

# COMPLÈMENT INFO 2A, CM1

Pépin Rémi, Ensai, 2020

remi.pepin@ensai.fr



1 / 60

## GÉNÉRALITÉS

2 / 60

### BUT DU COURS

**Vous apporter les connaissances nécessaires pour mener à bien un projet d'application de traitement de données.**

- Comprendre les développeurs
- Conduire un projet (informatique)
- Dépasser les 50 lignes de code
- Travailler à plusieurs

3 / 60

### LE PROGRAMME

1. Analyse fonctionnelle, génie logiciel
2. Programmation orientée objet avancée
3. Communication avec une base de données en python
4. Sécurité informatique
5. Communication client-serveur
6. Le versionnage avec git

4 / 60

### LE PLAN D'AUJOURD'HUI

1. L'analyse fonctionnelle
  1. Définition
  2. Diagrammes UML
2. L'architecture logiciel
  1. Définition
  2. Séparation des responsabilités
3. Notion avancé de POO
  1. Rappels
  2. Classes abstraites
  3. Bridge pattern
4. Génie logiciel
  1. Définition
  2. Single responsibility principle
  3. Design patterns

5 / 60

## ANALYSE FONCTIONNELLE

6 / 60

## C'EST QUOI L'ANALYSE FONCTIONNELLE ?

- Première étape de tous les projets
- Détermine les **fonctions**, **acteurs** du produit pour répondre aux **besoins** du client
- Diagrammes pour échanger avec le client
- Priorise les travaux

7 / 60

## LES QUESTIONS À SE POSER

- Quels sont les types d'utilisateur qui vont utiliser mon application (administrateur, gestionnaire, client etc) ?
- Quelles sont les fonctionnalités ? Fonctionnalités communes entre les profils ?
- Comment fonctionne les processus de l'application ?

Quels sont les diagrammes à utiliser ?

8 / 60

## LES DIAGRAMMES UML

- Cours de 1A
- UML 2.5, Pascal Roques, Eyrolles, Mémento (bibliothèque Ensai)

9 / 60

10 / 60

## ARCHITECTURE LOGICIELLE

### C'EST QUOI L'ARCHITECTURE LOGICIELLE?

- Le pendant technique de l'analyse fonctionnelle
- Maintenant que l'on a le **qui et quoi**, on détermine le **comment**
- On dessine le code de notre application
- Vision macro de notre application (agencement des grandes pièces)

11 / 60

12 / 60

## POURQUOI C'EST IMPORTANT : PARALLÈLE AVEC L'ARCHITECTURE

## POURQUOI C'EST IMPORTANT : PARALLÈLE AVEC L'ARCHITECTURE

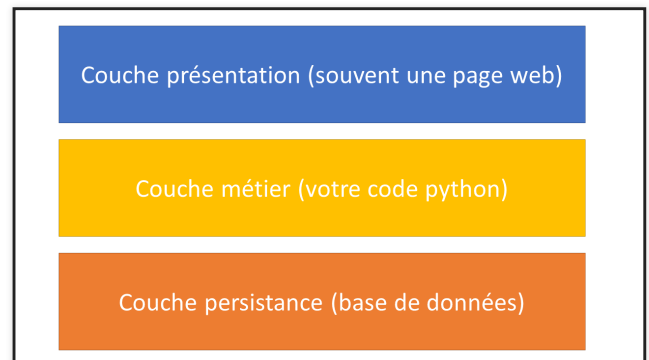
- Pièces, l'installation électrique, l'eau, le gaz, contraintes législatives, s'adapter au terrain ...
- Besoin de réfléchir comment il faut agencer tout ça dès le début
- Si on construit au fil de l'eau on risque d'avoir une maison incohérente (au mieux)
- **Ce n'est pas du temps perdu !**

13 / 60

14 / 60

## UN GRAND PRINCIPE : SEPARATION OF CONCERNS

## UN GRAND PRINCIPE : SÉPARATION DES RESPONSABILITÉS



15 / 60

16 / 60

## LES PRINCIPALES COUCHES D'UNE APPLICATION

- Présentation : tout ce qui se charge de l'affichage (page web, console, fenêtre)
- Métier : c'est le métier de votre application, sa **plu value**
- Persistance : gère la persistance des données. Base de données ou système de fichiers

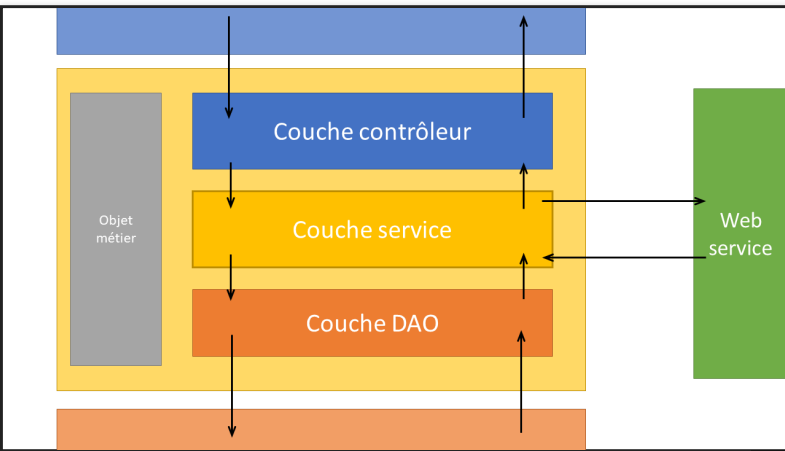
## POUR VOTRE PROJET

- Présentation : terminal
- Métier : votre code python
- Persistance : base de donnée

17 / 60

18 / 60

## ZOOM SUR LA COUCHE MÉTIER



19 / 60

## LES COUCHES DE LA COUCHE MÉTIER 1/2

- DAO : *Data access object* c'est la partie de votre code qui communique avec la base de données (TP2)
- Service :
  - code métier
  - manipule des objets métiers pour créer de l'information ou de la valeur
  - Demande des objets à la couche DAO
  - Appelle les webservice externe

20 / 60

## LES COUCHES DE LA COUCHE MÉTIER 2/2

- Objet métier :
  - couche transverse
  - représente des concepts métier que votre code va manipuler
  - peu de comportements
- Contrôleur :
  - récupère les inputs des utilisateurs
  - renvoie les données à afficher
  - TP 4

21 / 60

## POURQUOI SÉPARER EN COUCHE ?

- Travail en groupe 🧑🧑🧑🧑
- Lisibilité du code 📖
- Débogages 🐛

Limiter les risques d'erreur quand on modifie le code (éviter l'assiette de spaghetti) 🍝

22 / 60

## INFORMATIONS À RETENIR

- Passer du temps à réfléchir aux différents modules d'une application n'est pas une perte de temps 🧠
- Diviser en couche permet de travailler en parallèle 🛠️🍳
- Mais il faut encore réfléchir comment bien coder 🐱

23 / 60

24 / 60

# PROGRAMMATION ORIENTÉE OBJET AVANCÉE

## QUELQUES RAPPELS

Les trois principes de l'objet :

- **Encapsulation** : un objet va contenir des **attributs** et des **méthodes**
- **Héritage** : un objet peut **hériter** des attributs et méthodes d'une autres classes pour les **redéfinir**. Il va pouvoir également ajouter d'autres attributs/méthodes
- **Polymorphisme** : une méthode peut être associée à un **code différent** en fonction des paramètres passés ou de l'objet à qui elle appartient

25 / 60

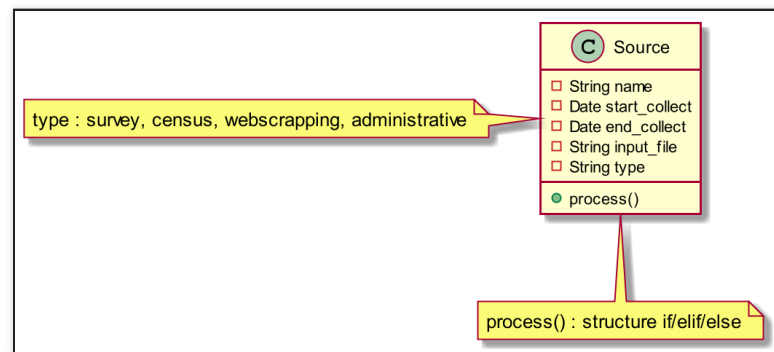
26 / 60

## UN EXEMPLE POUR ILLUSTRER TOUS ÇA

Application de traitement automatique des données :

- Plusieurs sources de données : enquêtes, webscrapping, fichiers administratifs, ...
- Plusieurs formats de données : csv, xml, json, ... (*on en reparlera*)
- Plusieurs algo de traitements : stat exploratoire, regression, "machine learning", ...

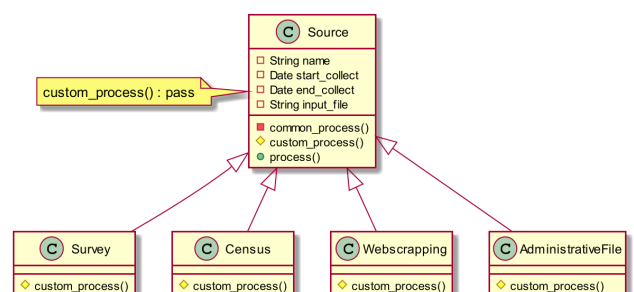
## EXEMPLE DE DIAGRAMME DE CLASSE



27 / 60

28 / 60

## UN EXEMPLE AVEC DE L'HÉRITAGE



29 / 60

30 / 60

## LES CLASSES ABSTRAITES

**Classe Abstraite** : classe dont l'implémentation n'est **pas complète** et qui n'est **pas instantiable**. Permet de passer un **contrat**, les classes filles vont devoir implémenter ce qui manque.

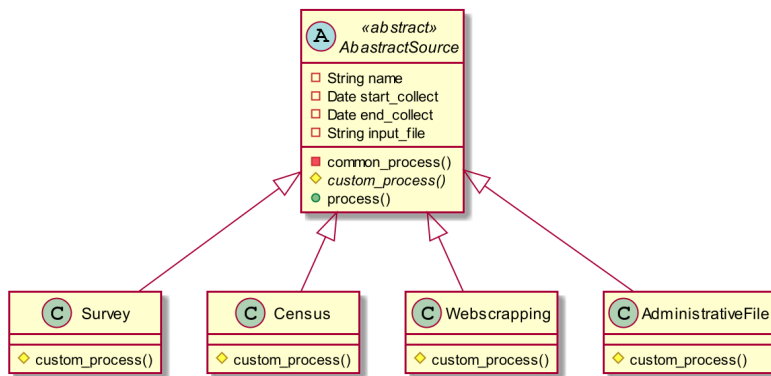
Avantages :

- On sait ce que toutes les classes filles doivent faire 👍
- On peut générer du code 🙏
- Limitent le risque d'erreurs !! 🧐

31 / 60

32 / 60

### PAR EXEMPLE



33 / 60

### ET EN PYTHON ?

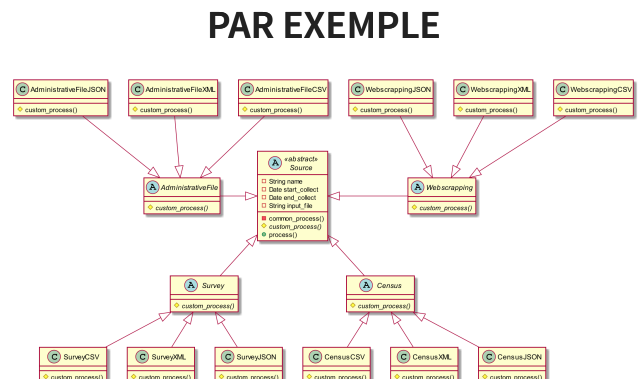
- Pas de gestion native des classes abstraites 🙅
- Module Abstract Base Classes (abc) pour résoudre le problème 😊
  - Step 1 - Télécharger le module
  - Step 2 - Importer le module
  - Step 3 - Hériter de ABC
  - Step 4 - Définir les méthodes abstraites
  - Step 5 - ???
  - Step 6 - Profit

34 / 60

### ET SI ON AJOUTAIT LES FORMATS DE DONNÉES ?

Actuellement 3 formats de données dans notre application :

- CSV : Comma Separated Values (tabulaire)
- XML : eXtended Markup Language (format à balise)
- Json : JavaScript Object Notation (format clef-valeur)



Voyez-vous un problème ?

35 / 60

36 / 60

## LA PUISSANCE DE LA POO

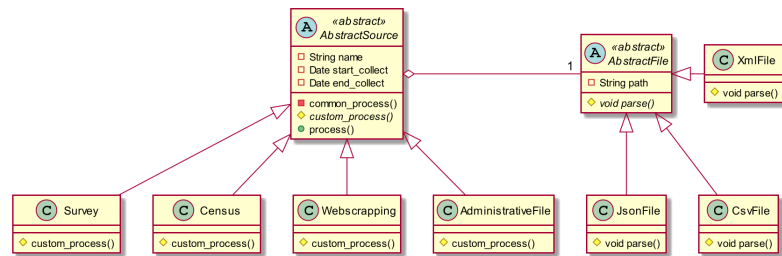
- Actuellement 4 \* 3 classes "concrètes" à définir 🙄
- La lecture du format de fichier est dépendante de la source 🤖



MAIS 🤖

- On peut externaliser ce traitement ! 😊
- Relation d'agrégation 🤖

## LE BRIDGE PATTERN



37 / 60

38 / 60

## WORK SMART, NOT HARD

- Composition + héritage : 9 classes 😊
- Héritage : 17 classes 🤖
- On peut facilement ajouter des types et des formats 🤖

**Pattern Bridge 🏗️** : découpage d'une grosse classe en un groupe de petites classes avec leur propre hiérarchie qu'il faut ensuite assembler.

## POUR RÉSUMER

- Utiliser la puissance de la POO 💣
- Préférer les objets spécifiques (héritage) au if/elif/else 🤖
- Les classes abstraites sont des plans pour les futures classes 🤖
- La POO permet de créer des codes plus lisibles, évolutifs et maintenables 🏰

39 / 60

40 / 60

## LE GÉNIE LOGICIEL

41 / 60

42 / 60

## C'EST QUOI LE GÉNIE LOGICIEL ?

- Un constat : coder bêtement ne permet pas de faire application de qualité
- Mais empiler des briques bêtement ne permet pas de faire une maison même si on a un plan
- Besoin de planifier, de documenter, de tester etc

43 / 60

## POURQUOI C'EST IMPORTANT : PARALLÈLE LA CONSTRUCTION D'UNE MAISON

44 / 60

## POURQUOI C'EST IMPORTANT : PARALLÈLE LA CONSTRUCTION D'UNE MAISON

- Vous avez le plan de construction d'une maison (fourni par l'architecte)
- Mais implémenter ce plan demande des connaissances techniques
- Besoin refaire des schémas pour des zones précises (arches, escaliers ...)
- **Ce n'est pas du temps perdu !**

Faire du code de qualité c'est comme faire de l'artisanat de précision, cela demande outils, expérience et méthodes.

45 / 60

## QUELQUES PRINCIPES DE BASE

- Décomposition d'un programme en modules simples **cohérents**
- Les modules **exposent** des méthodes utilisables / surchargeable par d'autres modules mais restent protégés aux modifications non prévues
- Chaque module doit être une **boîte noire** pour les autres
- Si on garde les mêmes **entrées/sorties** on peut changer un module sans risque
- Privilégier abstractions + héritage que if/then/else

46 / 60




## UN MANTRA


### Faible couplage, forte cohérence

- **Faible couplage inter-classes** : modifier une classe doit impacter les autres le moins possible
- **Forte cohérence intra-classe** : chaque classe doit être un ensemble cohérent d'attributs et méthodes

47 / 60

## FAIBLE COUPLAGE, FORTE COHÉRENCE : POURQUOI LE RESPECTER ?

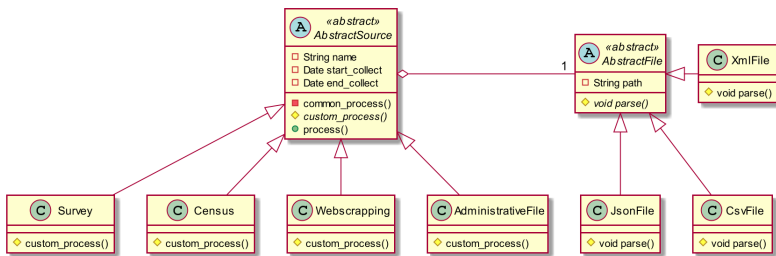
- Travail en groupe 
- Lisibilité du code 
- Débogages 

Limiter les risques d'erreur quand on modifie le code (éviter l'assiette de spaghetti) 

48 / 60



## RETOUR SUR LE BRIDGE PATTERN



- La partie "Source" gère les traitements liés à la source
- La partie "Fichier" gère la lecture de fichier
- Seules les entrées/sorties comptent
- On peut ajouter une partie "Traitement" pour des traitements supplémentaires
- Pas de if/elif/else inutiles

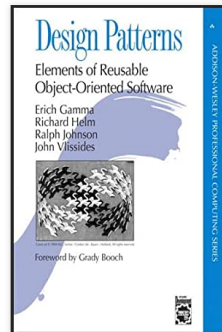
**Chaque partie de notre code s'occupe d'une seule chose**

49 / 60

50 / 60

## LES DESIGN PATTERNS : HISTOIRE

- 1994 : *Design Patterns – Elements of Reusable Object-Oriented Software* du Gang of Four
- 23 patrons au début
- Aujourd'hui environ 40



51 / 60

## LES DESIGN PATTERNS : DÉFINITION

*“En informatique, et plus particulièrement en développement logiciel, un patron de conception (souvent appelé design pattern) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.”*

52 / 60

## LES DESIGN PATTERNS : IN A NUTSHELL

- Bonnes pratiques
- Solutions standards à des problèmes de conception
- Solutions robustes
- Indépendants de la technologie
- Indépendants du métier

## LES DESIGN PATTERNS : EXEMPLE

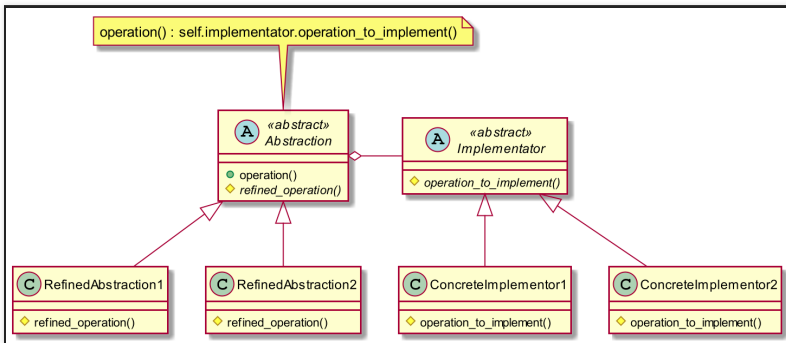
**Problème récurrent :**

- Créer des objets complexes qui sont une composition de caractéristiques indépendantes
- Dit autrement : découpler l'abstraction de son implémentation pour qu'elles puissent varier indépendamment

53 / 60

54 / 60

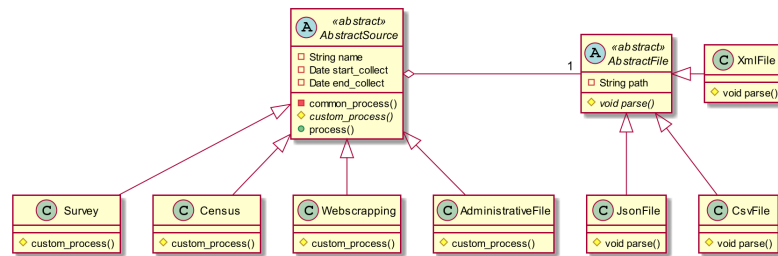
## LES DESIGN PATTERNS : EXEMPLE



C'est la forme "pure" du bridge pattern

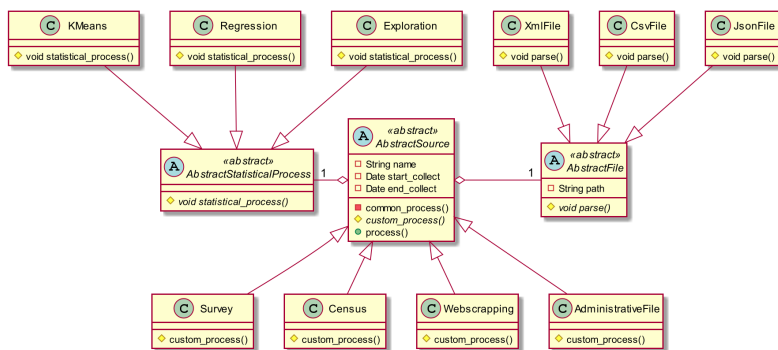
55 / 60

## LES DESIGN PATTERNS : EXEMPLE



56 / 60

## LES DESIGN PATTERNS : EXEMPLE



57 / 60

## POUR RÉSUMER

- Faire une application complexe demande un code complexe 🧩
- Sans phase de conception on va dans le mur 🏠
- Il existe des solutions prêtes à l'emploi à des problèmes courants 📦

Faible couplage, forte cohérence

58 / 60

## LES DESIGN PATTERNS : RESSOURCES UTILES

- Design patterns : catalogue de modèles de conception réutilisables / Erich Gamma ; Richard Helm ; Ralph Johnson ; John Vlissides ; Jean-Marie Lasvergères
- [Refactoring guru](#)
- [Python Patters github](#)

THAT'S ALL FOLKS

59 / 60

60 / 60