

Clean Code Projects

- [WizBakTop](#)
- [Args Parser](#)
- [OCR](#)
- [Oh My Sandwich](#)

WizBakTop

[GitHub repo](#)

Créer un outil qui permette de générer le WizBakTop pour un nombre donné en ligne de commande.

Partie I

Les règles du WizBakTop sont les suivantes :

- Si le nombre est divisible par 3, écrire 'Wiz' à la place du nombre
- Si le nombre est divisible par 5, ajouter 'Bak'
- Si le nombre est divisible par 7, ajouter 'Top'
- Pour chaque digit 3, 5, 7, ajouter 'Wiz', 'Bak' et 'Top' dans l'ordre des digits

Exemples :

```
my-pc$: wizbaktop 1
```

```
my-pc$: 1
```

```
my-pc$: wizbaktop 2
```

```
my-pc$: 2
```

```
my-pc$: wizbaktop 3
```

```
my-pc$: WizWiz (divisible par 3, contient 3)
```

```
my-pc$: wizbaktop 5
```

```
my-pc$: BakBak (divisible par 5, contient 5)
```

```
my-pc$: wizbaktop 6
```

```
my-pc$: Wiz (divisible par 3)
```

```
my-pc$: wizbaktop 15
```

```
my-pc$: WizBakBak (divisible par 3 et 5, contient 5)
```

my-pc\$: wizbaktop 33

my-pc\$: WizWizWiz (divisible par 3, contient 3 et 3)

my-pc\$: wizbaktop 35

my-pc\$: BakTopWizBak (divisible par 5 et 7, contient 3 et 5)

my-pc\$: wizbaktop 357

my-pc\$: WizTopWizBakTop (divisible par 3 et 7, contient 3, 5 et 7 dans cet ordre)

my-pc\$: wizbaktop 703

my-pc\$: TopWiz (contient 7 et 3 dans cet ordre)

my-pc\$: wizbaktop 13705

my-pc\$: BakWizTopBak (divisible par 5, contient 3, 7 et 5 dans cet ordre)

Partie II

Vous devez maintenant garder une trace des 0 contenus dans les nombres traités. Pour cela, chaque 0 doit être remplacé par le caractère '*'.

Exemples précédents avec la nouvelle règle :

my-pc\$: wizbaktop 703

my-pc\$: Top*Wiz (contient 7, 0 et 3 dans cet ordre)

my-pc\$: wizbaktop 13705

my-pc\$: BakWizTop*Bak (divisible par 5, contient 3, 7, 0 et 5 dans cet ordre)

Args Parser

GitHub repo

Nous avons tous plus ou moins déjà été amenés à parser des arguments en entrée d'un programme.

L'objectif de ce TP est de fournir un utilitaire sous forme de classe, permettant de faire ce travail à notre place.

Les arguments passés au programme consistent en des **flags** et des **values**.

- Un flag contiendra toujours un signe moins (-) suivi d'un caractère (**[a-zA-Z]**).
- Un flag doit avoir **zero** ou **une value** associée.

Vous devez écrire un **parser** pour ce genre d'arguments.

Ce parser doit prendre un **schéma** détaillant les arguments auxquels le programme peut s'attendre.

Le schéma spécifie **précisément** les flags possibles ainsi que le **type associé**.

Une fois le schéma spécifié, l'utilisateur doit pouvoir **passer une liste d'arguments** au parser.

Le parser se charge alors de **vérifier** qu'il peut parser toutes les valeurs (selon le schéma spécifié). Si c'est OK, l'utilisateur doit alors pouvoir **récupérer** les valeurs du parser en utilisant les **noms** des flags.

Si l'utilisateur essaie de récupérer la valeur pour un **flag inexistant**, une valeur doit être renvoyée par défaut

- false pour un boolean
- 0 pour un number
- "" (chaîne vide) pour une string

Un exemple d'utilisation est donné en Typescript :

```
// Schema explanations:
// -> char   - Boolean arg
// -> char#   - Number arg
// -> char*   - String arg

const schema = 'd,p#,h*';
try {
  const args = new Args(schema);
```

```
args.parse(`-d -p 42 -h 'Vincent Vega'`);

const detach = args.getBoolean('d');
const port = args.getNumber('p');
const hero = args.getString('h');
executeApplication(detach, port, hero);
} catch (e) {
  console.error(`Parse error: ${e.message}`);
}

const executeApplication = (d, p, h) => {
  console.log(`Application running - detached (${d}), port: (${p}), hero is (${h})`);
}
```

OCR

[GitHub repo](#)

Subject

User Story 1

The input format created by the machine is as follows:

```
-- -- -- --  
|_|_|_|_|_|_|_|_|  
||_|_|_|_|_|_|_|  
|_|_|_|_|_|_|_|_|
```

Each entry is exactly **4 rows and 27 columns** (9 x 3). The first three lines describe numbers using pipes and underscores. The fourth line is blank.

Each entry or **code** created has 9 digits, each ranging from 0 to 9. A typical file can contain up to 100 entries.

Write a program that takes this file as input and manages to parse the codes contained.

User Story 2

Sometimes the machine generates wrong codes. You should now be able to validate the codes using a checksum. It can be calculated as follows:

code : 3 5 6 6 0 9 7 0 1 position : p9 p8 p7 p6 p5 p4 p3 p2 p1

checksum computing : $((1p1) + (2p2) + (3p3) + \dots + (9p9)) \bmod 11 == 0$

User Story 3

Your manager wants the results of your program. It asks you to write an output file, for each of the input files, on this format :

457508000 664371495 ERR

The output file has one code per line. If the checksum is bad, it is indicated by ERR in a second column indicating the status.

User Story 4

Sometimes the machine produces unreadable numbers, such as the following :

```
--  -----  
|_| | | |_| | ||||  
||_| | |_| || |_|_|
```

Your program should be able to spot such problems. In this case, the unknown numbers are replaced by '?'. Update your file output. With the previous unreadable number, this would give :

```
457508000  
664371495 ERR  
12?13678? ILL
```

User Story 5

Your manager would like to do some classification. For a set of files given as input, he would now like to have the possibility of:

- Either keep the current behavior and create an output file for each input file
- Or use a new behavior that allows it to "group" similar codes

This behavior is as follows: Regardless of the number of input files, the program will create 3 outputs named authorized, errored, and unknown

Authorized contains all valid checksums Errored contains all invalid checksums Unknown contains all unreadable checksums

User Story 6

Provide a command tool to other developers in your company, so they can easily use all the features you just created.

Its implementation is free.

Implementation

Usage

You can run few commands to start/build this app :

- `npm run start` : Run compiled app in **dist** folder.
- `npm run build` : Compile the app and generate **dist** folder
- `npm run build-dev` : Compile the app and generate **dist** folder then run it (build + start)
- `npm run dev` : Directly run the TS source project
- `npm run test` : Run cucumber tests. This generates two report :
 - One in **coverage** folder which show the test coverage
 - One in **cucumber_report.html** at the root of the project that show how cucumber tests results
- `npm run lint` : Run `Eslint` on source code

Run command, arguments

When running the program, there is a few (optional) arguments that we can use :

- **h** (optional): **boolean** => display help
- **s** (optional, default=false): **boolean** => split classifier into multiple files
- **m** (optional, default=100): **number** => set the max number of lines to process
- **l** (optional, default=9): **number** => number of digits per lines
- **i** (optional, default='input.txt'): **string** => input filename
- **v** (optional): **string** => valid output filename
- **e** (optional): **string** => error output filename
- **u** (optional): **string** => unreadable output filename

Classes

The main class is probably the one that can parse each line, regarding the given schema :

```
interface Parser {  
    parseText(text: string, maxLines: number, lineSize: number): string[];  
  
    parseLine(line: string, length: number): string;  
}
```

This class consists in splitting each character, and it uses another class to get the parsed char :


```
interface CharParser {
    parseChar(input: string): string;
}
```

To be able to match the digit correctly, we are using a string map :

```
const defaultDigitMap: Map<string, string> = new Map([
    ['_\\n|\\n|_', '0'],
    ['\\n \\n |', '1'],
    ['_\\n _\\n|_', '2'],
    ['_\\n _\\n _|', '3'],
    ['\\n|_\\n |', '4'],
    ['_\\n|_\\n _|', '5'],
    ['_\\n|_\\n|_|', '6'],
    ['_\\n |\\n |', '7'],
    ['_\\n|_\\n|_|', '8'],
    ['_\\n|_\\n _|', '9'],
]);
```

To allow the user to choose the way the outputs are classified, we use the following class to return a said output destination for a given state :

```
export interface Classifier {
    getDestination(lineState: LineState): string;
}
```

The user can choose between two predefined classifiers, the default one is the unified classifier :

```
export const splitClassifierStateAssociation: Map<LineState, string> = new Map([
    [LineState.VALID, 'authorized.txt'],
    [LineState.ERROR, 'errored.txt'],
    [LineState.UNREADABLE, 'unknown.txt'],
]);

export const unifiedClassifierStateAssociation: Map<LineState, string> =
    new Map([
        [LineState.VALID, 'output.txt'],
        [LineState.ERROR, 'output.txt'],
        [LineState.UNREADABLE, 'output.txt'],
    ]);
```

Tests & Coverage

When we run the cucumber tests, it generates two reports

Coverage Report

The first one is a code coverage on cucumber's tests. This file is viewable on **coverage/index.html** and look like this :

image-20220301120104082.png

image-20220301120202609.png

Cucumber report

The other report is the result of the ran cucumber's tests :

image-20220301120254863.png

image-20220301120448078.png

Oh My Sandwich

Dépôt GitHub : <https://github.com/Nouuu/OhMySandwich>

Énoncé

1. Informations pratiques

Le code rendu devra être compilé sans erreurs.

Il vaut mieux rendre un code incomplet qui compile qu'un code ne compile pas.

Le projet sera noté selon plusieurs critères :

- Qualité du code fourni
- Qualité des présentations intermédiaires
- Prise en compte des différents retours suite aux présentations
- Qualité de la soutenance finale

Vous n'oublierez pas d'inclure les slides de votre soutenance finale ainsi qu'un rapport PDF précisant vos choix, les problèmes techniques rencontrés et les solutions trouvées.

2. Sujet

Une sandwicherie souhaite simplifier sa prise de commande et l'élaboration de ses factures.

Chaque sandwich est constitué d'une liste précisé d'ingrédients et possède un prix.

On souhaite écrire un programme qui prend en entrée une commande de sandwiches et produit une facture formatée.

Prise en compte des commandes

Votre programme devra récupérer les commandes sous la forme d'entrée textuelle en console.

Les commandes sont de la forme :

- A Sandwich1

- A Sandwich1, B Sandwich2, C Sandwich3

Une commande peut contenir plusieurs occurrences du même sandwich, ainsi une commande de la forme :

- A Sandwich1, B Sandwich2, C Sandwich1 devra être considérée comme :
- A+C Sandwich1, B Sandwich2

Édition d'une facture

Après avoir interprété la commande en entrée, vous produirez une sortie console suivant la forme suivante :

```
A Sandwich1
  Ingredient1
  Ingredient2
  [...]
  IngredientN
B Sandwich2
  Ingredient1
  [...]
[...]
Prix total : XXX€
```

Sandwichs disponibles

La sandwicherie est capable de produire les sandwichs suivants :

- **Jambon beurre** : 1 pain, 1 tranche de jambon, 10g de beurre => 3,50€
- **Poulet crudités** : 1 pain, 1 oeuf, 0.5 tomate, 1 tranche de poulet, 10g de mayonnaise, 10g de salade => 5€
- **Dieppois** : 1 pain, 50g de thon, 0.5 tomate, 10g de mayonnaise, 10g de salade => 4,50€

Comportement attendu du programme

Votre programme devra récupérer l'entrée de l'utilisateur et valider sa conformité.

En cas de commande incorrecte, votre programme produira une erreur compréhensible, mais ne devra pas crasher.

En cas de commande correcte, votre programme écrira dans la console la facture.

Après avoir traité une commande, votre programme attendra la commande suivante, il ne doit pas s'arrêter après avoir écrit une facture.

3. Déroulement du projet

Il vous sera demandé une première implémentation naïve ne vous demandant pas d'utiliser de design patterns.

Cette première implémentation sera présentée et servira de base aux discussions de pistes d'améliorations de votre projet.

Vous devrez ensuite revoir votre implémentation afin d'y implémenter des designs pattern appropriés.

Cette deuxième implémentation vous servira de base pour les modules complémentaires du projet.

Une fois la première implémentation présentée, les différents modules complémentaires vous seront présentées.

Votre rendu final devra contenir le sujet de base ainsi qu'au moins un module complémentaire.

Vous devrez présenter une première fois votre implémentation ainsi que vos choix.

Choix d'implémentation

Solution & Projets

Afin d'avoir une architecture propre, nous avons créé une solution contenant plusieurs projets. Ils sont au nombre de 4 :

- **CLI** : Implémentation d'un Adaptateur de Commande en ligne de commande pour l'application.
- **Domain** : Contient la partie métier, nos objets du domaine ainsi que nos interfaces
- **Infrastructure** : Contient les objets techniques nécessaires à l'implémentation de l'application
- **Test** : Contient les tests unitaires de l'application

Tests

Nous avons créé un projet de tests unitaires afin de pouvoir nous assurer que notre code fonctionne correctement.

Cela nous a aussi été utile au moment du refactoring de notre code pour ne pas faire de regression.

Design Patterns

Command

```
public interface ICommand
{
    ICommand? Execute();

    string GetCommandHelp();

    void Display();
}
```

CLI

- InvoiceGeneratorCommand
- MenuCommand
- NewOrderCommand
- SandwichSelectorComman

Adapter

```
public interface IAdapter
{
    void AcceptInteractions();
}
```

```
public interface IMarshaller<in T>
{
    public string Serialize(T data);
}
```

CLI

- CliAdapter

Infrastructure

- ConsoleBasketMarshaller

- ConsoleIngredientMarshaller
- ConsoleInvoiceMarshaller
- ConsolePriceMarshaller
- ConsoleSandwichMarshaller

Facade + Singleton

```
public interface Context
{
    InvoiceGenerator GetInvoiceGenerator();
    IMarshaller<IngredientStack> GetIngredientMarshaller();
    IMarshaller<Invoice> GetInvoiceMarshaller();
    IMarshaller<Price> GetPriceMarshaller();
    IMarshaller<Sandwich> GetSandwichMarshaller();
    IMarshaller<Basket> GetBasketMarshaller();
    Basket GetBasket();
    List<ICommand> GetAvailableCommands();
    List<Sandwich> GetAvailableSandwichs();
    IAdapter GetAdapter();
}
```

Infrastructure

- CliContext

Factory

```
public interface InvoiceGenerator
{
    Invoice GenerateInvoice(Basket basket);
}
```

Infrastructure

- DefaultInvoiceGenerator

Iterator

```
public interface Iterator<out T>
{
    T NextIteration();
}
```

Infrastructure

- AlphabetIterator

Builder

“ SandwichBuilder

```
public class SandwichBuilder
{
    private readonly string? _name;
    private readonly Price? _price;
    private readonly ISet<IngredientStack> _ingredients;

    public SandwichBuilder()
    {
        _ingredients = new HashSet<IngredientStack>();
    }

    public SandwichBuilder(string? name, Price? price, ISet<IngredientStack> ingredientStacks)
    {
        this._name = name;
        this._price = price;
        _ingredients = ingredientStacks;
    }

    public Sandwich GetSandwich()
    {
        if (_name == null || _price == null)
        {
            throw new InvalidOperationException();
        }

        return new Sandwich(_name, _ingredients.ToArray(), _price.Value);
    }

    public SandwichBuilder SetName(string newName)
    {
        return new SandwichBuilder(newName, _price, _ingredients);
    }
}
```



```

}

public SandwichBuilder AddIngredient(IngredientStack ingredientStack)
{
    var newIngredients = new HashSet<IngredientStack>(_ingredients) { ingredientStack };
    return new SandwichBuilder(_name, _price, newIngredients);
}

public SandwichBuilder AddIngredient(Ingredient ingredient, double count)
{
    var newIngredients = new HashSet<IngredientStack>(_ingredients) { new(ingredient, count) };
    return new SandwichBuilder(_name, _price, newIngredients);
}

public SandwichBuilder SetPrice(Price price)
{
    return new SandwichBuilder(_name, price, _ingredients);
}

public SandwichBuilder SetPrice(double price)
{
    return new SandwichBuilder(_name, new Price("€", price), _ingredients);
}

public SandwichBuilder FromSandwich(Sandwich sandwich)
{
    return new SandwichBuilder(
        sandwich.Name,
        sandwich.Price,
        new HashSet<IngredientStack>(sandwich.Ingredients)
    );
}
}

```

Value Object

Domain

- Basket
- Ingredient
- IngredientStack
- Invoice

- Price
- Sandwich
- UnitType