

Développez un programme logiciel en Python

OpenClassRooms - Parcours Python - Projet 4

Bérenger Ossété Gombé

2 juin 2022

Sommaire

- 1 Introduction
- 2 Méthodes et outils
 - Agilité et tests
 - Style et conventions
 - Bibliothèques Python
- 3 Modélisation objet
 - Conception et modélisation
 - Domaine métier
- 4 Démonstration
- 5 Conclusion

Rappel du sommaire

- 1 Introduction
- 2 Méthodes et outils
- 3 Modélisation objet
- 4 Démonstration
- 5 Conclusion

Présentation



Bérenger Ossété Gombé

- Master 1 en informatique (2016-2017)
 - Université de Franche-Comté (UFR-ST)
 - Spécialité génie logiciel
- Formation Python chez OpenClassRooms depuis janvier 2022

Le Projet n°4 : développer un programme logiciel en Python

Objectifs

- Produire du code robuste
- Utiliser la POO
- Structurer un projet python

Contexte fictif : le club d'échecs



Figure – Logo du club d'échecs

Personnages

- Nous incarnons un développeur indépendant
- Quelques membres du club d'échecs
 - Élie, notre amie
 - Édouard, l'organisateur
 - Charlie, l'assistant informatique

Contexte fictif : le club d'échecs

Problématique

- « *Nous utilisons actuellement une **application en ligne** pour nous aider à gérer nos tournois d'échecs hebdomadaires. Malheureusement, cette application nous a déçus par le passé. Elle **tombe souvent en panne**, ce qui signifie que les matches sont retardés.* » - Spécification technique
- « *Après que nous avons eu terminé le premier tour, **Internet a cessé de fonctionner** et le directeur du tournoi n'a pas pu entrer les scores sur le site des résultats* » - Témoignage d'Élie

Exigences fonctionnelles

- Gestion des tournois
- Gestion des classements
- Génération de rapports pouvant être exportés¹
- Fonctionnement hors-ligne
- Interaction utilisateur *via* une interface console
- Persistance *via* une base de données

1. Il faut permettre cette future fonctionnalité.

Exigences non-fonctionnelles²

■ Maintenabilité

- « Le code doit être aussi propre et maintenable que possible pour éviter les bugs. »

■ Fiabilité

- « Le code doit être aussi propre et maintenable que possible pour éviter les bugs. »

■ Portabilité (GNU/Linux, Windows, MacOS)

- « Le programme devrait fonctionner sous Windows, Mac ou Linux »

■ Utilisabilité

- « Tant que le programme affiche les résultats du tournoi proprement, nous serons heureux ! »

Rappel du sommaire

1 Introduction

2 Méthodes et outils

- Agilité et tests
- Style et conventions
- Bibliothèques Python

3 Modélisation objet

4 Démonstration

5 Conclusion

Mise en place du backlog

- Utilisation d'un tableau Kanban *via* kanboard³
- Découpage des *user stories* avec critères d'acceptations

Pourquoi tester ?


- La fiabilité est un enjeu du projet
 - « *Le code doit être aussi propre et maintenable que possible **pour éviter les bugs.*** » ⁴
- L'algorithme du tournoi Suisse doit être fiable car au cœur du domaine métier

Outils de tests

- Tests unitaires *via* PyTest ⁵
- Tests d'acceptations *via* Behave (cucumber) ⁶

4. D'après la spécification technique.

5. <https://docs.pytest.org/en/7.1.x/>

6. <https://behave.readthedocs.io/en/stable/> 

Style et conventions

PEP 8

- Vérification de la conformité du programme avec PEP 8 *via* Flake8⁷

Google et OpenStack

- Google Python Style Guide⁸
- OpenStack Style Guidelines⁹
- Utilisation de l'extension hacking¹⁰ de Flake8

7. <https://flake8.pycqa.org/en/latest/>

8. <https://google.github.io/styleguide/pyguide.html>

9. <https://docs.openstack.org/hacking/latest/user/hacking.html>

10. <https://pypi.org/project/hacking/>

Bibliothèques Python

Bibliothèques

- Style et conventions
 - Flake8
 - Flake8-html
 - Hacking
- Tests
 - PyTest
 - Behave
- Base de données
 - TinyDB
- Parseur XML
 - BeautifulSoup

Rappel du sommaire

- 1 Introduction
- 2 Méthodes et outils
- 3 Modélisation objet**
 - Conception et modélisation
 - Domaine métier
- 4 Démonstration
- 5 Conclusion

Vocabulaire : traductions

Traduction en anglais du vocabulaire

Tournoi *tournament*

Classement *ranking*

Rapport *report*

Joueur *player*

Pair de joueur *pair of players*

Un tour *a round*

Résultats *scores*

Match nul *draw*

Découpage en modules

Modèle MVC

- un modèle (*package model*)
- une vue (*package view*)
- un contrôleur (*package controller*)

Architecture

Propriétés

- 1 Indépendance du modèle vis-à-vis du contrôleur et de la vue.
- 2 Le contrôleur est le point central de l'architecture.
- 3 La vue et le modèle communiquent *via* des évènements.

Architecture

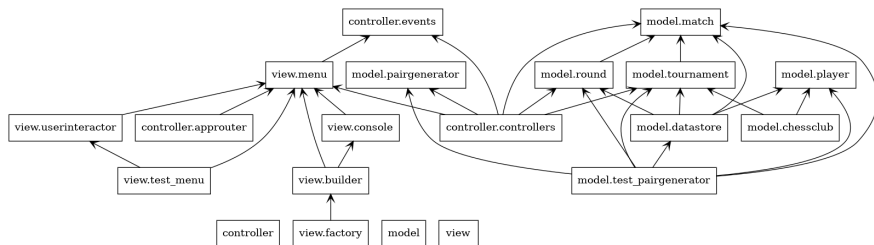


Figure – Les *packages*

Le modèle

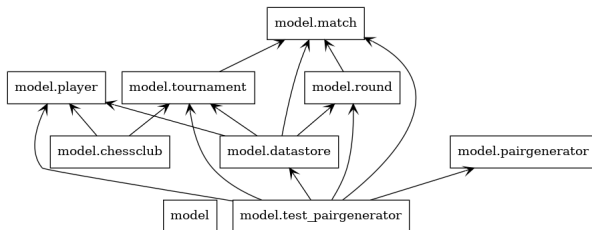


Figure – Structure du modèle

Remarque

- ChessClub implémente le *design pattern Facade*.

La vue

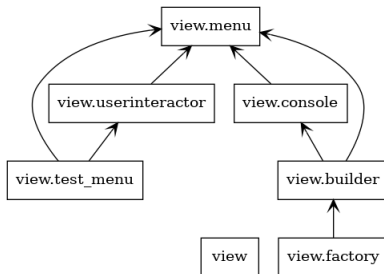


Figure – Structure de la vue

Construction de la vue

- La factory
 - permet de charger la description XML de la vue.
 - utilise un Builder pour construire la vue.

La vue : représentation XML

```
<menu title='Menu Principal'>
  <entry action='S'>Sauvegarder</entry>
  <entry action='C'>Charger</entry>

  <menu title="Rapports" action="R">
    <entry action='a'>Acteurs (par nom)</entry>
    <entry action='A'>Acteurs (par classement)</entry>
    <entry action='t'>Tournoi (acteurs par nom)</entry>
    <entry action='T'>
      Tournoi (acteurs par classement)
    </entry>
    <entry action='l'>Liste des tournois</entry>
    <entry action='L'>Liste des tours</entry>
    <entry action='m'>Liste des matchs</entry>
    <link action='q'>Menu Principal</link>
  </menu>
```

```
<!-- ... -->
```

Événements

```
class Event:
    def __init__(self, name, **kwargs):
        self._name = name
        self._values = kwargs

    def name(self):
        return self._name

    def get(self, value_name):
        return self._values[value_name]
```

Principe

- Utilisation du *design pattern Observer*.
- EventSource émet des événements.
- EventListener reçoit des événements.
- → EventListener observe EventSource.

Émettre des événements

```
class EventSource:
    def __init__(self):
        self._listeners = []

    def add_listener(self, listener):
        self._listeners.append(listener)

    def remove_listener(self, listener):
        self._listeners.remove(listener)

    def notify(self, event):
        for listener in self._listeners:
            listener.on_event(event)
```


Recevoir des événements

```
class EventListener(ABC):  
    def __init__(self):  
        pass  
  
    @abstractmethod  
    def on_event(self, event):  
        pass
```

Le contrôleur

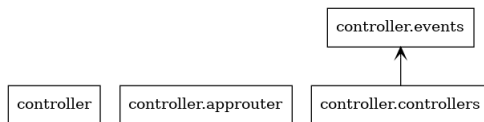


Figure – Structure du contrôleur

Le routeur

- AppRouter contrôle l'application.
- Chaque contrôleur est un état de AppRouter.
- Utilisation du *Design Pattern State*.

Exemple de contrôleur

```
class MainController(Controller):
    def __init__(self):
        super().__init__()
        self._player_info = []
        self._play_ctrl = PlayController(self)
        self._setup_ctrl = SetupController(self)
        self._report_ctrl = ReportController(self)

    def on_event(self, event):
        if event.get('action') == 'S':
            self.on_save()

        if event.get('action') == 'C':
            self.on_load()

        if event.get('action') == 'R':
            self._router.set_controller(self._report_ctrl)

# ...
```

Interactions avec l'utilisateur

```
class UserInteractor(ABC):
    @abstractmethod
    def ask(self, msg):
        pass

    @abstractmethod
    def tell(self, msg):
        pass


class ConsoleUserInteractor(UserInteractor):
    def ask(self, msg):
        if msg == '\\\\quitter':
            raise StopAndSave()
        return input(msg)

    def tell(self, msg):
        print(msg)
```

...

Exemples d'interactions

```
self._view.io().tell(f'\nMatch: {pair[0].name} '
                    f'contre {pair[1].name}')
res = self._view.io().ask('Résultat du match '
                          f'(0: {pair[0].name}, '
                          f'1: {pair[1].name}, '
                          f'2: match nul): ')
```

Instanciación de los componentes

```
if __name__ == '__main__':  
    user_interactor = ConsoleUserInteractor()  
    factory = XMLFactory(user_interactor)  
    view = factory.load_from_file('data/menu.xml')  
    datastore = TinyDBStore()  
    model = ChessClub(datastore)  
  
    router = AppRouter(user_interactor, model, view)  
    router.set_error_manager(PrintErrorManager(user_interactor))  
    router.set_controller(MainController())  
  
    try:  
        router.run()  
    except KeyboardInterrupt:  
        print()  
        model.quit()
```

Système Suisse des tournois

```
class PairGenerator(ABC):
    @abstractmethod
    def generate(self, tournament):
        pass

class SwissPairGenerator(PairGenerator):
    def __init__(self):
        pass

    def generate(self, tournament):
        if tournament.is_first_round():
            return self._generate_first_round(tournament)
        return self._generate(tournament)
```

Système Suisse des tournois : premier tour

```
def _generate_first_round(self, tournament):
    players = copy.deepcopy(tournament.players)
    result = []

    players.sort(key=lambda p: int(pranking))

    i = 0
    total = int(len(players)/2)

    while i < total:
        result.append((players[i], players[total + i]))
        i += 1

    return result
```


Système Suisse des tournois : tours suivants

```
def _generate_from_scores(self, tournament, player_scores):
    players = copy.deepcopy(tournament.players)
    result = []

    players.sort(reverse=True, key=lambda p: player_scores[p.name])

    while len(players) > 0:
        p0 = 0
        p1 = 1

        my_round = tournament.previous_round()
        if my_round is not None:
            while True:
                matches = tournament.find_matches_by_players(players[p0],
                                                                players[p1])

                if len(matches) > 0:
                    p1 += 1
                else:
                    break

            result.append((players[p0], players[p1]))

        del players[p0]
        del players[p1 - 1]

    return result
```

Principes

- Un DataStore est un objet capable de
 - stocker,
 - sauvegarder et
 - charger les données importantes du système.
- InMemoryStore → garde tout en mémoire durant l'exécution du programme.
- TinyDBStore → travaille avec une base de données via TinyDB.

Persistence

```
class DataStore(ABC):
    def __init__(self):
        self._players = []
        self._tournaments = []

    @abstractmethod
    def save(self):
        pass

    @abstractmethod
    def load(self):
        pass

    def store_player(self, player):
        self._players.append(player)

    def store_tournament(self, tournament):
        self._tournaments.append(tournament)

    def players(self):
        return self._players

    def tournaments(self):
        return self._tournaments

    def find_players_by_ranking(self, ranking):
        return [p for p in self._players if int(p.ranking) == int(ranking)]

    def find_players_by_name(self, name):
        return [p for p in self._players if p.name == name][0]
```

Persistence : sérialisation d'un match

```
@property
def __dict__(self):
    return {
        'player_0': self._player_0.name,
        'player_1': self._player_1.name,
        'result': int(self._result)
    }
```

Persistence : sérialisation d'un tour

```
@property
def __dict__(self):
    result = {
        'name': self._name,
        'start': str(self._start),
        'end': str(self._end),
        'matches': []
    }

    for m in self._matches:
        result['matches'].append(m.__dict__)

    return result
```

Persistence : sérialisation d'un tournoi

```
@property
def __dict__(self):
    result = {}
    result['name'] = self._name
    result['place'] = self._place
    result['start_date'] = str(self._start_date)
    result['end_date'] = str(self._end_date)
    result['category'] = self._category
    result['description'] = self._description
    result['players'] = [p.name for p in self._players]

    rounds = []
    for r in self._rounds:
        rounds.append(r.__dict__)
    result['rounds'] = rounds
    result['current_round'] = self._current_round

    return result
```

Sauvegarde

```
def save(self):  
    player_table = self._db.table('Player')  
    player_table.truncate()  
    for player in self._players:  
        player_table.insert(player.__dict__)  
  
    tournament_table = self._db.table('Tournament')  
    tournament_table.truncate()  
    for tournament in self._tournaments:  
        tournament_table.insert(tournament.__dict__)
```

Chargement : les joueurs

```
def load(self):
    player_table = self._db.table('Player')

    for p in player_table.all():
        self.store_player(Player(
            p['last_name'],
            p['first_name'],
            p['date_of_birth'],
            p['gender'],
            p['ranking']
        ))

    tournament_table = self._db.table('Tournament')
    for t in tournament_table.all():
        # -> suite
```


Chargement : les tournois

```
the_tournament = Tournament('', '', '', '', '', '')
the_tournament._name = t['name']
the_tournament._place = t['place']
the_tournament._start_date = t['start_date']
the_tournament._end_date = t['end_date']
the_tournament._category = t['category']
the_tournament._description = t['description']
the_tournament._current_round = t['current_round']

for name in t['players']:
    the_tournament.add_player(self.find_players_by_name(name))

for r in t['rounds']:
    # -> suite
```

Chargement : les tours et les matches

```
the_round = Round(r['name'],
                  datetime.date(*[
                      int(i)
                      for i in r['start'].split('-')
                  ]),
                  datetime.date(*[
                      int(i)
                      for i in r['end'].split('-')
                  ]))

for match in r['matches']:
    m = Match(self.find_players_by_name(match['player_0']),
              self.find_players_by_name(match['player_1']))
    m.set_result(int(match['result']))
    the_round.add_match(m)

the_tournament.add_round(the_round)

self._tournaments.append(the_tournament)
```

Rappel du sommaire

- 1 Introduction
- 2 Méthodes et outils
- 3 Modélisation objet
- 4 Démonstration**
- 5 Conclusion

Démonstration

Fonctionnalités clefs

- 1 Création et exécution d'un tournoi
- 2 Mise-à-jour du classement
- 3 Génération d'un rapport
- 4 Sauvegarde et chargement

Rappel du sommaire

- 1 Introduction
- 2 Méthodes et outils
- 3 Modélisation objet
- 4 Démonstration
- 5 Conclusion**

Conclusion

- Mise en place d'outils pour améliorer la robustesse du code.
 - PEP-8
 - L'extension *hacking* de Flake8
 - Tests unitaires et d'acceptations
- Modélisation objets du domaine métier ainsi que de l'interface console conformément aux principes SOLID.
- Utilisation de divers Design Patterns.
 - **MVC** → pour l'architecture globale.
 - **Facade** → pour accéder au modèle.
 - **Builder** → pour construire la vue.
 - **Factory** → pour importer la vue xml.
 - **Observer** → pour la communication vue-modèle *via* le contrôleur.

Merci de votre attention

1 Introduction

2 Méthodes et outils

- Agilité et tests
- Style et conventions
- Bibliothèques Python

3 Modélisation objet

- Conception et modélisation
- Domaine métier

4 Démonstration

5 Conclusion