "Opening phase not started yet");
    require((block.number - startPhaseBlock) <= opening_len, "Opening phase is ended");
    _;
}
```

Fig. 11. Source code of *VickreyAuction* modifiers

4.5 Commitment

As already explained in Section 4.3 every `bid` is composed by three parameters:

1. the `uint` value of the bid;
2. a `bytes32` nonce;
3. the hash of $< bid\,, nonce >$.

Now, since the bids must be secret until the opening phase, the contract should not write any information about these bids on the blockchain. To make it possible, the function requires and stores only the hash. In this way, only during the opening phase, we require both the value and the nonce, in order to verify the hash and thus the correctness of the bid.

Of course, to avoid misbehavior, people have to submit a deposit greater or equal the minimum deposit decided at deploy time by the seller.

Note that all these bids are stored in a non-public `mapping(address=>Bid)` called *bids*.

```
function bid(bytes32 _bidHash) public duringCommitment payable {
    require(msg.value >= min_deposit, "The value sent is not sufficient");
    require(bids[msg.sender].value == 0, "You have already submitted a bid");

    Bid memory tmp_bid;
    tmp_bid.hash = _bidHash;
    tmp_bid.deposit = msg.value;

    bids[msg.sender] = tmp_bid;
}
```

Fig. 12. Source code of *bid* function

4.6 Withdrawal

Another interesting aspect of the Vickrey auction, is the *Withdrawal* phase. During this period of time, people can retire their bids, but only half of the deposit will be reimbursed.

As Wohrer et al. [3] suggest, when a contract calls another contract, it hands over control to that other contract. The called contract can then, in turn, re-enter the contract by which it was called and try to manipulate its state or hijack the control flow through malicious code. A solution is the *Checks-Effects-Interaction pattern*. Basically, all the calls to external contracts, are always placed at the end of the function. This way, the possible attack surface is limited.

This pattern is divided in three phases: first, check all the preconditions, then make changes to the contracts state, and finally interact with other contracts.

```
function withdrawal() public duringWithdrawal {
    //1. Checks
    require(bids[msg.sender].deposit > 0, "You don't have any deposit to withdraw");

    uint bidderRefund = bids[msg.sender].deposit / 2;
    uint sellerRefund = bids[msg.sender].deposit - bidderRefund;

    //2. Effects
    bids[msg.sender].deposit = 0;
    emit withdrawalExecuted(msg.sender, bidderRefund, description.seller, sellerRefund);

    //3. Interaction
    description.seller.transfer(sellerRefund);
    msg.sender.transfer(bidderRefund);
}
```

Fig. 13. Source code of *withdrawal* function

4.7 Opening

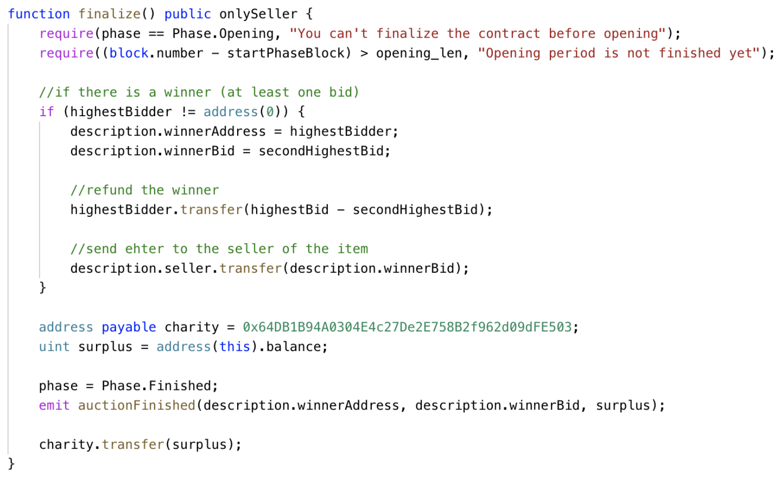
As written in the requirements, in the opening phase all bidders should reveal their bid by sending the nonce used previously. At the top of Figure 14, we can see that the only parameter required by the `open` function is a `bytes32` nonce. If bidders used our generator (see 4.3), this nonce value can be easily pasted and sent to the contract. The first control that is done, is the correctness of the hash previously sent, by checking the hash of the tuple $< bid, nonce >$. If this constraint is respected, then the bidder can be refunded. Afterwards, we check whether the current bid is the first or not, and then by looking at the value sent, we declare the winner and the highest second bid. Note that in case of overwriting of the winner, there is a refund the old highest bidder.

```
function open(bytes32 _nonce) public duringOpening payable {
    //control the correctness of the bid
    require(keccak256(abi.encodePacked(msg.value, _nonce)) == bids[msg.sender].hash, "Wrong hash");

    //refund the deposit
    uint deposit = bids[msg.sender].deposit;
    bids[msg.sender].deposit = 0;
    msg.sender.transfer(deposit);

    //update the bid status
    bids[msg.sender].value = msg.value;
    bids[msg.sender].nonce = _nonce;

    //if it is the first opening
    if (firstOpen) {
        highestBidder = msg.sender;
        highestBid = msg.value;

        //if there is only one bid, the winner have to pay at least the reservePrice
        secondHighestBid = reservePrice;

        firstOpen = false;
    } else {
        //if the msg.value is more than the highest bid
        if (msg.value > highestBid) {
            //the highest bid becomes the second highest bid
            secondHighestBid = highestBid;

            //now we need to refund the bidder of the (old) highest bid
            refund(highestBidder, highestBid);

            //set the new highest bidder and its own bid
            highestBidder = msg.sender;
            highestBid = msg.value;

        } else {
            //check whether the msg.value is higher than the second highest bid
            if (msg.value > secondHighestBid) secondHighestBid = msg.value;

            //since the current opening is not the highest we can refund the sender
            refund(msg.sender, msg.value);
        }
    }
}
```

Fig. 14. Source code of *open* function

4.8 Finalize

This function is the last step before closing definitely the contract. If there is at least a bid, or in other words, if there is a winner, we refund him with the difference between his bid and the second highest. Afterwards we close the auction, we send money to the seller and if there is a surplus (for instance someone who sent a bid, but forgot to opening it), we transfer this money to a charity address.



```
function finalize() public onlySeller {
    require(phase == Phase.Opening, "You can't finalize the contract before opening");
    require((block.number - startPhaseBlock) > opening_len, "Opening period is not finished yet");

    //if there is a winner (at least one bid)
    if (highestBidder != address(0)) {
        description.winnerAddress = highestBidder;
        description.winnerBid = secondHighestBid;

        //refund the winner
        highestBidder.transfer(highestBid - secondHighestBid);

        //send ehter to the seller of the item
        description.seller.transfer(description.winnerBid);
    }

    address payable charity = 0x64DB1B94A0304E4c27De2E758B2f962d09dFE503;
    uint surplus = address(this).balance;

    phase = Phase.Finished;
    emit auctionFinished(description.winnerAddress, description.winnerBid, surplus);

    charity.transfer(surplus);
}
```

Fig. 15. Source code of *finalize* function

4.9 Gas Evaluation

Function	**Transaction Cost**	**Execution Cost**	**Caller**
constructor	2226512	1664784	Seller
`activateAuction()`	68350	47078	Seller
`bid()`	65790	42342	All
`startWithdrawal()`	33619	12347	Seller
`withdrawal()`	30811	24539	All
`startOpening()`	33685	12413	Seller
`open()`	114170	120722	All
`finalize()`	119843	98571	Seller
Total	2578610	2022796	

Table 2. Gas consumption of *VickreyAuction* contract

References

1. Ethereum Blog, `https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times`
2. Block Explorer and Analytics Platform for Ethereum, `https://etherscan.io/chart/blocktime`
3. Wohrer, Maximilian, and Uwe Zdun. "Smart contracts: security patterns in the ethereum ecosystem and solidity." 2018 *International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018.