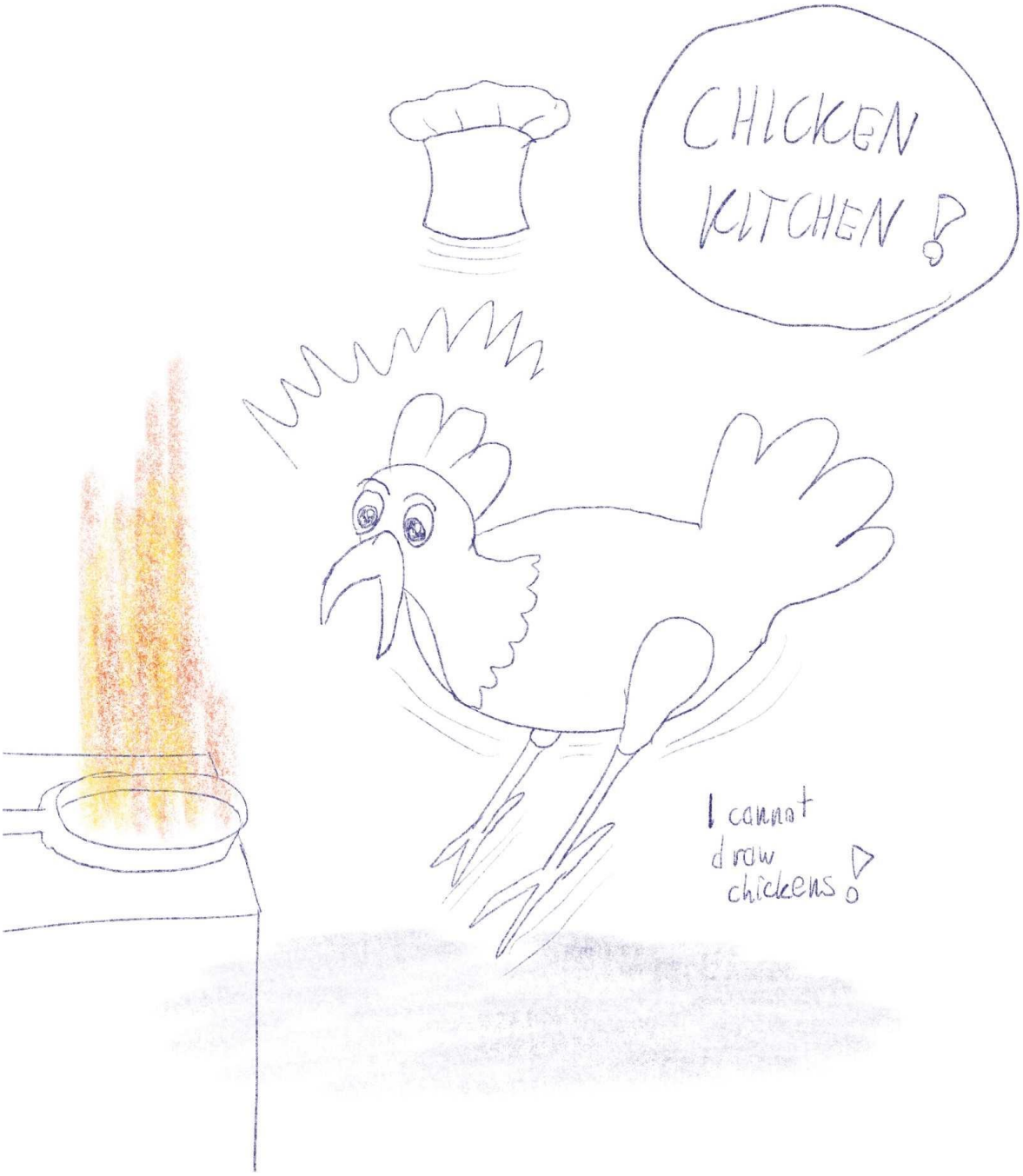
Chicken Kitchen Extreme, version 211123. Author: Żółw.



(Yes, I have drawn that).

Contents

1. What is this exercise – an introduction

This exercise is written individually (you do not cooperate with other people while writing it). It starts relatively simple but then expands a bit more. Should take 1 week at most and is language-agnostic. I have written that in C#, I have seen it written in Javascript/Node.JS.

This exercise tests:

- Ability to write a code based on documentation and data
- Ability to modify your own existing code
- Ability to write test cases and hold multiple simultaneous stuff a program needs to do
- Ability to write some horrible, horrible stuff from time to time.
    - (I promise – no regular expressions *this time*. I am not a complete monster, after all)

2. Short Story

Congratulations, Commander.

You are now an owner of a small restaurant called „Chicken Kitchen”. It has nothing to do with chickens, but it does contain a kitchen. And customers, of course.

You have some nice dishes here, fan favorite. Some returning customers. And some basic ingredients.

3. Your purpose – how this works?

You have a set of DATA. It is:

- Regular Customers
- Food
- Base Ingredients

Food is made of Basic Ingredients. Say:

Food	**Ingredients**
Princess Chicken	Chicken, Youth Sauce
Youth Sauce	Asparagus, Milk, Honey
Base ingredients	
Chicken, Tuna, Potatoes, Asparagus, Milk, Honey, Paprika, Garlic, Water, Lemon, Tomatoes, Pickles, Feta, Vinegar, Rice, Chocolate	

So, Princess Chicken is made of Chicken and Youth Sauce. And Youth Sauce is made of Asparagus, Milk, Honey. So, Princess Chicken is made of Chicken, Asparagus, Milk, Honey.

Now, let's apply a regular customer:

Regular customer	**Allergies**
Julie Mirage	Soy

If Julie Mirage wants to buy Princess Chicken, she can – because she is allergic to Soy and Princess Chicken has no Soy in its ingredients (only Chicken, Asparagus, Milk, Honey).

But poor Julie Mirage (allergic to soy) cannot buy, say Soy Chicken (ingredients: Chicken, Soy). We must refuse her service.

4. Data

Regular customer	Allergies
Julie Mirage	Soy
Elon Carousel	Vinegar, olives
Adam Smith	-
Barbara Smith	Chocolate
Christian Donnovan	Paprika
Bernard Unfortunate	Potatoes

Food	Ingredients
Emperor Chicken	Fat Cat Chicken, Spicy Sauce, Tuna Cake
Fat Cat Chicken	Princess Chicken, Youth Sauce, Fries, Diamond Salad
Princess Chicken	Chicken, Youth Sauce
Youth Sauce	Asparagus, Milk, Honey
Spicy Sauce	Paprika, Garlic, Water
Omega Sauce	Lemon, Water
Diamond Salad	Tomatoes, Pickles, Feta
Ruby Salad	Tomatoes, Vinegar, Chocolate
Fries	Potatoes
Smashed Potatoes	Potatoes
Tuna Cake	Tuna, Chocolate, Youth Sauce
Fish In Water	Tuna, Omega Sauce, Ruby Salad
Irish Fish	Tuna, Fries, Smashed Potatoes

Base ingredients
Chicken, Tuna, Potatoes, Asparagus, Milk, Honey, Paprika, Garlic, Water, Lemon, Tomatoes, Pickles, Feta, Vinegar, Rice, Chocolate

5. What I want from you:

Your primary purpose is to deliver an application performing **all the functions without any bugs according to specification.** And to be able to prove you have done it using automated tests.

At start, I want you to write **a program** which will solve the following test cases:

1. Julie Mirage wants to buy Fish in Water. She gets her Fish in Water.
2. Elon Carousel wants to buy Fish in Water. You have to refuse him service, because one dependency uses Vinegar and he is allergic to it.
3. Julie Mirage wants to buy Emperor Chicken. She gets her Emperor Chicken.
4. Bernard Unfortunate wants to buy Emperor Chicken. You have to refuse him service, because one dependency uses Potatoes and he is allergic to them.

Start from the simplest case and work up. Say, start from a customer who buys only fries and has no allergies. Then when that works, work up from this. Remember how I did it with Magic: The Gathering cards (Dream Trawler, first implemented the easy card)? Take the **simplest** case, next case, build up…

Initial inputs / outputs:

- For algorithm:
    - INPUT:
        - Assume you have four text files, csv-formatted (comma separated variables):
            - Regular Customers – Allergies
            - Food – Ingredients
            - Base Ingredient List
        - Console. Write who buys what. Assume case is irrelevant
            - Example: “Julie Mirage, Fish in water”.
    - OUTPUT
        - For now, Console will output an answer:
            - Example of success: “Julie Mirage – Fish in Water: success”
            - Example of failure: “Elon Carousel - Fish in Water: can’t order, allergic to: Chocolate”

Quality requirements on the code:

- I expect to see automated tests proving that main cases are working. In case of C# I usually use NUnit. In case of Node, I used JEST if I remember properly.
- I expect that your work will be committed and pushed to some Gitlab repository. If you don’t know how to use Git, set up gitignore etc., please tell the leader – you don’t have to do it right now in this case.
    - **At the very least** commit after every phase (base code, change 1, change 2 etc). This code will be modified A LOT, so you want to return to something “that works” in case you fail at one point.
- I think you will separate the code into many files (C# files or Typescript files). Personally, I think this application is impossible to write in one file as a whole – I would get lost.

Output:

I am giving you examples of outputs in this document. Those are just that – examples. You don’t have to give me “exactly the same strings”. I simply want to see all the data.

6. Next phases

6.1. Change 1: Everything is better if you have money.

Base ingredients have prices now:

Base ingredients
Chicken: 20, Tuna: 25, Potatoes: 3, Asparagus: 50, Milk: 5, Honey: 15, Paprika: 4, Garlic: 3, Water: 1, Lemon: 2, Tomatoes: 4, Pickles: 2, Feta: 7, Vinegar: 1, Rice: 2, Chocolate: 5

Let's look at one dish. Say, Princess Chicken.

- Princess Chicken → Chicken, Youth Sauce → Chicken, Asparagus, Milk, Honey
    - Cost: Chicken: 20, Asparagus: 50, Milk: 5, Honey: 15 → 90

So, the **cost** of this dish is 90 (dollars? Whatever; I wouldn't eat it if you PAID me).

Ah, of COURSE if a dish has multiple of the same ingredient, you count money many times. Say - Irish Fish:

- Irish Fish → Tuna, Fries, Smashed Potatoes → Tuna, Potatoes, Potatoes → 25 + 3 + 3 = 31 (not 28)

Also, our regular customers also have budgets:

Regular customer	**Allergies**	**Budget**
Julie Mirage	Soy	100
Elon Carousel	Vinegar, olives	50
Adam Smith	-	100
Alexandra Smith	-	500
Barbara Smith	Chocolate	250
Christian Donnovan	Paprika	500
Bernard Unfortunate	Potatoes	15

Say, Julie Mirage wants to buy Princess Chicken. Her budget is 100 and we established that the Princess Chicken costs 90. Therefore, we can sell it to her. But if she wants to buy a SECOND Princess Chicken (without restarting a program), she has budget of only 10. So, she can't buy it anymore and we should send a message like: "Julie Mirage – can't order, budget 10 and Princess Chicken costs 90".

Of course, add proper automated tests and modify old ones.

6.2. Change 2: Batches, dynamic data, budgeting and fighting for profit

6.2.1. Batches and dynamic data

We will modify inputs. It is very, very *unwieldy* to work with Console. So, we shall introduce a new input and output.

Input – batch file. Say, new text file, looks like this:

“

```
Buy, Julie Mirage, Princess Chicken
Buy, Barbara Smith, Tuna Cake
```

”

And output for those would look more or less like this (I don’t care what EXACTLY is the text of the strings as long as I get all the information):

“

```
Buy, Julie Mirage, Princess Chicken -> Julie, 100, Princess Chicken, 90 -> success
Buy, Barbara Smith, Tuna Cake -> Barbara, 250, Tuna Cake, XXX -> can’t buy, allergic to Chocolate
```

“

Also – from now on assume you can get a malformed file. Either batch file (this one to inform what to do) or some ingredient files. If there is a damaged record, you **skip** that record and read everything else. If a customer asks for a food which does not exist in a file, give an appropriate output (for example “Elon can’t buy Haggis, cook has no idea how to make it”).

So, if you get a file like this:

“

```
Buy, Julie Mirage, Princess Chicken
Buy, Elon Carousel, Tuna Cake
Sadkl,jaslkdjasldkjaskldjsa
Buy, Julie Mirage,
Buy, Adam Smith, Fries
```

”

You should process only the lines which are NOT white-on-black in the above example – 1, 2, 5 (the “sakjdsalkdj” is impossible to parse and “Buy, Julie Mirage, “ lacks one parameter – so those lines should be skipped)

You might have to change all your automated tests to make them function without relying on external text files. If you don’t know how, tell your leader. Either a mentor or I will explain asap.

6.2.2. Restaurant budget

- At start, by default, your restaurant has the budget: 500.
- From now on you sell stuff at 30% markup.
    - Princess Chicken costs 90 (as we established earlier)
        - So you sell it for 90 * 1.3 = 117
- At the beginning of the output batch file write how much money you have
- At the end of the output batch file please write how much money you have

Example of output file:
“

Restaurant budget: 500

Buy, Alexandra Smith, Princess Chicken -> Alexandra, 500, Princess Chicken, 117 -> success

Buy, Barbara Smith, Tuna Cake -> Barbara, 250, Tuna Cake, XXX -> can’t buy, allergic to Chocolate

Restaurant budget: 617

“

If at **any time** budget drops below 0, restaurant will go bankrupt. Instead of output of EVERY NEXT ORDER write “RESTAURANT BANKRUPT”:

INPUT

“

```
Budget, =, 100
Buy, Adam Smith, Fries
Order, Tuna, 1000
Buy, Adam Smith, Fries
Budget, =, 1000
```

“

OUTPUT

“

```
Restaurant budget: 500
Restaurant budget: 100
RESTAURANT BANKRUPT
RESTAURANT BANKRUPT
RESTAURANT BANKRUPT
```

“

6.2.3. Budgeting and Ordering ingredients

You have two more commands – Order and Budget you can write in a batch file:

“

```
Budget, =, 300
Buy, Julie Mirage, Princess Chicken
Buy, Elon Carousel, Tuna Cake
Order, Tuna, 5
```

”

Order command is relatively simple:

- You want to buy the ordered ingredients at cost of ingredient
- Syntax
  - Order, <WHAT>, <HOW MANY>
- So “*Order, Tuna, 5*” means “buy 5* Tuna” -> subtract 5*25 from budget.
- You cannot order something which is NOT a basic ingredient. So, you can’t order, say, Princess Chicken.

Budget command is a bit more fun:

- You want to modify the restaurant budget
  - Budget, =, 300 → no matter what the budget was, now it is 300.
  - Budget, +, 300 → add 300 to budget
  - Budget, -, 300 → subtract 300 from budget
- Syntax
  - Budget, <SIGN>, <AMOUNT>

6.3. Change 3: The nightmare of restaurant tables

How often does a single person go to a restaurant? Rarely, right? So, it's time we get a new mode of operation – from now on we can work with single people or with a group of people.

We add a new command – **Table**.

The **Table** command works as follows:

- Syntax
  - Table, <PERSON 1>, <PERSON 2>… <FOOD FOR PERSON 1>, <FOOD FOR PERSON 2>…
- Example (yes, all examples should be added as separate automated test cases, why do you ask?)
  - *Table, Julie Mirage, Princess Chicken → should work like Buy, Julie Mirage, Princess Chicken* (except different format of output)
  - *Table, Alexandra Smith, Adam Smith, Irish Fish, Fries →*
    - Alexandra Smith gets Irish Fish and pays for Irish Fish only
    - Adam Smith gets Fries and pays for Fries only
  - *Table, Alexandra Smith, Bernard Unfortunate, Irish Fish, Fries →*
    - FAILURE. Bernard is allergic to potatoes, and they are in fries. So, he can't buy it. So, whole table fails.
  - *Table, Alexandra Smith, Adam Smith, Fries →*
    - ERROR. Every person needs something to eat. So, whole table fails.
  - *Table, Alexandra Smith, Fries, Irish Fish →*
    - ERROR. One person can have one type of food only. So, whole table fails.
  - *Table, Adam Smith, Adam Smith, Fries, Fries →*
    - ERROR. One person can appear only once at the table. So, whole table fails.
- In other words – if everything at a table works, we have a go (it works). Otherwise, do not proceed with the transaction.

Example of a batch file using a Table command:

"

Buy, Julie Mirage, Princess Chicken
Buy, Barbara Smith, Tuna Cake
Table, Alexandra Smith, Adam Smith, Irish Fish, Fries

"

How an output of the above looks like (XXX below because I didn't calculate it):

"

Buy, Julie Mirage, Princess Chicken -> Julie, 100, Princess Chicken, 117 -> can't buy, cannot afford it.
Buy, Barbara Smith, Tuna Cake -> Barbara, 250, Tuna Cake, XXX -> can't buy, allergic to Chocolate
Table, Alexandra Smith, Adam Smith, Irish Fish, Fries -> success; money amount: 45
{
Alexandra Smith, 500, Irish Fish, 41
Adam Smith, 100, Fries, 4
}

"

6.4. Change 4: Warehouses and quantities

6.4.1. How warehouses work

We get an extra file – a warehouse. It will tell us *how many things we already have in the inventory*. It will look more or less like this:

“

Chicken, 10, Tuna, 10, Potatoes, 10, Asparagus, 10, Emperor Chicken, 1, Fries, 3

“

So, it can have both the amount of base ingredients and the number of prepared dishes.

- If the warehouse file does not exist, assume all base ingredients to be at 5 and all prepared dishes to be at 0.
- If there is a prepared dish in warehouse file and someone asks for that dish, use it instead of making a new dish.

Example 1:

Warehouse: “*Potatoes, 10, Asparagus, 10, Emperor Chicken, 1, Fries, 3*”

Input file:

“

Buy, Julie Mirage, Fries

“

What happens:

- Warehouse has both Potatoes and Fries. Julie gets her fries.
    - Warehouse on output: “*Potatoes, 10, Asparagus, 10, Emperor Chicken, 1, Fries, 2*”

Example 2:

Warehouse: “*Potatoes, 10, Asparagus, 10, Emperor Chicken, 1*”

Input file:

“

Buy, Julie Mirage, Fries

“

What happens:

- Warehouse doesn’t have Fries. So, the cook has to make Fries from Potatoes. Julie gets her fries.
    - Warehouse on output: “*Potatoes, 9, Asparagus, 10, Emperor Chicken, 1*”

6.4.2. What do we do with quantities?

- If an order fails **because of allergy**, all ingredients and dishes are lost, and we don't get any profit.
- If an order fails due to any other reason, ingredients and dishes are not lost.

Example:

Initial warehouse: "Princess Chicken, 1, Tuna, 5, Chocolate, 5, Youth Sauce, 3, Potatoes, 10"

Command:

"

Buy, Julie Mirage, Princess Chicken
Buy, Barbara Smith, Tuna Cake
Buy, Alexandra Smith, Fries

"

Julie can't afford Prince Chicken (failure; can't afford). Barbara Smith can't buy Tuna Cake (allergy to Chocolate). Alexandra Smith can buy fries.

So, warehouse at end: "Princess Chicken, 1, Tuna, 4, Chocolate, 4, Youth Sauce, 4, Potatoes, 9"

- Tuna Cake needs 1 Tuna, 1 Chocolate, 1 Youth Sauce and Barbara is allergic to Chocolate
- Julie can't afford Princess Chicken. As it is not due to allergy, we don't reduce ingredients
- Alexandra can buy Fries, so she buys Fries.

- If something is ordered and we can't make it because we lack ingredients or prepared dishes, give it as a reason for an error.

6.4.3. The Order command is not useless anymore

Until now, Order command did just one thing – it allowed us to reduce the budget. Now it allows us to increase the quantity of base ingredients in the warehouse.

Order, new version:

- You want to buy the ordered base ingredients at cost of ingredient
- Syntax
  - Order, <WHAT>, <HOW MANY>
- So "*Order, Tuna, 5*" means "buy 5* Tuna"
  - subtract 5*25 from budget.
  - Add 5 Tuna to warehouse
- You cannot order something which is NOT a basic ingredient. So, you can't order, say, Princess Chicken.

6.5. Change 5: The darkness of audits

Business sure loves their audits. We get a new command, **Audit**.

Audit:

- You want to get information on stuff and write it in separate Audit File.
- Syntax
    - Audit, Resources
- This shows ALL changes in budget and warehouses for every step from the start to the moment an Audit command was called
- Example:

Command:

“
Buy, Alexandra Smith, Fries
Order, Potatoes, 10
Buy, Alexandra Smith, Fries
Audit, Resources
Buy, Alexandra Smith, Fries
”

Will generate the audit file:

“
INIT
Warehouse: potatoes, 10, tuna: 10, …
Budget: 500

START

command: Buy, Alexandra Smith, Fries -> Alexandra Smith, 500, Fries, 4 -> success
Warehouse: potatoes, 9, tuna: 10, …
Budget: 504

command: Order, Potatoes, 10 -> success
Warehouse: potatoes, 19, tuna: 10, …
Budget: 474

command: Buy, Alexandra Smith, Fries -> Alexandra Smith, 496, Fries, 4 -> success
Warehouse: potatoes, 18, tuna: 10, …
Budget: 478

AUDIT END

“

Audit command is “invisible” for the output file, leaves no trace in output file. It creates a new file.

6.6. Change 6: Command configuration conundrum

We don't necessarily want everyone to be able to use every command. For example, we might want to disable "Order" command for an intern, don't you think? 😊

So, from now on we have a JSon file with command names:

```
{
    "order": yes,
    "buy": yes,
    "audit": no,
    "table": no,
    …
}
```

By default, all commands are **disabled**. So, if a command is not enabled, we don't have access to it. If a command exists in a system and is not present in the JSon, it is also disabled.

If you call a command which is disabled, we will get an error message in the output calling "command disabled" and no transaction takes place, ingredients are not lost etc.

So, for the JSon config file presented above we would have the following:

INPUT:

```
"
Buy, Julie Mirage, Princess Chicken
Table, Barbara Smith, Tuna Cake
Morningstar, Alexandra Smith, Adam Smith, Irish Fish, Fries
"
```

OUTPUT:

```
"

Buy, Julie Mirage, Princess Chicken -> Julie, 100, Princess Chicken, 117 -> can't buy, cannot afford it.
Table command disabled
Morningstar command disabled
"
```

And because "command disabled" takes precedence over "take the ingredients", you don't have to reduce the ingredients in Table command.

6.7. Change 7: Welcome to legacy code of your own doing.

6.7.1. Introduction

We haven't tested how well you have written your code, right? If your structure fails, you have a hopeless "if-else" tree combined with some private Boolean flags, and you are desperately trying to find yourself in this mess.

This change tests the robustness of your code. It's a lot of small changes to different commands. If your code is not structured good enough, this change will break it. **Remember – commit before starting to work on this**.

You may consider looking at the codebase before you proceed with this change. Maybe some refactoring will help. Some structure. Responsibility separation and all that. **If you have no idea what I am talking about, tell a leader.**

If you have written all the automated tests you should have up to this point, you will find this task irritating but possible. If you didn't, well… good luck 😊.

6.7.2. On budgets and taxes

Our JSon file will have additional configurations:

```
{
   …
   “profit margin”: 40,
   “transaction tax”: 10,
   “daily tax”: 20
   …
}
```

- Profit margin
  - Profit margin until now was fixed at 30%. We introduced it in “6.1.1. Restaurant budget”.
    - Princess Chicken has ingredient cost of 90, but we sell it for 90 * 1.3 = 117
  - From now on profit margin will be read from configuration file
    - default profit margin is 30%. If not present in config file, use default.
- Transaction tax
  - Transaction tax until now was fixed at 0%.
  - Assume profit margin at 30% and Princess Chicken
    - Princess Chicken has ingredient cost of 90, but we sell it for 90 * 1.3 = 117
    - If transaction tax is 10%, we are taxed 117 * 0.1 = 12
    - So, we pay **12** in taxes and add **105** to the restaurant budget.
  - Transaction tax concerns EVERY transaction, not just buying
  - So, if we do Order, Tuna, 5
    - we would pay 25*5 = 125 with transaction tax 0.
    - we are taxed **13** with transaction tax 10
    - so, we pay **138** to get 5*Tuna.
  - From now on, transaction tax has default of 10%. If not present in config file, use default
- Daily tax is the tax we pay at the end of the day (at the end of single run of the program).
  - We want to tax only **profits** and we don’t want to tax twice.
  - So, we calculate:
    - profit = end restaurant budget – start restaurant budget – already collected tax
    - daily tax = profit * 0.2 by default
    - if daily tax is below 0, we pay tax of 0.

Write the daily tax at the end of the output file.

Show transaction tax in every line where that tax appears. For example, new output:

“

Buy, Alexandra Smith, Fries -> Alexandra Smith, 500, Fries, 5 -> success; money: 4, tax: 1

“

Ah, you are aware that **Audit** command will monitor taxes as well, right? It should 😊.

6.7.3. Automated coupons - discounts for every third appearance

Let's consider a following input file:

```
“
Buy, Alexandra Smith, Princess Chicken
Buy, Adam Smith, Tuna Cake
Buy, Alexandra Smith, Fries
Table, Alexandra Smith, Julie Mirage, Fries, Fries
Buy, Alexandra Smith, Tuna Cake
”
```

Alexandra Smith appears 4 times in that file. I would like to focus your attention on her third appearance (bolded).

Every three **successful** transactions a person gets a discount for that single transaction. And how much is the discount? Why, of course – it comes from the JSon file:

```
{
    …
    “every third discount”: 10,
    …
}
```

By default, discount is 0%.

Write the fact that there was a discount and how high it was in an output file. Format depends on you.

6.7.4. Warehouses and maximum size

From now on warehouses have maximum size. Of course, configured in JSon:

- Total maximum: warehouse can't have more stuff inside in total than this threshold.
- Max ingredient type: maximum amount of a single base ingredient type
- Max dish type: maximum amount of a single prepared dish type

In JSon (this here are the defaults from now on):

```
{
   …
   "total maximum": 500,
   "max ingredient type": 10,
   "max dish type": 3
   …
}
```

How it works:

Assume a warehouse:

"

Chicken, 8, Tuna, 9, Potatoes, 10, Asparagus, 10, Emperor Chicken, 1, Fries, 3

"

Sum of everything here is: 8+9+10+10+1+3. So current TOTAL is 41. Total Maximum is 200.

TOTAL stuff in warehouse cannot ever exceed total maximum.

You add one more Emperor Chicken to the warehouse (because there is only one and max dish type is 3). But you can't add one more Fries to the warehouse – it will be wasted.

You can add 2 more Chicken and 1 Tuna to warehouse, because max ingredient type is 10. But you can't add any more asparagus or potatoes.

If at any time anything gets wasted, you should write it in output file. Something like:

"

Wasted: 3 potatoes (limit: 10)

"

Ah, you are aware that **Audit** command will monitor the waste of stuff as well, right? It should 😊.

6.7.5. Allergies and warehouses

First, new JSon configuration (as usually, the value in the example below is the default):

```
{
   …
   “dishes with allergies”: waste
   …
}
```

You have a new field “dishes with allergies”. It can be in three states:

- waste
    - Works as it used to
    - If someone asks for a dish and they have an allergy to that dish, ingredients are wasted.
- keep
    - Put the newly created dish in the warehouse instead of wasting ingredients
    - Pay extra 25% of the cost of the dish (this is not taxed); represents the heater
    - Still fail the operation (“failure, allergies et al”)
    - Conditions:
        - if there is space in the warehouse
        - if you can afford it
- <NUMBER> (for example, 20)
    - If the dish exceeds the cost of <NUMBER>, keep. If it doesn’t, waste.

6.7.6. The Budget command can reverse bankruptcy

From now on, Budget command can override bankruptcy. So even if the company is bankrupt, if you do:

“

```
budget, +, 500
budget, =, 500
```

“

Then bankruptcy will be annulled from now on (assuming the amount of money is enough to get company to positive values).

6.7.7. The Order command – order whatever (according to config)

We are back to Order command. It changed considerably.

```
{
   …
   “order”: ingredients,
   …
}
```

It can have the following states:

- No
  - disabled
- Ingredients
  - Enabled, allows to order only base ingredients (default, works as it used to)
- Dishes
  - Enabled, allows to order only the prepared dishes
- All
  - Enabled, allows to order both base ingredients and prepared dishes

6.8. Change 8: Random hopes spoil and go to trash

This change might require you to pass something new (random generator) to your functions / methods. Make git commit before you start working on this one.

High level:

- Every time we execute an operation influencing the warehouse there is a chance that every instance of an ingredient has been spoiled and we need to throw it to trash.
- Trash has a limit, and we need to throw trash away if we are close to a limit.
- We can order multiple things.
- We can throw trash away.

6.8.1. Order multiple stuff

High level: We can order multiple things.

New syntax for Order is still simple:

- Syntax
  - Order, <WHAT1>, <HOW MANY1> , <WHAT2>, <HOW MANY2> …
- So "*Order, Tuna, 5*" means "buy 5* Tuna"
  - -> buy 5 * Tuna exactly like in the past
- So "Order, *Tuna, 10, Potatoes, 5, Sunflower seeds, 7*"
  - -> buy 10 * Tuna, 5 * Potatoes, 5 * Sunflower seeds, same conditions like in the past

If the parameters are missing ("Order, 10, Tuna, 10" or so) **do nothing** – skip this operation, do not buy anything, do not change the budget, do not change the warehouse state etc. In other words – it is as if nothing was ever called.

Yes, if configured as such, you can order dishes and ingredients in the same order command:

- *"Order, Emperor Chicken, 3, Potatoes, 2"* is a perfectly valid record.
  - If configured to "order: all" in JSon, it will perform appropriate stuff.
  - If configured to "order: ingredients" in JSon, it will be a **malformed record** -> thus **do nothing**; as if Order was never called.

As old syntax for Order is a subset of new Syntax, this change should not be problematic to you.

6.8.2. Spoiling of basic ingredients

High level: every time we perform an operation influencing the warehouse there is a chance that basic ingredients will spoil, cannot be used, and need to be moved to trash.

Details:

- Operations influencing the warehouse:
  - If you order something, you add a set of ingredients to a warehouse. For every single operation of ordering check if ingredients got spoiled once.
    - If you write “order, 100, potatoes”, you check all ingredients once.
    - If you write “order, 1, potatoes”, you check all ingredients once.
    - If you write “order, 100, potatoes, 50, tuna”, you check all ingredients once.
    - So, any time you call an Order command you call the check once
  - If you make food, check it as many times as you made some food.
    - If you make fries, you do:
      - Potatoes -> fries
      - *So, you check all ingredients once*
    - If you make Princess Chicken, you do:
      - Princess Chicken -> Youth Sauce, Chicken
        - Youth sauce -> Asparagus, Milk, Honey
      - *So, you check all ingredients TWICE*
    - If you make Princess Chicken and you have a Youth Sauce in the warehouse you do:
      - Princess Chicken -> Youth Sauce, Chicken
      - *So, you check all ingredients ONCE*
  - At this point you should see the pattern- every time you **make food,** or you **change the state of the warehouse** you should check if stuff did spoil or did not spoil. Every change you do makes one check.
- Chance to spoil
  - Configured in JSon;
    - “spoil rate”: 0.1 ← default
      - if “0”: food does not spoil
      - 0.1 means 0.1%.
  - It is applied for every **instance** of a basic ingredient in the warehouse.
    - If you have warehouse: {potatoes: 10, fish: 5} then **every** potato and fish have a 0.1% chance to spoil. So, you perform the check 10 times for potatoes, 5 times for fish.
    - If you have warehouse: {potatoes: 100} then you check 100 times for potatoes, each with 0.1% chance of spoilage.
  - Spoiled ingredient is removed from the warehouse and put to trash.
  - Only basic ingredients can be spoiled. Dishes cannot.

6.8.3. Trash

High level: every time you waste food or food got spoiled you move the food to the trash. Trash has a limit – max amount of space. If you exceed that space, you will poison the kitchen and restaurant will stop working.

Details:

- Every time we used to waste food in the past, we will now throw it into trash.
- Waste limit is configured in JSon:
  - "waste limit": 50 ← default
- Every dish / ingredient takes as much space as the amount of non-unique **basic ingredients** it requires:
  - Potato takes "1" of space because it is a basic ingredient
  - Fries take "1" of space, because:
    - Fries <- potato
    - One basic ingredient
  - Princess Chicken take "4" of space, because:
    - Princess Chicken <- Youth Sauce, *Chicken*
      - Youth Sauce <- *Asparagus, Milk, Honey*
    - 4 basic ingredients (chicken, asparagus, milk, honey)
  - Irish Fish take "3" of space, because:
    - Irish Fish <- Tuna, Fries, Smashed Potatoes
      - Fries <- potatoes
      - Smashed Potatoes <- potatoes
    - 3 basic ingredients (Tuna, Potatoes, Potatoes)
- If you ever exceed trash limit, the restaurant is Poisoned
  - For every command write "Restaurant Poisoned".
    - Like Bankruptcy, this makes the restaurant not able to do anything.
    - You cannot recover from Poisoning.

6.8.4. Take out the trash command

You have a new command:

- You want to throw trash away
- Syntax
  - Throw trash away
    - No parameters
- Result:
  - Whatever was in the trash → nothing there
    - Create a {file / database table} (your choice) named **waste_pool**
      - with information what has been thrown away
      - the exact text is left to you; I don't care. It can be "potatoes: 5, tuna: 12…"
      - there should be no duplicates in the **waste_pool**.
        - You call "Throw trash away" having 5 potatoes and 1 tuna in trash
          - Waste pool: {potatoes: 5, tuna: 1}
        - Later, you call "throw trash away" having 1 potato in trash
          - Waste pool: {potatoes: 6, tuna: 1}

6.9. Change 9: Tips and volatility of ordering

High level:

- Customers can leave tips if they can afford it and feel like it (randomness)
- Ordering has some volatility in terms of prices (randomness)

6.9.1. On tips without happiness

Every time customers pay something to the restaurant, they may give restaurant a tip.

- The tip amount is configured in JSon
    - There is 50% chance that the tip is 0.
    - There is 50% chance that the tip is between [1, 10] % of transaction.
        - So, Chicken Marmelade which costs 95 would have 10% tip at 9.5 (→ 10 for ease of calculating).
    - The JSon key is:
        - "max tip": 10 ← default
        - 10 represents the 10%
    - The lower bound (50% chance of no tip) is NOT configurable. There is 50% chance that a customer will not pay a tip. That's ok.
- If the tip would exceed the amount of money for the customer, the tip will be as high as possible so that customer has 0 money.
    - For example, Janet Mesmer (100 money) is buying Chicken Marmelade which costs her 95
        - (As in, costs X, then markup / profit margin → she pays 95).
    - Janet wants to pay a tip of 10%, but she has 100 money and 10% would require her to pay 105 (10% of 95 is 9.5, so -> 10).
    - Therefore, Janet will pay the maximum amount she can – tip of 5, so she is left with 0 money.
- Tips are **not taxed**. Neither they are taxed per transaction, nor daily. You want to show in **audit file** which money comes from tips, and which comes from other sources.

6.9.2. On orders and deliveries

Whenever we order anything using Order command, we will have to deal with volatility. You know, some suppliers are more expensive, some are less.

So, we have two new fields in our JSon:

- "order ingredient volatility": 10 ← default
- "order dish volatility": 25 ← default

So:

- If we call "order, potatoes, 100" we would:
    - previously pay 100 * (3 (potatoes) + 1 (taxes))
    - currently pay 100 * (3 (potatoes) + 1 (taxes)) * [0.9, 1.1 (volatility)]

So, volatility means just "instability of price". For ingredients It is +/- 10%, for dishes +/- 25%.

6.10. Change 10: Eating together and taxing everything

High level:

- Tips are not taxed? Come on. You know – taxes are inevitable. We are adding a new tax area to JSon.
- It is sad that people can't eat together using Table command because some people are poorer. So… let's make it possible for people to pool their money.
- Ah, throwing trash away? No way – we need to tax waste! You need to be eco-friendly, after all.

6.10.1. On tips and taxes.

You know the drill – we have a new area in JSon file:

"tips tax": 5 ← default

Tips and **only tips** are taxed using this tax. Once per day (it is a daily tax).

Of course, Audit is supposed to show every tax in existence, right?

6.10.2. Taxing your Trash

We have a new area in JSon file:

"waste tax": 15 ← default

This is a daily tax which taxes only the things which have been wasted and were thrown into trash.

If at any point waste tax exceeds 100, for every 100 exceeded pay extra 20 in environmental fines.

So:

- You have thrown into trash 1010 potatoes. Cost: 3*1000 = 3030.
    - Tax is 3030 * 0.15 = 454.5
        - Fine is applied 454/5 = 4 times.
        - Fine is 20 per fine
        - Therefore, total owned is: 454.5 + 20 * 4 = 535.

6.10.3. On pooling money at Table command

It is so sad, that Alexandra Smith (can eat everything and is rich) cannot take Bernard for a lunch at the restaurant, isn't it?

(Repetition of Customers from above, but with Alexandra Smith having a bit more money):

Regular customer	**Allergies**	**Budget**
Julie Mirage	Soy	100
Elon Carousel	Vinegar, olives	50
Adam Smith	-	100
Alexandra Smith	-	999999999999999999999999999999999999999
Barbara Smith	Chocolate	250
Christian Donnovan	Paprika	500
Bernard Unfortunate	Potatoes	15

So, let's assume **Julie Mirage**, **Alexandra Smith** and **Bernard Unfortunate** want to eat something.

The command Table is called like this (note "Pooled" flag):

*Table, **Pooled**, Julie Mirage, Alexandra Smith, Bernard Unfortunate, Emperor Chicken, Emperor Chicken, Emperor Chicken*

Now, I don't exactly remember who is allergic to what, but until now this operation would fail because neither Julie nor Bernard can afford Emperor Chicken.

What I want to happen here; let's assume they need to pay 700 for Emperor Chicken to make example simpler:

- They need to pay total for 3 Emperor Chickens:
  - 700 * 3 = 2100.
- They **try** to pay equally:
  - First, Bernard, Julie and Alexandra pay 15 each.
    - Bernard has 0, Julie has 85, Alexandra has 999…9984
  - Now Bernard can't pay. Julie pays 85, Alexandra pays 85.
    - Bernard has 0, Julie has 0, Alexandra has 999…9899
  - Until now they have paid: (15*3 + 85*2) = 215 out of 2100. So, Alexandra pays 1885.
    - Bernard has 0, Julie has 0, Alexandra has 999…8014
- If they were to pay a tip, Alexandra would pay all of it. Because she can afford it. Assuming 5% tip, she would pay extra 2100 * 0.05 = 105.

All previous rules about Table are unchanged. But if the **Pooled** flag is present in the Table command, people are paying together and use the sum of their budgets like in the example above.

So, the Table command syntax is now:

- Table, <FLAG::POOLED?><PERSON 1>, <PERSON 2>… <FOOD FOR PERSON 1>, <FOOD FOR PERSON 2>…
  - Where "Pooled" is an optional flag.
    - Table, Julie Mirage, Emperor Chicken ← valid not pooled
    - Table, Pooled, Julie Mirage, Emperor Chicken ← valid pooled

6.11. Change 11: Would you recommend something with onions to me?

6.11.1. I really want to eat something…

Have you ever wanted to eat sushi like, **really seriously?** To the point you wanted to go out and buy it while it was raining? 😉 This is usually called a **want** – you want to eat something and you want it now.

So now **every customer can have a want**. The **want** can be between [0, 3] basic ingredients (50% no want, 35% 1 want, 10% 2 wants, 5% 3 wants). The want can be one of the existing **basic ingredients**.

If a customer has a **want**, for every **want** which is fulfilled there will be a tip. The tip will be doubled for every want and the tip cannot be 0% (it is guaranteed to happen).

In case the customer does not have enough money to fulfill the tip, the customer will pay as much as possible.

Example:

- Julie Mirage is allergic to Soy and has a Budget of 500.
- Julie Mirage checks for a Want. She has a Want for Chicken, Onions, Asparagus.
- Julie Mirage orders a Princess Chicken (basic ingredients: Chicken, Asparagus, Milk, Honey) using Buy command.
- Because she has any Want fulfilled, she is guaranteed to tip more than 0%. The tip will happen.
    - Say, she generated base tip of 4%
- The tip will be quadrupled, because she has 2 Wants fulfilled (and every Want doubles the amount of tip).
    - So, she will pay a tip of 4*2*2 = 16% of food.

6.11.2. … so can you recommend something to me?

This is simple to explain.

Both **Buy** command and **Table** command have a new optional flag – Recommend.

In case of Buy:

- *Buy, Julie Mirage, Recommend, Chicken*
- Buy, Person, Recommend (optional flag), Basic Ingredients to recommend

In case of Table the syntax gets way more difficult:

- *Table, (Alexandra Smith, Adam Smith), (Irish Fish), (Recommend, Potatoes, Chicken)*
    - Alexandra Smith wants an Irish Fish
    - Adam Smith wants something recommended with Potatoes and Chicken
- Table, (Pooled), *(Alexandra Smith, Adam Smith), (Irish Fish), (Recommend, Potatoes, Chicken)*
    - Alexandra Smith wants an Irish Fish
    - Adam Smith wants something recommended with Potatoes and Chicken
    - They are pooling the money

What happens when we call the Buy / Table with a flag Recommend:

- From the list of all the **dishes** we select the dish which has the following conditions:
    - The customer can afford it
        - It is the most expensive dish the customer can afford
    - We have ingredients to make it, or it is in the warehouse
    - The customer has no allergies for any ingredient in the dish
    - If it is **Pooled**, the table can afford it; not necessarily this customer

Example:

We have **Monica Mesmer** (allergic to **Vinegar**) and having budget of 300. She calls:

- *Buy, Monica Mesmer, Recommend, Tuna*

We have the following dishes containing Tuna (**bolded**), from which some contain Vinegar (*italics*). For the purpose of the example, I will write some random cost on the right side in red.

Food	**Ingredients**	
Emperor Chicken	Fat Cat Chicken, Spicy Sauce, **Tuna Cake**	**Total Cost: 500**
Fat Cat Chicken	Princess Chicken, Youth Sauce, Fries, Diamond Salad	
Princess Chicken	Chicken, Youth Sauce	
Youth Sauce	Asparagus, Milk, Honey	
Spicy Sauce	Paprika, Garlic, Water	
Omega Sauce	Lemon, Water	
Diamond Salad	Tomatoes, Pickles, Feta	
Ruby Salad	Tomatoes, Vinegar, Chocolate	
Fries	Potatoes	
Smashed Potatoes	Potatoes	
Tuna Cake	**Tuna**, Chocolate, Youth Sauce	**Total Cost: 200**
Fish In Water	**Tuna**, Omega Sauce, *Ruby Salad*	**Total Cost: 250**
Irish Fish	**Tuna**, Fries, Smashed Potatoes	**Total Cost: 150**

So, our recommendation engine:

- Sees **Emperor Chicken** as the most profitable one. But cannot recommend that one – it exceeds Monica's budget.
- Sees **Fish in Water** as the next most profitable one. But cannot recommend that one – it has *vinegar* in its dependency, and she is allergic to *vinegar*.
- Sees **Tuna Cake** as the next most profitable one. And it recommends Tuna Cake to Monica, as all criteria are met (stuff in the warehouse, no allergies, can afford it)

And the record:

- *Buy, Monica Mesmer, Recommend, Tuna*

Is executed as if it was a record:

- *Buy, Monica Mesmer, Tuna Cake*

6.12. Change 12: Business sure loves their powerpoint charts…

6.12.1. Expansion of Auditing

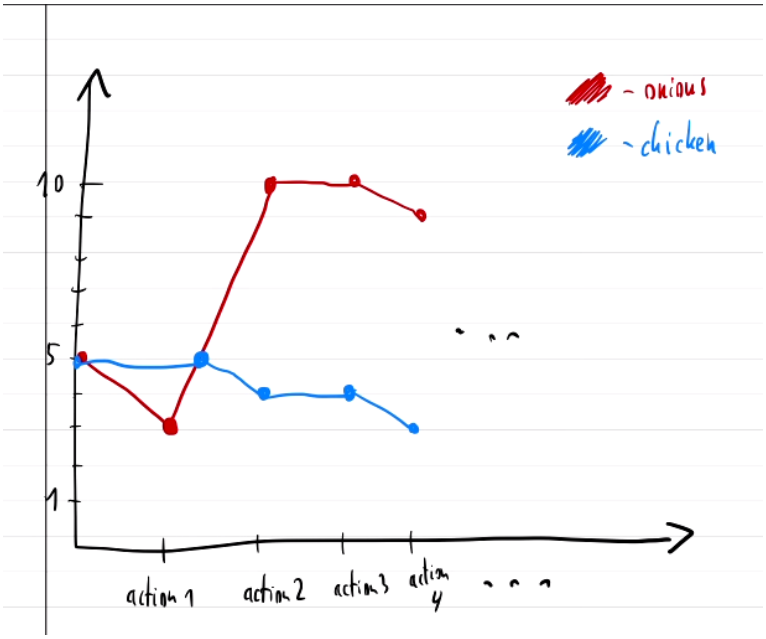
I want to see some statistics. Add to an audit file the following statistics:

- What were the three most popular **basic ingredients**? (Most often used)
  - How many times were those ingredients used?
- What was the dish which was **the most profitable**? (Most profit gained from it)
  - How many times was the most profitable dish bought (no matter if Buy or Table command)?
- How many dishes were bought during this day, total (no matter if Buy or Table command)?
- What was the **most recommended** dish? (Most often recommended)

The exact format is not important. I do want to be able to see an audit file with such a report.

6.12.2. Charts are cool

I want to see two charts showing the changes in the warehouse. Something like this:



The chart should show only **basic ingredients** and only those **basic ingredients** which had more than 20% changes during the day (single operation of the program).

So, if an ingredient changed only once, during one action (say, something spoiled) and there were 5 actions, it should be present on the chart. If there were 6 actions, it should not be present on the chart.

This 20% should be configured in the JSon, under the key:

- "chart visibility": 20 ← default
  - if "0": do not show anything on a chart and do not generate the chart file
  - 20 means 20%.

You will have to research the plotting / drawing library yourself.