Smart Auctions

Andrea Bongiorno

June 2019

Contents

1 Introduction

The final term of the *Peer to Peer and Blockchain* course requires to develop two different Solidity's smart contracts implementing different kind of auctions. For this project I have implemented the *English Auction* and the *Vickrey Auction*.

I decided also to implement a little escrow mechanism with the aim to highlight the problem of trust in the e-commerce world, it is described in Section 2.1. In Section 2.2.1, 2.2.2 respectively are described the two auction's implementations.

Since an e-commerce system is quite a critical one, I used some well defined programming pattern to achieve an high degree of security, they are illustrated in Section 3.

In the end, the Section 4 estimates the gas consumption of typical interactions between users and contracts developed.

2 Contracts

2.1 Escrow

Since this project aims to model online auctions and given the technology used, there is no trust between the main actors involved: seller and buyers. In order to fill this trust gap has been implemented a little escrow contract, contained into the file Escrow.sol, with the purpose of protecting both seller and buyer but keeping in mind that when a bid is made, the buyer immediately sends its money.

Using the following real world analogy, here is the rationale behind this escrow contract. The actors involved are: the seller who has the responsibility to send the good, the buyer, the auctioneer who act as a referee, and a courier chosen by the seller. Here is a step-by-step scenario:

- The winning buyer, after the conclusion of the auction, sends to the auctioneer, the SHA3 of a chosen nonce.
- The seller cannot deliver the good directly to the buyer, so he will use a third party courier.
- When the courier arrives at buyer's place, he will deliver the good only if the buyer reveals the correct nonce.
- The seller receives the nonce from the courier so he can prove to the auctioneer the correct deliver of the good and then get paid.
- Now, two scenarios are open:
    1. The buyer refuse to reveal the nonce or collude with the courier: in this case if the seller can prove to the auctioneer that he has sent the good (e.g. with an expedition number) then he get paid. Note that

this scenario is unlikely because the buyer has already paid when making the winning bid.

2. The seller doesn't send the good: in this case he won't be able to prove the expedition and the buyer will be refunded.

Obviously the escrow mechanism illustrated above is more a proof of concept than a practical solution, but I decided to implement it anyway in order to highlight the problem of trust with e-commerce.

2.2 Auction

Into the file Auction.sol is defined a *super-contract* that encapsulates some common features between the Vickrey auction and the English Auction. This contract contains:

- Events, used by the sub-contracts to notify the occurrence of some facts such new offers, auctions' state, withdrawals and refunds.
- Common state variable, such as seller and buyer addresses, reserve price of the good, initial and final block of the auction.
- Debug variables, used to bypass modifier when debugging.

Both the auctions implemented are modelled as a state machine, following the corresponding behavioural pattern described in Section 3.1.

2.2.1 Vickrey Auction

For this auction, in the real-world, bidders commit their offers in a sealed envelope, they have the possibility to withdraw their offer before the opening phase. In the proposed implementation, has been chosen to model the bids from the bidders using a struct composed by the following field:

- *bidHash*: a bytes32 that represents the sealed envelope.
- *value*: the actual value of the bid, this field will be filled once the envelope is opened.
- *opened, withdrawn, refunded*: three booleans that describe the state of the bid.

The bids are collected into a mapping using as key the address of the corresponding bidder.

This implementation propose a state machine view of the Vickrey auction. The several states are described by the enum *Phases* and they are the following:

1. Commitment
2. Withdrawal

3. Opening
4. Finished

with the obvious semantics.

The phases listed above follow each other temporally (without overlapping) and the flow of time has been implemented using the modifier *blockTimedTransition*: it checks if the actual block number allows a state transition, according to the parameter provided at deploy time. For sake of completeness, the function is reported in Listing 1. The modifier *duringPhase* checks if a certain function it's called in the correct phase. These modifiers are part of the *State Machine pattern*.

```
modifier blockTimedTransition() {
    if(phase == Phases.Commitment && block.number > end_commitment)
        nextPhase();
    if(phase == Phases.Withdrawal && block.number > end_withdrawal)
        nextPhase();
    if(phase == Phases.Opening && block.number > end)
        nextPhase();
    _;
}
```

Listing 1: Modifier used for modelling the time flow during a Vickrey auction. The function *nextPhase()* realise the transition from the actual phase to the next one, following the order described above and emitting an event.

The three main phases of the Vickrey auction are controlled mainly by three functions:

1. *commit*: this function takes a bytes32 as parameter, it is the *keccak-256* of a nonce concatenated to the value of the bid. Everyone can commit a bid, but it is allowed only one bid for bidder. When invoking this function, the bidder must send a deposit. The deposit requirement is set at contract construction by the auctioneer and it must be between a half and a quarter of the reserve price. On success this function emit a *LogEnvelopeCommitted* event, with the address of the bidder as parameter.

2. *withdrawEnvelope*: using this function a bidder that has previously committed a bid and has not withdrawn it yet, can withdraw it. In this case the bidder will immediately receive a refund equal to half of the deposit requirement. Since the bidder receiving the refund is the one who call the function, an immediate refund does not violate the *Withdrawal pattern* described in 3.2. The state of the bid is updated, setting to *true* the corresponding field.

3. *open*: this function can be invoked only once from a bidder that has not withdrawn his envelope. The bidder sends as parameter the nonce used to calculate the *bidHash* along with its actual bid. These values are used to calculate a *keccak-256* to be compared with the one previously sent. On failure the transaction is reverted, on success there are the following possibilities:

(a) The bid is less than the reserve price: this case is treated as cheating because the bidder made an offer knowing that it cannot be the winning one. In this case the bidder will receive a refund of his bid and a half of the deposit requirement. A *LogVoidBid* with information about the bidder and the bid is emitted.

(b) The bid is the first one: in this case the price to pay (second highest bid) is set to reserve price and information about the bidder and the bid are recorded into the variables *highestBid* and *highestBidder*. A *LogHighestBid* event is emitted.

(c) The bid is lower than the highest bid: the value of the bid is checked against the second highest bid and if it is greater then the price to pay is updated and also a *LogUpdateSecondPrice* is emitted. Anyway, the bidder will receive a full refund (deposit + bid value) and a *LogLosingBid* is emitted. Again, the *Withdrawal pattern* is not violated and the status of the bid is updated to *refunded*.

(d) The bid is the new highest one: when this situation occurs the price to pay is updated to the old highest bid and the new information about the actual highest bid and highest bidder are also updated. Is not possible to refund immediately the previous highest bidder because in this case the *Withdrawal pattern* will be violated, as explained deeply in 3.2.

When the opening phase is ended, and the contract is in the *Finished* state, there are two things left: finalize the auction and refund bidders that weren't refunded during the opening phase. The refund is performed using the function *getRefund* that can be called only once by those bidders that opened their envelope and has not received a refund yet (check point d of the list above). This function was written just to implement the *Withdrawal pattern* described in 3.2, if it is called by the highest bidder than he will receive as refund the entire deposit plus the difference between his bid and the second highest bid, otherwise the bidder who call this function will receive as refund the deposit plus his bid.

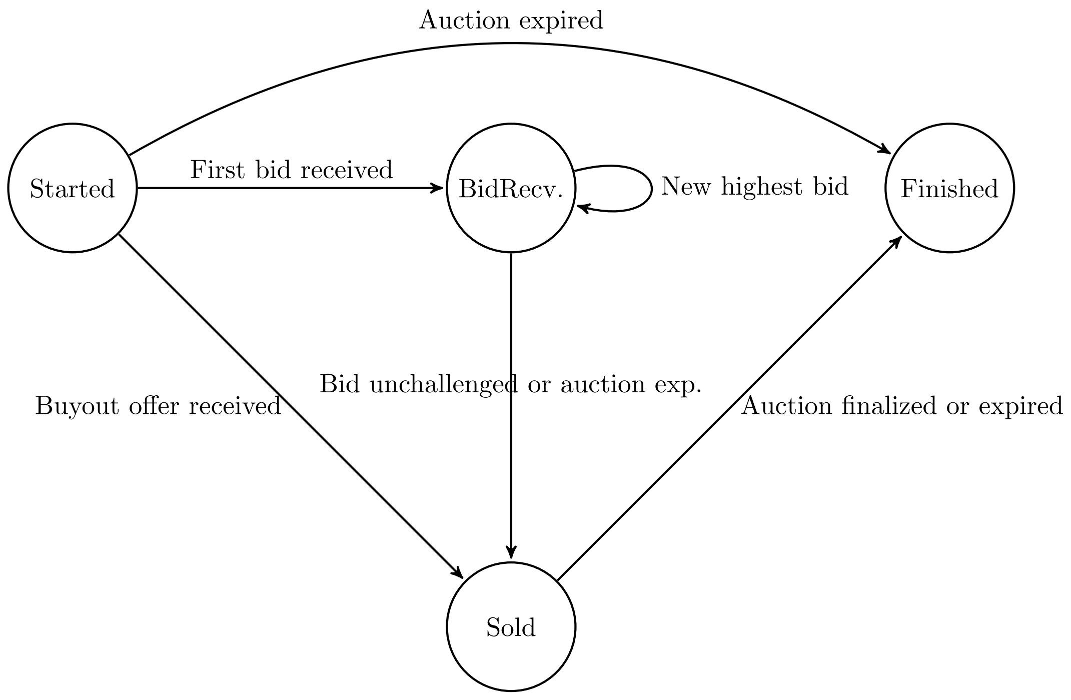
For what concern the finalization of the auction there is a function *finalize* that can be called only once by the auctioneer. This function does two things: first it deploy an escrow contract and second transfer the funds of bad bidders to a charity address specified at the construction of the auction's contract. If this method is called passing *true* as parameter, a *LogEscrow* event is emitted, which contains the address of the escrow contract. If there aren't valid bid, the good remains unsold and a *LogUnsold* event is emitted.

2.2.2 English Auction

Also this auction has been modelled as a state machine but in a slightly different way with respect to the previous one. The possibile states are described by the struct *Phases* and they are the following:

1. Started,
2. BidReceived,
3. Sold,
4. Finished

For this auction the states does not follow each other temporally and the state transitions are described by the automaton below.



Once again the flow of time has been implemented using the modifier *block-TimedTransaction*, that is slightly different from the previous one, as shown by Listing 2.

```
modifier blockTimedTransition {
    if(phase == Phases.BidReceived && (block.number > end || block.
        number > bidBlock+unchallengedInterval)) {
        phase = Phases.Sold;
        emit LogSold(highestBidder, highestBid);
    }
    else if(phase == Phases.Started && block.number > end) {
        phase = Phases.Finished;
        emit LogPhaseTransition("Finished");
    }
    _;
}
```
Listing 2: Modifier used for modelling the time flow during an English auction, according to the automaton.

The main operations available for this kind of auction are the following:

- *buyNow*: it can be invoked only in the *Started* phase and its successful execution triggers a transition to the *Sold* state. When invoking this function, the buyer has to provide the exact amount for the buyout, established at contract deploy. This function cannot be invoked if a bid has been already received i.e. if the contract is in the *BidReceived* state, as required by the text of the assignment.

- *bid*: this is the function for bidding. A bid is valid if it is made in the *Started* phase or in *BidReceived* phase (when the *unchallenge period* is not expired yet). The bid must be greater than the previous one plus a minimum increment specified, in percentage, at contract deploy time and stored in the variable *minIncrement*. Obviously, in the case of the first bid of the auction, this function triggers a phase transition form *Started* to *BidReceived*.

When a bid is outbid, there is one bidder that has to be refunded: this cannot be done inside the *bid* function because this would violate the *Withdrawal pattern*. If a bidder wants to be refunded, he can call the *withdrawal* function and get a refund.

3 Security consideration

Since for this project there are theoretically some money involved, guarantee the security of the systems developed is not trivial. The main risk when programming in *Solidity*, excluding trivial bugs, is the re-entrancy attack: when a contract interacts with another one (e.g. when transferring funds) it has to consider the possibility that the called contract can re-call it and this can result in bad behaviour such double-spending of some kind of token or funds, or denial of service. In this section are described three widely used design patterns that if well implemented they guarantee an high level of protection from attacks and misbehaviour.

3.1 State Machine pattern

The State Machine pattern is a behavioural pattern, useful to model systems that can be in several state with well defined rules for the transitions between states. Contracts that fit in this scheme are often characterised by the fact that they expose certain functionality only when they are in a precise state.

This pattern when well implemented ensure that the contract exposes only the correct functionality and guarantee consistency between the different states.

Auctions are a typical scenario where this pattern is used since they are often divided into stages and at each stage the bidders are allowed to execute only certain functions. Stages in auctions are controlled by time: in this case the State Machine pattern ensures soundness in the time flow: this is the rationale behind the modifiers *blockTimedTransition* and *duringPhase*, note that they must follow this order.

Implementation details for this pattern are not standard although it is very common the use of enum and struct for modelling the state and of modifiers in order to determine function exposure. The details for the code developed for this project have been described in Section 2.2.2 and Section 2.2.1.

3.2 Withdrawal pattern

The Withdrawal pattern is a security pattern commonly used to send funds after an effect. Although it can be counterintuitive, this pattern is very powerful when the aim is to prevent a failure during a refund.

In order to understand its power let consider the following scenario. During an English auction, a dishonest bidder want to spend the least possible amount of ether to buy the selling good. To reach its goal the bidder make the first offer from an address associated with a contract. Next, another bidder makes his bid, higher than the one of the briber. Without using the withdrawal pattern the code executed will be something similar to Listing 3.

```
 1  function bid(...) ... {
 2      ...
 3      if(msg.value > highestBid) {
 4          /* refund the previous bidder*/
 5          highestBidder.transfer(highestBid);
 6          /*update info about winning bid*/
 7          highestBidder = msg.sender;
 8          highestBid = msg.value;
 9      }
10      ...
11  }
```

Listing 3: Buggy code that do not use the withdrawal pattern

The problem is in line 5: *highestBidder* contains the address of a contract owned by the briber and if this contract has a fallback function (called by *transfer*) that always fails, it will be not possible to record an offer that outbid the one of the briber and the contract is trapped in its state. When the auction ends, the briber win the good spending the minimum amount of ether.

To implement the withdrawal pattern it is sufficient to remember who has to be refunded, for example using a mapping. When a bidder wants to withdraw his funds, he can call a specific function that make the transfer. Since the function is called by the owner of the funds, an eventual failure will only prevent the owner to recover its funds. A correct example is shown in Listing 4, that differs from the previous only for line 5.

```
 1  function bid(...) ... {
 2      ...
```

```
    if(msg.value > highestBid) {
        /* refund the previous bidder*/
        pendingRefunds[highestBidder] += highestBid;
        /*update info about winning bid*/
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    ...
}
```

Listing 4: Correct code that do use the withdrawal pattern

In the code developed, this pattern has been implemented for the English auction as shown by the above example and for the Vickrey auction because of the situation when the opening of an envelope outbid a previous winning bid.

3.3 Checks-Effects-Interactions pattern

This security pattern has to be used when it cannot be avoided to hand over control flow to an external entity and it is desired protection against re-entrancy attacks. The C-E-I pattern is implemented in three steps:

1. Check the eligibility of the function invocation.
2. Adopt an *optimistic accounting* policy by updating the state of the contract before a possible interaction with an external contract.
3. Interact with the external contract.

Is possible to find a perfect example of the C-E-I pattern considering the withdrawal pattern described in Section 3.2.

Once implemented the withdrawal pattern we have the be sure that a bidder can ask its refund exactly once. The intuitive solution is to implement a *refund* function like the one in Listing 5.

```
1  function refund() public {
2      require(pendingRefunds[msg.sender] > 0)
3  
4      msg.sender.transfer(pendingRefunds[msg.sender]);
5  
6      pendingRefunds[msg.sender] = 0;
7  }
```

Listing 5: Buggy refund function, vulnerable to re-entrancy attacks

Now, the function *refund* respect the check-first property but the effects, i.e. zeroing the balance of *msg.sender*, are performed after the interaction with an external contract. In such a situation if the *transfer* in line 5 triggers a re-entrancy attack (through the fallback function) to the *refund* function, *msg.sender* will receive a double refund since its balance has not been updated yet.

The correct implementation is simple and requires just to swap line 4 with line 6. As a rule of thumb, when implementing this pattern, interactions (e.g.

transfer, send, call etc.) should be the last commands of a function. Obviously in some cases a fail-safe policy should be implemented: if instead of *transfer* I had used *send*, on failure I had to update again the balance of *msg.sender*.

For sake of completeness in Listing 6 is shown the correct code.

```
function refund() public {
    require(pending_refunds[msg.sender] > 0)

    uint refund = pending_refunds[msg.sender];
    pending_refunds[msg.sender] = 0;

    msg.sender.transfer(refund);
}
```

Listing 6: Correct refund function. Since on failure *transfer* revert the transaction there is no need to check its returned value.

4 Gas consumption consideration

In this section are provided some estimations of the gas needed for typical sequences of operations that a bidder does when interacting with the two contracts. In the considerations below will be used the following constants:

$$B = 21000, T = 2300.$$

The constant above are respectively the base cost of a transaction and the cost of a *transfer*.

Another useful constant for these estimations is

$$L = 1850$$

that is an estimation based on several execution of the cost of the primitive LOG1 with 3 arguments of 32 bytes each. A LOG1 is the main primitive used when emitting an event. The constant L has been chosen in order to consider the worst case: not all the event emitted have 3 parameters.

The last two constants are the following:

$$S = 20000, S' = 5000$$

they are respectively the cost of an SSTORE that update a value changing its *zeroness* and the cost of an SSTORE that does not change the *zeroness* of the value updated.

For the following calculations are assumed successful invocations of the involved functions and a good behaviour of the bidders. Given the assumptions above, in many case will be possible to leave out the cost of modifiers since they perform comparisons (operations LT, GT, ISZERO etc) and conditional jumps (operations JUMP and JUMPI) that have a lower cost with respect to other operations performed by the functions.

Note: experiments shown that the effective gas consumption can be higher than the estimations in certain situations because of the timed nature of the code: one function call could trigger a state transition, this in turn triggers event and function calls. Is not so fair that this gas is paid by the bidder, one solution can be to enclose this code between *msg.gas* calls and at the end calculate a refund for the caller.

4.1 Vickrey auction scenario

In the case of a Vickrey auction there are two main scenarios:

- *Commit and Withdrawal*: *commit* stores an hash into an entry of a mapping and emit an event. Storing the hash is a first-time write so its cost is S. In this scenario, when *commit* terminates, *withdrawEnvelope* is called by the bidder. This function does 3 operations: update the status of the caller's bid, refund the caller, emit an event. First main operation is again an SSTORE that costs S gas units like above because we are setting to true (a non-zero value) the *withdrawn* field of the bid that was false (a zero value). Second operation is the refund that costs T and third operation is the event with cost L. Finally this sequence of operations will cost to a bidder $\approx 2 \times (B + L + S) + T$.

- *Commit and Open*: the commit cost is the same of the above scenario $(B{+}S{+}L)$. The *open* function has some fixed costs: some gas is consumed for computing a *keccak* to check the validity of the bid (its cost is irrelevant and less than 50 gas, exactly 30 + 6 for each word, and we have 2 words so it's 42 gas) and $2 \times S$ not negligible gas units for updating the bid status (two SSTORE that change the zeroness of the previous values of the fields *opened* and *value*). *Open* can successfully terminate in three ways:

  1. The opened bid is less than reserve price: this cost S for updating the status of the bid into the mapping (the field *refund* is set to true), L for logging and T for a partial refund.
  2. The opened bid is the first valid one: this cost $S \times 3$ for updating information about the price to pay, the highest bid and the highest bidder. In the end, an event is emitted paying L gas units.
  3. The opened bid is a losing one: this may cost S' gas unit plus L if it is the new second highest (update *priceToPay* and log) and also others S for updating the bid status to refunded, T for the actual refund and again L for logging. In this case could be spent up to $S' + 2 \times L + T + S$ gas units.
  4. The opened bid is the new highest one: in this case the bidder pays $S' \times 3$ to update the variables that store information about the first and second highest bid (*priceToPay, highestBid and highestBidder*), that cannot be zero, plus L for logging.

Summing up, the bidders pay a fixed amount of about 40000 gas units that can go up to 140000 (case 2), depending on the situation.

According to *ethgasstation.info*, at the time of writing this report, commiting and opening a bid can cost up to 0.5 dollars, overestimating a gas consumption of 250000 gas units

4.2 English auction scenario

For the English auction the are two simple scenarios:

- Buy now: in this case the bidder decide to buy the good by buyout. This case is simple and completely managed by the function *buyNow* that costs $3 \times S + L$: it updates the phase of the bid, the information about highest bid and highest bidder and emit an event for logging purpose. Note: the contract is passing from phase *Started* to *Sold*, since *Started* is the first field of the enum *Phases* its value is zero, instead the value of *Sold* is two.

- Bids: this scenario is divided in two cases: the bid received is the first one or it will be the new highest.

  In the first case there is a first SSTORE on three zero variables: *highestBid*, *highestBbidder* and *bidBlock*. Then is emitted an event and the contract goes to *Sold* phase, for a total of $4 \times S + L$ gas units.

  In the second case the function is cheaper: non-zero information are updated for $3 \times S'$, the refund for the previous highest bidder is saved, for S gas units if is the first time his offer is outbid or for S' otherwise.

5 Testing guidelines

In this section it is described how to test the two contracts implemented. For this purpose each contract has a variable, *debug*, that if it's set to true at construction time it will disable the modifier *blockTimedTransaction* and allow to manually switch between phases through the *nextPhase* function. I assume that the contracts will be tested using Remix IDE.

5.1 Testing Vickrey auction

To test this contract follow the steps:

- Deploy: the constructor has the following signature:

```
constructor (address payable _seller, address payable _charity
    , uint _reserve_price, uint _commitment_phase_length, uint
     _withdrawal_phase_length, uint _opening_phase_length,
    uint _deposit_requirement, uint _escrow_duration, bool
    _debug) public;
```

The parameters *_seller* and *_charity* can be freely chosen among the ones proposed by the IDE, keeping in mind that they are not allowed to commit a bid. The various **_phase_length* are intended to by number of blocks and they can be ignored when debugging or testing. *_deposit_requirement* must be between one quarter and one half of *_reserve_price*.

An example call is the following:

```
constructor (..., ..., 1000 wei, 1, 1, 1, 500 wei, 50, true);
```

Note: this operation requires about 3,200,000 gas units.

- Execution: once the contract has been deployed it is in the *Commitment-Phase*, follow the following steps to simulate the auction:

  1. Choose one address, different from *sender*, *charity* and the one that deploied the contract (i.e. the auctioneer).
  2. Using the function *debugKeccak* generate a hash, for example calling *debugKeccak(0x01, 1000)* where the first value is a bytes32 nonce and the second is the bid promised.
  3. Using the hash of the previous step as paramter, call the function *commit* and send also an amount equal to *depositRequirement*, on success a *LogCommitment* event is emitted.
  4. Go to step 1 and repeat as many time desired for send other commitment (up to one commitment per address).
  5. To switch to the withdrawal phase call the *nextPhase* function with 1 as parameter, every time this function is called a *LogTransition* is emitted.
  6. Call *withdraw* function as many time as desired from the addresses used when committing. This function can be called only once for every address that committed. For every successful call an event is emitted.
  7. Once the withdrawals are finished, call *nextPhase(2)* to switch to the *OpeningPhase*.
  8. During the opening phase, who has not withdrawn at step 6 can open his envelope. In order to do this, choose a valid address and call the *open* function providing as parameter the correct nonce used when the bid was committed. When calling this function send also the correct amount of ether for the bid. Depending on the order of opening different event could be emitted, they are described in Section 2.2.1. Checking the value of the variables *highestBidder*, *highestBid* and *priceToPay* is possibile to gain information about how the auction is going.
  9. Call *nextPhase(3)* to switch to *Finished* state.

10. Now, using the address of the auctioneer, the seller or the buyer is possible to call the function *finalize* with a boolean as parameter. If the boolean is true than an escrow contract is deployed and its address is returned into a *LogEscrow* event, otherwise an amout equal to *priceToPay* is directely sent to the seller. In any case the funds left in the contract are sent to a charity address.

5.2 Testing English auction

To test this contract follow this steps:

- Deploy: the constructor has the following signature:

```
constructor (address _seller, uint _duration, uint8
    _min_increment, uint _buyout_price, uint _reserve_price,
    uint _unchallenged_interval, bool debug)
```

  The parameter *_seller* can be freely chosen among the ones proposed by the IDE, keeping in mind that the seller is not allowed to commit bids. The parameter *_min_increment* indicates the percentage of how greater a bid must be than the highest one in order to become the new highest whereas *_duration* and *_unchallenged_interval* indicates respectively how many blocks lasts the auction and how many blocks are necessary before a bid is declared the winning one. The meaning of the others parameters is trivial, note that *buyoutPrice* is required to be greater than *reservePrice*.

  An example call is the following:

```
constructor (..., 50, 25, 1500 wei, 550 wei, 5, true);
```

- Execution: once the contract has been deployed it is in the *Started* phase, follow the following steps to simulate the auction:

  1. Call the function *buyNow* from an address different to the one of the seller or the one that deployed the contract (i.e. the auctioneer) if you want to terminate the auction by buyout. In this case if the correct amount is sent, the contract will switch to *Sold* phase, as shown by the automata in Section 2.2.2. A *LogSoldByBuyout* is emitted.
  2. If it has been chosen to not terminate the auction by buyout, select an address and call the function *bid* with an amount at least equal to reserve price. Since this is the first bid, the contract will switch to *BidReceived* phase.
  3. Do as many bid as desired, calling the *bid* function with other addresses keeping in mind that the amount sent has to be greater than *highestBid + (highestBid * minIncrement / 100)*.
  4. Since the flow of time has been locked using the *debug* variable, when desired is possible to call *nextPhase(2)* to switch to *Sold* state, making the value stored in *highestBid* the winning one.

5. Now, whatever the auction is come to *Sold* state, it his possible to call the method *finalize* to end the auction. This method bring the auction to the *Finished* state and if it is called with true as parameter it deploy an escrow contract just like the *finalize* method of the Vickrey auction.

A Auction.sol

B Escrow.sol

C VickreyAuction.sol

D EnglishAuction.sol