# D3: Requirements

**D3: Requirements**: for each sub requirements (R1A → R4C) state how this was achieved or if it was not achieved. Explain where it can be found in the code. Use focused, short code extracts if they make your explanation clearer.

Classes that were identified as being necessary for the purposes of the specifications R1A -> R4C:

- Bot class
    - We decided the logger should output csv format, because it will be machine-readable.
    - The log class should have a single function, log, which takes as input a string rather than more structured data because it will because it will be used to log multiple different kinds of data and it's easier this way. It will have a constructor that takes a filename of the log file which it should write.
    - We are not sure what to do if the log file already exists, so we will replace it.
- Strategy class
    - This should be able to be swapped to try different bot trading strategies
- Offer class
    - We decided to represent bids and asks inside the bot using an "Offer" class which contains the amountOffered, typeOffered, amountWanted, typeWanted.
    - We decided not to differentiate bids from asks, instead we reversed the offer and wanted assets.
- Balance class
    - This is a representation of a wallet balance of a particular asset.
- Balances class
    - Representation of all balances of assets in the wallet
    - Able to be compared to another Balances object to create a list of ChangeOfBalance objects.
- ChangeOfBalance class
    - The old balance and new balance after a change of the balance of a particular asset.
    - This made it possible to be converted to a CSV for logging purposes.

You can see in Figure 1 a diagram of the software architecture:

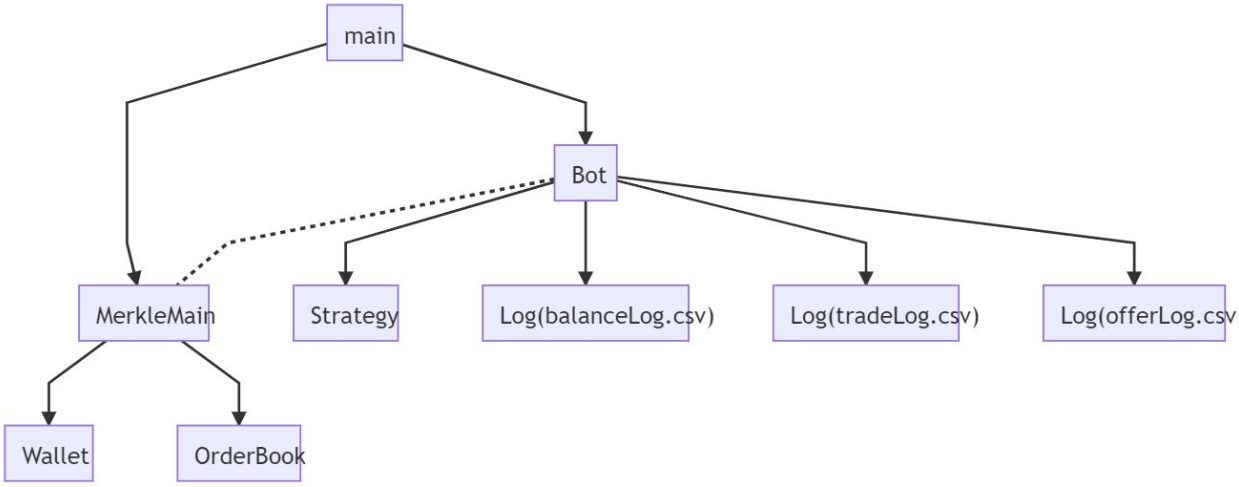


Figure 1

There are 2 ways the bot can interact with the Merkelrex program:

1. either the bot is in a separate program that sends key presses to the main program and reads the result
2. or the bot is integrated in the Merkelrex program.

The first solution requires spotting an external program and text parsing. The second solution requires modifications of the Mekelrex program to create an API which the bot can communicate with. We chose the second solution because the specification is ambiguous and it seems to be easier to do than the first.

To fulfill the specifications R1A -> R4C for the assignment, we decided that the best way in order for the bot to do its job, we would need 7 new functions in the Merkelrex app.

1. `getBalances()` - will return a list of number (the amount, ballance) and as well the type of the asset
2. `getBestOffers()` - will return a list of Offer which will contain the amountOffered, typeOffered, amountWanted, typeWanted
3. `makeOffer(Offer o)` - this takes an Offer and return an error if it's impossible to make the offer, otherwise nothing
4. `getMyOffers()` - returns a list of Offer, all the offers that our bot has as outstanding

5. `withdrawOffer(Offer oe)` - will remove / cancel an offer; returns an error if the offer does not exist, otherwise will be succeeded
6. `getAcceptedOffers()` - will return a list of the offers (Offer) that our bot has made which has been matched and traded
7. `getCurrentTime()` - Return the current (simulated) time so that the bot is able to determine the velocity of price changes. This was converted from the textual representation used in the app to a number of seconds since the UNIX epoch so that the bot could easily compute timespans.

We decided that the main loop of the bot would be as follows:

1. Get the balances from the wallet
2. Compare the current balances to the previous balances
    - From the comparison, this will yield an array of ChangeOfBalance, log each one in the balanceLog
3. Store the new balances in the field which previously held the old balances
4. Call getAcceptedOffers() to see what offers were accepted
    - If any offers were accepted then log these in the tradeLog
5. Call getMyOffers() to get the currently outstanding offers which we made ourselves
6. Call getCurrentTime() to get the current simulated time
7. Call getBestOffers() to get the current best offers in the orderbook
8. Call the strategy with:
    - the time
    - the current wallet balance
    - the current outstanding offers by the sim user
    - the best offers (for getting price info)
9. The strategy should return a list of offers to withdraw and offers to create
10. Withdraw and create offers
11. Repeat the loop

We decided to use an "abstract" class to represent the strategy, from searching on the internet we found that this could be achieved by using a pure virtual function. We created one implementation of the Strategy which we called SimpleStrategy. The logic of SimpleStrategy is to FOMO in and out of any assets where prices are moving "quickly". The definition of quickly is based on moving more than a given percent within a given time window.

SimpleStrategy has 3 configuration parameters:

- `TIMESPAN_SECONDS` - The length of the time window to use for deciding if prices are moving "fast".
- `TRIGGER_BUY` - If the price of an asset becomes greater than this (percent of price) in the time window then buy.
- `TRIGGER_SELL` - If the price of an asset becomes less than this (percent of price) in the time window then sell.

We decided that to avoid complexity we would only trade BTC pairs and because BTC is a comparatively safe asset we would buy "some" of a given asset but on TRIGGER_SELL we would sell all of it. However since the bot is triggered for each trade, we needed to avoid buying more and more and more of an asset until we had 100% exposure, so to fix this we limited our buying of any one asset to 25% of our available BTC.

We found that on an Apple computer, the bot crashed due to a use-after-free memory error which we were unable to identify. On Windows, as seen in Figure 2, the bot did work but slowly as the orderbook grew very large with unmatched orders, we think this is correct behavior and that the orderbook is simply big.

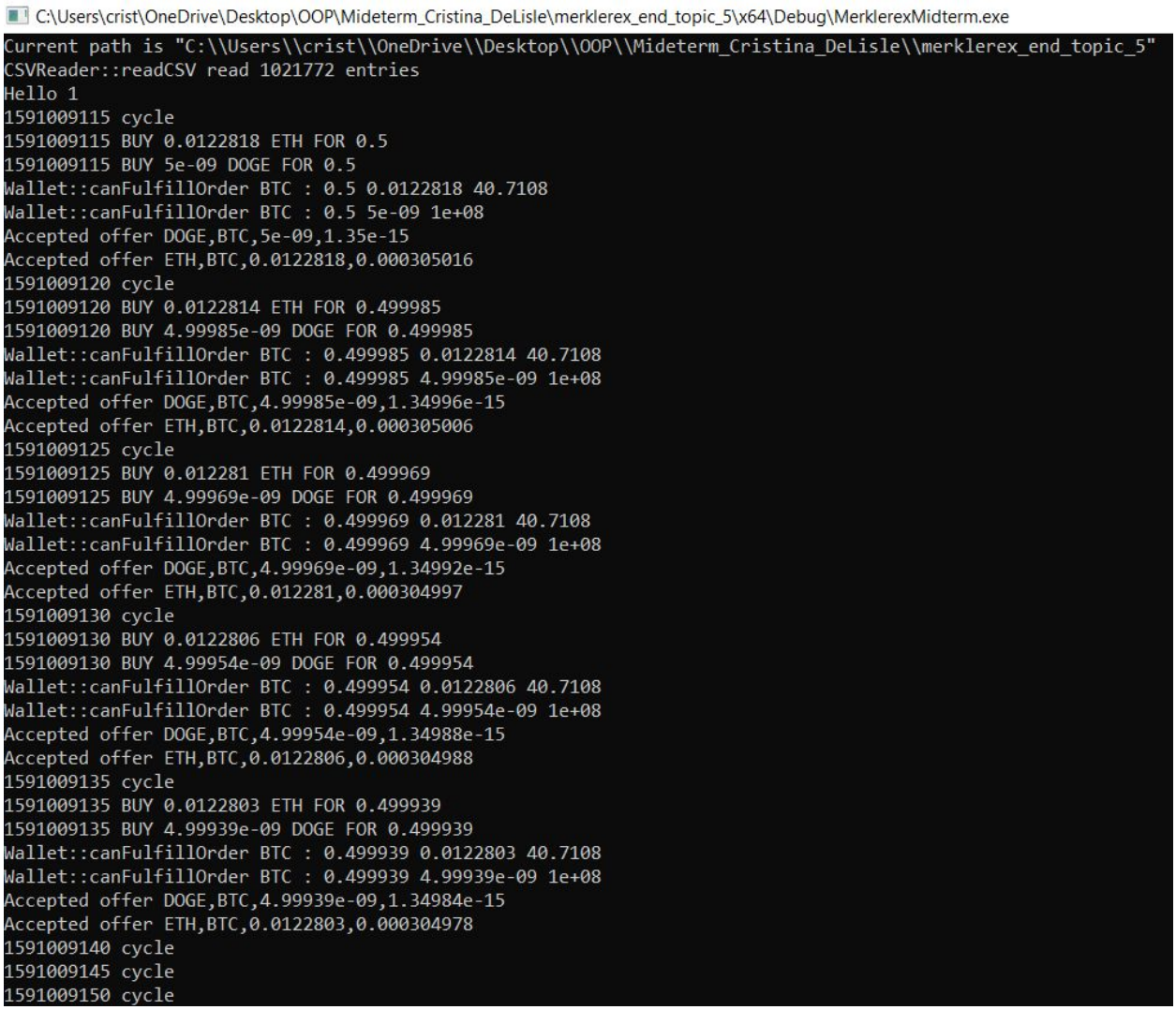


Figure 2

# D4: Algorithms

**D4: Algorithms**: where you created algorithms, e.g. for predicting future market, explain each algorithm and state where it can be found in the code. Use focused, short code extracts if they make your explanation clearer.

The bot has the following algorithmic structure, in the main loop:

- Get the balances from the wallet
- Compare the current balances to the previous balances

    - From the comparison, this will yield an array of ChangeOfBalance, log each one in the balanceLog
- Store the new balances in the field which previously held the old balances
- Call getAcceptedOffers() to see what offers were accepted
    - If any offers were accepted then log these in the tradeLog
- Call getMyOffers() to get the currently outstanding offers which we made ourselves
- Call getCurrentTime() to get the current simulated time
- Call getBestOffers() to get the current best offers in the orderbook
- Call the strategy with:
    - the time
    - the current wallet balance
    - the current outstanding offers by the sim user
    - the best offers (for getting price info)
- The strategy should return a list of offers to withdraw and offers to create
- Withdraw and create offers
- Repeat the loop.

# Challenge requirement

**Challenge requirement**: Document the optimisation you made to the exchange code, if you did this, as described above.

There were multiple improvements that were necessary to be performed in order to make the bot follow the specifications. For the purposes of the challenge requirement, the notable one to be mentioned consists of the changes to the orderbook. We found a few issues with the orderbook, firstly it traverses the entire order history numerous times in the operation of the program, this was fixed by establishing a list of "currently active" orders.

Secondly and perhaps more problematic, the matchAsksToBids() function seems to have been wrong initially because it was only matching bids and asks which happened to have been made in the same time period rather than carrying them over until they can be matched. We improved this, creating a running list of active orders, but unfortunately we found that as the order history piles up, the number of open orders becomes very large and we still do significant manipulation of the active orders list which meant that the performance of the program was improved, but not nearly as much as we would have liked.