**Code Quality exercise**

Considering the code in ​src/smelly\_code/trade\_execution\_service.rb:

1. How would you improve the code quality (testability, readability, maintainability...) of the previous Service object? Please enumerate any code smells or bad practices that are present in the code above.

**1.1 There is a gap when the amount is equal to 100k USD, there is not a specific rule to select a provider when the amount is 100k**, but the code assumes that an amount equal or bigger than 100k is going with the provider A. It is needed to validate with the customer or user the correct rule.

**1.2 It is necessary a method (function) to validate the entries pararameters of the method "execute\_order" before to execute it.**

execute\_order (side, size, currency, counter\_currency, date, price, order\_id)

**- side:** Validates the possible values: 'buy', 'sale'

**- size:** Validates that is a numeric value > 0, not negative or zero values are allowed

**- currency:** Validates the possible allowed values: 'USD'. According with the bussiness rules based on 'USD', this service will only accept 'USD' as a currency.

**- counter\_currency:** Validates the possible allowed values from a list of currencies that the company works with: 'EUR', 'JPY', etc., always allowing 3 letters, non numeric values are allowed. If the trade is only between USD and EUR, then the only valid value is ‘EUR’.

**- date:** Validates a correct date format dd/mm/yyyy or mm/dd/yyyy. It is pending to confirm the correct format defined.

**- price:** Validates a numeric value > 0, can be floating, with the decimal symbol '.', not ','. I recommend also if there is a limitation for the maximum price, what happens if this field is bigger than 2, for example. Maybe we can have a reference value and generate a warning.

**- order\_id:** Pending to validate if any particular format for this field exists and the maximum of characters allowed.

**1.3 The method 'amount\_in\_usd' is not necessary to use it to define the Liquidity Provider 'lp'.** You can use directly the parameter size:

if size < 10\_000

lp = LIQUIDITY\_PROVIDER\_C

elsif (size >= 10\_000 && size < 100\_000)

lp = LIQUIDITY\_PROVIDER\_B

else

lp = LIQUIDITY\_PROVIDER\_A

end

or to be sure that there is not issue with rounding, then use the method 'amount\_in\_usd´, but **make more strong the first condition to avoid amount <=0:**

if (amount > 0.to\_money(USD) && amount < 10\_000.to\_money(USD))

lp = LIQUIDITY\_PROVIDER\_C

elsif (amount >= 10\_000.to\_money(USD) && amount < 100\_000.to\_money(USD))

lp = LIQUIDITY\_PROVIDER\_B

else

lp = LIQUIDITY\_PROVIDER\_A

end

**1.4 When an exception happens, it is better to return more details about the possible error, not just return "Execution of order\_id failed".** I recommend to capture the exception and return it in the log file. In this way, we will help the one who is calling the method and show where the problem is, maybe in the entry parameters, or in the web connection or in the DB.

**1.5 In the method 'issue\_fix\_market\_trade', it is better to catch and handle the exception in case of the method ‘check\_fix\_service\_status(lp)’ returns it**. In this way we can get the specific error and put it in the log file.

**1.6 What would happen if in the future we have more providers?** Currently we need to update the code. I suggest to make a reference table (txt file, DB table, etc.) where we can load:

- each provider

- a range of order amount that can be handled by each provider

- the protocol used by the provider

- the currency used by the provider.

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Provider** | **Min Range** | **Max Range** | **Protocol** | **Currency** |
| LIQUIDITY\_PROVIDER\_C | 0 | <10K | REST | USD |
| LIQUIDITY\_PROVIDER\_B | >=10K | <100K | FIX | USD |
| LIQUIDITY\_PROVIDER\_A | >=100K |  | FIX | USD |
| LIQUIDITY\_PROVIDER\_D | 0 | <=50K | REST | JPY |
| ... |  |  |  |  |

In this way the code can be generic and does not depend of the current providers and can have the flexibility to handle other currency different from 'USD'. This depends from the bussiness rule.

2. Refactor the code in order to amend the issues described above. Feel free to decompose current TradeExecutionService into a set of smaller objects or services if needed. Push the new implementation into the src/clean\_code/ folder.

Done. Code delivered in the mentioned folder

3. Write tests (in Rspec ) to verify that the current code implementation satisfies the following order routing rules:

|  |  |
| --- | --- |
| Order amount | routed to |
| Less than 10k USD | LIQUIDITY\_PROVIDER\_C |
| Equal or bigger than 10k USD but less than 100k USD | LIQUIDITY\_PROVIDER\_B |
| Bigger than 100k USD | LIQUIDITY\_PROVIDER\_A |

See the file \code\_quality\_exercise\src\clean\_code \ trade\_execution\_service\_ver01\_spec.rb

4. Make sure all ruby code compiles and tests pass. Add require statements when needed, use a Gemfile to specify versions and Bundler to install any required libraries (gems).

See the files Gemfile, Gemfile.lock, Bitacora.txt, errors.log in Github repository.