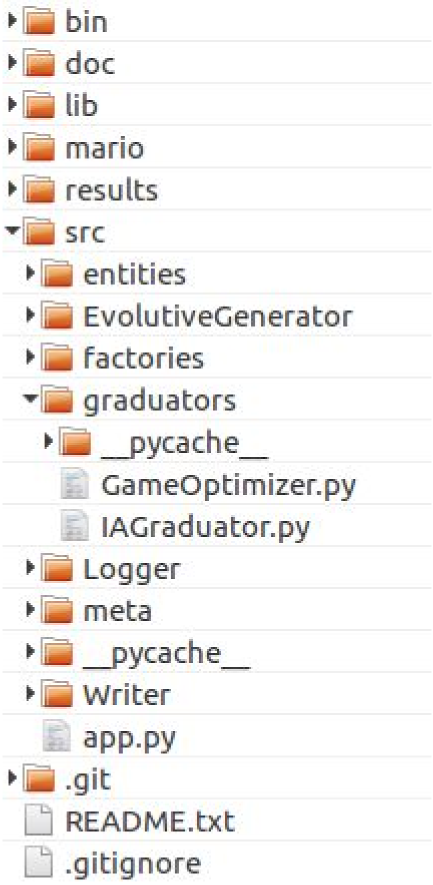
Étape 4 : Évaluer les intelligences avec l’*IAGraduator*

Dossier /src/graduators



```
bin
doc
lib
mario
results
src
  entities
  EvolutiveGenerator
  factories
  graduators
    __pycache__
    GameOptimizer.py
    IAGraduator.py
  Logger
  meta
  __pycache__
  Writer
  app.py
.git
README.txt
.gitignore
```

- *Graduator*
- *IAGraduator*
- *GameOptimizer* : Optimise le temps d'évaluation, notamment grâce à la détection des boucles.

```
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from abc import ABCMeta, abstractmethod


class Graduator(metaclass=ABCMeta):
    """Graduate individuals

    This is an abstract class to inherit.
    Assess individual's performances and assign them a score.
    The Graduator is to think as a bridge between the Generator and the software.
    It is designed to use the software to make evolute individuals.
    IT IS THE NATURE.
    Individuals are represented by root GeneticElement instances.
    """


    @abstractmethod
    def grade(self, individual, generation_id):
        """Assign a score to a individual

        Has to be implemented.

        Expects:
            individual to be an GeneticElement

        return int or any sortable object The score
        """

        raise NotImplementedError


    def gradeAll(self, individuals, generation_id, dispatch):
        """Assign a score to each individual

        Expects:
            individuals to be a list of GeneticElement

        Return a list of couple (score, GeneticElement)
        """

        grading  = []
        for individual  in individuals:
            graduation  = self.grade(individual , generation_id )
             grading.append((graduation , individual ))
             dispatch(individual , graduation )
        return grading
```

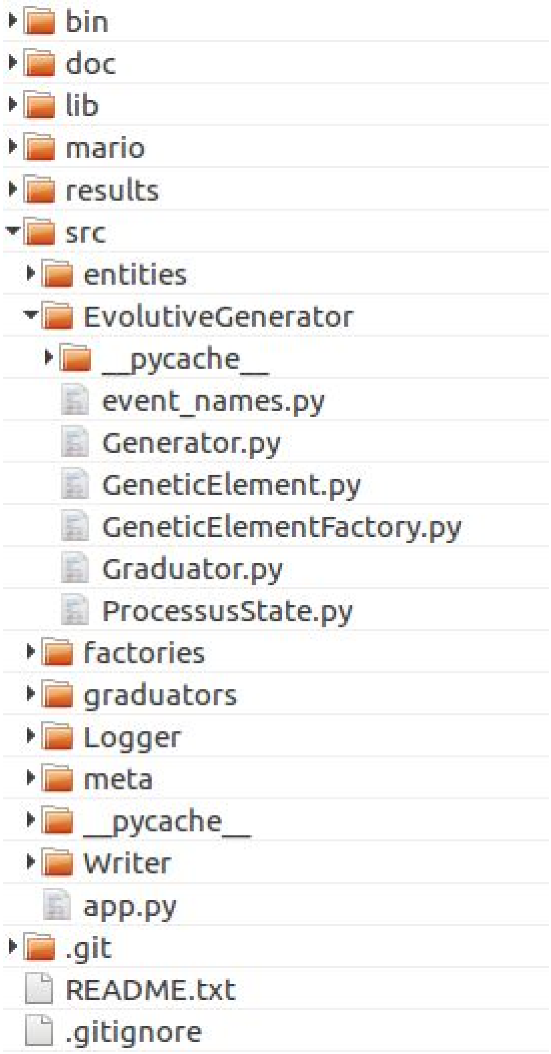
```
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from math import ceil

from lib.inject_arguments import inject_arguments
from lib.inherit_docstring import inherit_docstring

from src.meta.ABCInheritableDocstringsMeta import ABCInheritableDocstringsMeta
from mario.bridge.config import Config
from mario.bridge.launch import launch
from src.EvolutiveGenerator.Graduator import Graduator
from src.entities.Result import Result


class IAGraduator(Graduator, metaclass=ABCInheritableDocstringsMeta):
    """Graduate IA"""

    @inject_arguments
    def __init__(self, event_dispatcher, show = False):
        self.mario_x = 0
        self.max_y = -500


    def gradeIAWithConfig(self, ia, config):
        # Init
        self.mario_x = 0
        self.max_y = -500

        # Give the event_dispatcher to neurons
        for neuron in ia.neurons:
            neuron.event_dispatcher = self.event_dispatcher

        self.event_dispatcher.listen('game.frame', self.onFrame)

        # Launch game
        persist = launch(config)

        # Remove the event_dispatcher from neurons
        for neuron in ia.neurons:
            del neuron.event_dispatcher

        # Make the result
        result = Result(persist['camera start x'] + self.mario_x, self.max_y)

        # Return the score
        return result


    @inherit_docstring
    def grade(self, ia, generation_id):
        time = 1 + ceil(generation_id / 2)
        if time > 401:
            time = 401

        return self.gradeIAWithConfig(ia, Config(self.show, self.event_dispatcher, time))


    def onFrame(self, frame):
        self.mario_x = frame.mario.rect.x
        self.max_y = max(self.max_y, - frame.mario.rect.y)
```

06/14/17 05:33:34 /home/zz/Documents/TIPE/src/graduators/GameOptimizer.py

```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from lib.inject_arguments import inject_arguments


class GameOptimizer:
    @inject_arguments
    def __init__(self, event_dispatcher):
        self.event_dispatcher.listen('game.frame', self.onFrame)
        self.event_dispatcher.listen('action', self.onAction)


    def onFrame(self, frame):
        # Reset
        if frame.current_frame < 5:
            self.action_detected = False
            self.mario_x = 0
            self.last_mario_x_change = 0
            self.last_points = []

        # Detect inactivity (10 frames)
        if frame.current_frame > 10 and not self.action_detected:
            self.event_dispatcher.dispatch('stop')

        # Detect x-inactive IA (2,5 sec == 150 frames)
        if self.mario_x != frame.mario.rect.x:
            self.last_mario_x_change = frame.current_frame
        if frame.current_frame > self.last_mario_x_change + 150:
            self.event_dispatcher.dispatch('stop')

        self.mario_x = frame.mario.rect.x

        # Detect looping IA (12 sec == 720 frames)
        point = int(self.mario_x / 10), int(frame.mario.rect.y / 10)
        self.last_points.append(point)
        if len(self.last_points) > 720:
            self.last_points.pop(0)

        if frame.current_frame < 720:
            return
        indexes = [i for i, v in enumerate(self.last_points) if v == point]
        if len(indexes) >= 5:
            self.event_dispatcher.dispatch('stop')


    def onAction(self, event):
        self.action_detected = True
```

Étape 5 : Créer l'algorithme génétique avec le *Generator*

Dossier /src/EvolutiveGenerator



```
bin
doc
lib
mario
results
src
  entities
  EvolutiveGenerator
    __pycache__
    event_names.py
    Generator.py
    GeneticElement.py
    GeneticElementFactory.py
    Graduator.py
    ProcessusState.py
  factories
  graduators
  Logger
  meta
  __pycache__
  Writer
  app.py
.git
README.txt
.gitignore
```

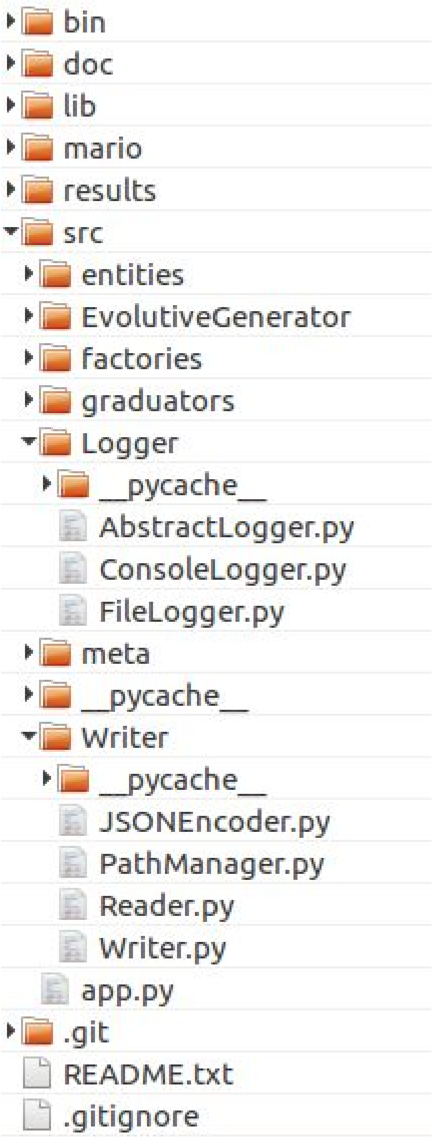
- *Generator*

```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from random import choice, sample
from math import ceil, inf
from inspect import isfunction, ismethod
from re import match
from operator import itemgetter

from lib.eventdispatcher import EventDispatcher
from .ProcessusState import ProcessusState
from .event_names import *


class Generator:
    """Handle the generation proccess

    The generator, at the heart of the generation process, has three charges:
        - create a population of individuals
        - select a subset of the population, based on their performances
        - breed individuals of the selection to form a new population
    Individuals are represented by root GeneticElement instances.
    Use a Graduator to grade performances.
    Extending it is strongly adviced.

    The generator dispatches several events through its internal dispatcher:
        processus.start,
        processus.done,
        creation.start,
        creation.done,
        generation.start,
        generation.done,
        grading.start,
        grading.process,
        grading.done,
        selection.start,
        selection.done,
        breeding.start,
        breeding.process,
        breeding.done,
    and processus.resume
    See ProcessusState.py for informations carried by events.
    In particular, the population is available through creation.done and
    generation.start/done.
    """


    def __init__(self, factory, graduator, listeners = [], end_statement = None):
        """Init

        Expects:
            factory to be a class inheriting of GeneticElementFactory
            graduator to be a instance inheriting of Graduator
            listeners to be a list of listeners (see below)
            end_statement to be a boolean function
        Listeners can be:
            - couples (event_name, listener)
            - tuples (event_name, listener, priority)
            - functions and methods if their names follow the format
              'onEventName'. For example, listener 'onProcessusStart' will
              listen on 'processus.start'.
              If they have a priority attribute, it will be used as priority.
            - objects: every method following the format above is added to
            listeners.
        factory and graduator are automatically added to listeners.
        Priorities has to be strictly smaller than 1000.
        """

        self.factory = factory
        self.graduator = graduator
        self.end_statement = end_statement

        self.state = ProcessusState()
        self.iterating = False

        self.dispatcher = EventDispatcher()
        listeners.append(factory)
        listeners.append(graduator)

        listeners.extend([
            (PROCESSUS.START, self.create, 1000),
            (CREATION.DONE, self.initGeneration, 1000),
            (GENERATION.START, self.grade, 1000),
            (GRADING.DONE, self.select, 1000),
            (SELECTION.DONE, self.breed, 1000),
            (BREEDING.DONE, self.endGeneration, 1000),
        ])

        # Get all objects' methods
        listenersMethods = listeners.copy()
        for listener in listeners:
            if not (type(listener) is tuple or isfunction(listener) or ismethod(listener)):
                listenersMethods.remove(listener)
                for method in [method for method in dir(listener) if ismethod(getattr(listener, method))]:
                    if match('on([A-Z]\w+)', method):
                        listenersMethods.append(getattr(listener, method))

        # Inscribe all listeners
        for listener in listenersMethods:
            if type(listener) is tuple:
                self.dispatcher.listen(*listener)
            else:
                # Parse method names to get event names
                m = match('on([A-Z]\w+)', listener.__name__)
                if m:
                    event_name = ''
                    camel_event_name = m.group(1)
                    while True:
                        m = match('([A-Z][a-z0-9_]+)(\w*)', camel_event_name)
                        if not m:
                            break
                        if event_name:
                            event_name += '.'
                        event_name += m.group(1).lower()
                        camel_event_name = m.group(2)
                    self.dispatcher.listen(event_name, listener,
                        0 if not hasattr(listener, 'priority') else listener.priority)
                else:
                    raise ValueError('The given listener do not follow the format onEventName.')


    def dispatch(self, event_name):
        self.state.event_name = event_name
        self.dispatcher.dispatch(event_name, self.state)

    def dispatchGrading(self, individual, graduation):
        """Shorthand to dispatch grading events"""
        self.state.individual = individual
        self.state.graduation = graduation
        self.dispatch(GRADING.PROGRESS)


    def initProcessus(self):
        self.dispatch(PROCESSUS.START)

    def endProcessus(self):
        self.dispatch(PROCESSUS.DONE)

    def initGeneration(self, state):
        """Handle iteration"""
        self.iterating = True

        try:
            while True:
                state.generation_id += 1
                self.dispatch(GENERATION.START)
        except StopIteration:
            pass

        self.iterating = False

    def endGeneration(self, state):
        self.dispatch(GENERATION.DONE)
        if (
            state.generation_id >= state.generations
            or (state.generations == inf and self.end_statement(state))
        ):
            self.endProcessus()
            if self.iterating:
                raise StopIteration

        elif not self.iterating:
            self.initGeneration(state)


    def create(self, state):
        """Generate a whole initial population"""

        state.generation_id = 0
        self.dispatch(CREATION.START)

        state.population = set([self.factory.create() for i in range(state.pop_length)])

        self.dispatch(CREATION.DONE)


    def resumeGrading(self, state):
        """Grade non-graded individuals"""

        graded_individuals = set([individual for score, individual in state.grading])
        to_grade = state.population.difference(graded_individuals)

        state.grading.extend(
            self.graduator.gradeAll(to_grade, state.generation_id, self.dispatchGrading)
        )
        state.grading.sort(key=itemgetter(0), reverse=True)

        self.dispatch(GRADING.DONE)


    def grade(self, state):
        """Grade all individuals"""

        self.dispatch(GRADING.START)

        state.grading = []
        self.resumeGrading(state)


    def select(self, state):
```

```python
        """Operate the selection

        This is a basic system to be overcome.
        The selection is a subset of the population.
        """

        self.dispatch(SELECTION.START)

        # Get a list of individuals
        ordered_individuals = [c[1] for c in state.grading]

        # The number of individuals to select
        selection_length = ceil(len(state.population) * state.proportion)
        # Among the [selection_length] best individuals select selection_length*(1-state.chance) ones
        selection = set(sample(
            ordered_individuals[:selection_length],
            int(selection_length * (1 - state.chance))
        ))
        # Complete selection with random individuals
        unused_individuals = state.population.difference(selection)
        while len(selection) < selection_length:
            choiced = choice(list(unused_individuals))
            selection.add(choiced)
            unused_individuals.remove(choiced)

        state.selection = selection
        self.dispatch(SELECTION.DONE)


    def breed(self, state):
        """Generate a new population based on selection

        This is a basic system to be overcome.
        """

        self.dispatch(BREEDING.START)

        new_pop = set()

        # Add artificially the best individual to the new pop : survival principle
        best = state.grading[0][1]
        new_pop.add(best)
        state.offspring = best
        state.parents = (best, best)
        self.dispatch(BREEDING.PROGRESS)

        while len(new_pop) < state.pop_length:
            parents = tuple([choice(list(state.selection)) for i in range(2)])
            offspring = self.factory.breed(*parents)
            new_pop.add(offspring)

            state.offspring = offspring
            state.parents = parents
            self.dispatch(BREEDING.PROGRESS)

        state.population = new_pop

        self.dispatch(BREEDING.DONE)


    def process(self, processus_id, generations, pop_length = 500, proportion = .5, chance = 0):
        """Process multiple generations

        If generations == inf then self.end_statement will be the stopping statement.

        Expects:
            generations to be an int or inf
            pop_length to be an int

            proportion to be a float between 0 and 1
            chance to be a float between 0 and 1

        Return the last generation
        """

        self.state.processus_id = processus_id
        self.state.generations = generations
        self.state.pop_length = pop_length
        self.state.proportion = proportion
        self.state.chance = chance

        self.initProcessus()

        return self.state.population


    def resume(self, state):
        """Resume a stopped processus """

        self.dispatcher.dispatch(PROCESSUS.RESUME, state)

        self.state = state
        if state.event_name in (
            PROCESSUS.START, CREATION.DONE, GENERATION.START, GRADING.DONE, SELECTION.DONE, BREEDING.DONE
        ):
            self.dispatch(self.state.event_name)
        elif state.event_name == CREATION.START:
            self.create(state)
        elif state.event_name == GRADING.START:
            self.grade(state)
        elif state.event_name == GRADING.PROGRESS:
            self.resumeGrading(state)
        elif state.event_name == SELECTION.START:
            self.select(state)
        elif state.event_name in (BREEDING.START, BREEDING.PROGRESS):
            self.breed(state)
        elif state.event_name == GENERATION.DONE:
            self.endGeneration(state)
        elif state.event_name == PROCESSUS.DONE:
            self.endProcessus()
        else:
            raise ValueError(state.event_name + 'is not handled.')

        return self.state.population
```

Étape 6 : Enregistrer les données dans des fichiers

Dossiers /src/Writer et /src/Logger



```
▸ bin
▸ doc
▸ lib
▸ mario
▸ results
▾ src
  ▸ entities
  ▸ EvolutiveGenerator
  ▸ factories
  ▸ graduators
  ▾ Logger
    ▸ __pycache__
      AbstractLogger.py
      ConsoleLogger.py
      FileLogger.py
  ▸ meta
  ▸ __pycache__
  ▾ Writer
    ▸ __pycache__
      JSONEncoder.py
      PathManager.py
      Reader.py
      Writer.py
    app.py
▸ .git
  README.txt
  .gitignore
```

- *JSONEncoder*
- *PathManager*
- *Reader*
- *Writer*

- *AbstractLogger*
- *ConsoleLogger*
- *FileLogger*

06/14/17 05:34:40 /home/zz/Documents/TIPE/src/Writer/JSONEncoder.py

```python
#!/usr/bin/python3.5
# -*-coding:Utf-8 -*

from json import JSONEncoder as BaseEncoder


class JSONEncoder(BaseEncoder):
    def default(self, obj):
        if hasattr(obj,'reprJSON'):
            return obj.reprJSON()
        elif type(obj) is set:
            return list(obj)
        else:
            return BaseEncoder.default(self, obj)
```

```python
#!/usr/bin/python3.5
# -*-coding:Utf-8 -*

from pathlib import Path
from os import getcwd
from re import fullmatch


class PathManager:
    """Make all paths

    This is a static class.
    """

    ROOT = Path(getcwd() + '/results/')


    @classmethod
    def newProcessusId(cls):
        path = Path(cls.ROOT)
        cls.makeDir(path)
        ids = [-1]

        for folder in path.iterdir():
            match = fullmatch('processus-(\d+)', folder.name)
            if match is not None:
                ids.append(int(match.group(1)))

        return max(ids) + 1


    @classmethod
    def getPath(cls,
        processus_id, generations = None,
        generation_id = None, ia_id_or_file = None, read_only = False
    ):
        path = Path(cls.ROOT)

        # processus-00000/
        path /= 'processus-' + '{0:05d}'.format(processus_id)

        # processus-00000/generation-00/...
        if generation_id is not None:
            generations = generations if type(generations) is int else '00000'
            path /= 'generation-' + '{0:0{1}d}'.format(generation_id, len(str(generations)))

            # processus-00000/generation-00/selection/...
            if ia_id_or_file in ('grading', 'final_grading', 'selection'):
                path /= 'selection/' + ia_id_or_file
            # processus-00000/generation-00/population/ia-000
            elif type(ia_id_or_file) is int:
                path /= 'initial_pop' if generation_id == 0 else 'population'
                ia_id = ia_id_or_file
                if ia_id is not None:
                    path /= 'ia-{}.json'.format(ia_id)
                else:
                    raise ValueError('ia_id not given')
            # processus-00000/generation-00/breeding
            elif ia_id_or_file == 'breeding':
                path /= 'breeding'
            # processus-00000/generation-00/generation
            elif ia_id_or_file is None:
                path /= 'generation'
            else:
                raise ValueError('wrong ia_id_or_file value')
        # processus-00000/processus
        else:
            path /= 'processus'

        if (
            path.name in ('generation', 'processus', 'final_grading', 'selection')
            or path.parent.name == 'population'
        ):
            path = path.with_suffix('.json')

        if not read_only:
            cls.makeDir(path.parent)

        return path


    @staticmethod
    def makeDir(path):
        path.mkdir(parents=True, exist_ok=True)
```

```python
#!/usr/bin/python3.5
# -*-coding:Utf-8 -*

from json import loads
from re import findall
from operator import itemgetter

from .JSONEncoder import JSONEncoder
from .PathManager import PathManager
from src.factories.IAFactory import IAFactory
from src.EvolutiveGenerator.ProcessusState import ProcessusState
from src.EvolutiveGenerator.event_names import *


class Reader:
    """Read files

    This is a static class.

    Public API:
        processusExists(processus_id)
        getProcessusState(processus_id)
        getIa(processus_id, ia_id)
        getBestIa(processus_id, generation_id=None)
        getData(processus_id)
    """


    @staticmethod
    def getPath(*args, **kwargs):
        return PathManager.getPath(*args, **kwargs, read_only=True)


    @staticmethod
    def readJSON(path):
        return loads(path.read_text())


    @staticmethod
    def readGrading(path):
        return [tuple(loads(json_array)) for json_array in findall('\[.+\]', path.read_text())]


    @classmethod
    def getProcessusParams(cls, processus_id):
        path = cls.getPath(processus_id)
        if not path.parent.exists():
            raise ValueError("Processus {} doesn't exists. ".format(processus_id))
        if not path.exists():
            raise ValueError("Processus {} doesn't have processus.json file. ".format(processus_id))

        return cls.readJSON(path)


    @classmethod
    def getLastGeneration(cls, processus_id, generations):
        '''Get id of the processus' last generation, else -1 '''
        # Get first inexistant generation
        generation_id = 0
        while cls.getPath(processus_id, generations, generation_id).parent.exists():
            generation_id += 1

        return generation_id - 1


    @classmethod
    def getLastGradedGeneration(cls, processus_id, generations):
        # Get first inexistant final_grading file's generation
        generation_id = 1
        while cls.getPath(processus_id, generations, generation_id, 'final_grading').exists():
            generation_id += 1

        return generation_id - 2


    @classmethod
    def getGenerationOf(cls, processus_id, generations, ia_id):
        generation_id = 1
        while True:
            path = cls.getPath(processus_id, generations, generation_id, 'final_grading')
            if not path.exists():
                raise ValueError("IA {} doesn't exist ! ".format(ia_id))
            final_grading = cls.readJSON(path)
            for score, _ia_id in final_grading:
                if _ia_id == ia_id:
                    return generation_id - 1
            generation_id += 1

        raise RuntimeError


    @classmethod
    def getPopulation(cls, processus_id, generation_id, generations):
        population = set()
        for ia_file in (
            cls.getPath(processus_id, generations, generation_id).parent
            / ('population' if generation_id > 0 else 'initial_pop')
        ).iterdir():
            population.add(IAFactory.hydrate(cls.readJSON(ia_file)))
        return population


    @classmethod
    def getIa(cls, processus_id, ia_id):
        generations = cls.getProcessusParams(processus_id)['generations']
        generation_id = cls.getGenerationOf(processus_id, generations, ia_id)
        ia_file = cls.getPath(processus_id, generations, generation_id, ia_id)
        return IAFactory.hydrate(cls.readJSON(ia_file)), generation_id


    @classmethod
    def getBestIa(cls, processus_id, generation_id = None):
        generations = cls.getProcessusParams(processus_id)['generations']
        if generation_id is None:
            generation_id = generations if type(generations) is int else cls.getLastGradedGeneration(processus_id, generations)
        grading = cls.readJSON(cls.getPath(processus_id, generations, generation_id + 1, 'final_grading'))
        grading.sort(key=lambda c: c[0]['score'], reverse=True)
        ia_id = grading[0][1]
        ia_file = cls.getPath(processus_id, generations, generation_id, ia_id)
        return IAFactory.hydrate(cls.readJSON(ia_file)), generation_id


    @classmethod
    def processusExists(cls, processus_id):
        path = cls.getPath(processus_id)
        if not path.parent.exists():
            return False
        return True


    @classmethod
    def getData(cls, processus_id):
        generation_id = 1
        generations = cls.getProcessusParams(processus_id)['generations']
        data = []

        while True:
            path = cls.getPath(processus_id, generations, generation_id, 'final_grading')
            if not path.exists():
                break
            final_grading = cls.readJSON(path)
            data.append((generation_id - 1, final_grading))
            generation_id += 1

        return data


    @classmethod
    def getProcessusState(cls, processus_id):
        '''
        for state.event_name in (
            PROCESSUS.START,
            CREATION.START,
            CREATION.DONE,
            GENERATION.START,
            GRADING.START,
            GRADING.PROGRESS,
            GRADING.DONE,
            SELECTION.START,
            SELECTION.DONE,
            BREEDING.START,
            BREEDING.PROGRESS,
            BREEDING.DONE,
            GENERATION.DONE,
            PROCESSUS.DONE
        )
        '''

        state = ProcessusState()
        state.processus_id = processus_id
        state.__dict__.update(cls.getProcessusParams(processus_id))

        getPath = lambda generation_id, file_name = None: cls.getPath(
            processus_id, state.generations, generation_id, file_name
        )

        generation_id = cls.getLastGeneration(processus_id, state.generations)
        # If none generation folder exist
        if generation_id == -1:
            state.event_name = PROCESSUS.START
            return state
        state.generation_id = generation_id

        # Get event_name
        state.event_name = cls.readJSON(getPath(state.generation_id))['event_name']

        if state.event_name in (CREATION.DONE, BREEDING.DONE, GENERATION.DONE, PROCESSUS.DONE):
            state.population = cls.getPopulation(state.processus_id, state.generation_id, state.generations)
        else:
            state.population = cls.getPopulation(state.processus_id, state.generation_id - 1, state.generations)

        if state.event_name in (GRADING.PROGRESS):
            state.grading = cls.readGrading(getPath(state.generation_id, 'grading'))
        elif state.event_name in (GRADING.DONE, SELECTION.START):
            state.grading = cls.readJSON(getPath(state.generation_id, 'final_grading'))
        if state.grading is not None:
            indexed_pop = dict([(ia.id, ia) for ia in state.population])
            state.grading = [(score, indexed_pop[ia_id]) for (score, ia_id) in state.grading]

        if state.event_name in (SELECTION.DONE, BREEDING.START, BREEDING.PROGRESS):
            state .selection = cls.readJSON(getPath(state.generation_id , 'selection'))

        return state
```

06/14/17 05:36:07 /home/zz/Documents/TIPE/src/Writer/Writer.py

```python
#!/usr/bin/python3.5
# -*-coding:Utf-8 -*

from json import dumps

from .JSONEncoder import JSONEncoder
from src.Writer.PathManager import PathManager


class Writer:
    """Write IA in files"""


    def onAll(self, event):
        if event.event_name != 'processus.start':
            try:
                self.writeJSON({'event_name': event.event_name}, self.getPath(event.generation_id))
            except (SystemExit, KeyboardInterrupt):
                self.writeJSON({'event_name': event.event_name}, self.getPath(event.generation_id))
                raise
    onAll.priority = 1

    def onProcessusResume(self, event):
        self.__dict__.update(event.__dict__)

    def onProcessusStart(self, event):
        self.onProcessusResume(event)

        self.writeJSON(
            {
                'generations': event.generations,
                'pop_length': event.pop_length,
                'proportion': event.proportion,
                'chance': event.chance
            },
            self.getPath()
        )

    def onCreationDone(self, event):
        for ia in event.population:
            self.writeJSON(ia, self.getPath(event.generation_id, ia.id))

    def onGradingProgress(self, event):
        with self.getPath(event.generation_id, 'grading').open('a') as grading_file:
            grading_file.write(
                dumps([event.individual.id, event.graduation], cls=JSONEncoder, sort_keys=True, indent=4)
            )

    def onSelectionDone(self, event):
        self.writeJSON(
            [(score, ia.id) for (score, ia) in event.grading],
            self.getPath(event.generation_id, 'final_grading')
        )
        self.writeJSON(
            [ia.id for ia in event.selection],
            self.getPath(event.generation_id, 'selection')
        )

    def onBreedingProgress(self, event):
        self.writeJSON(event.offspring, self.getPath(event.generation_id, event.offspring.id))
        with self.getPath(event.generation_id, 'breeding').open('a') as breeding_file:
            breeding_file.write(
                '{} + {} -> {}\n'.format(event.parents[0].id, event.parents[1].id, event.offspring.id)
            )


    def getPath(self, *args, **kwargs):
        return PathManager.getPath(self.processus_id, self.generations, *args, **kwargs)

    def write(self, text, path):
        path.write_text(text)

    def writeJSON(self, data, path):
        self.write(dumps(data, cls=JSONEncoder, sort_keys=True, indent=4), path)
```

```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from abc import ABCMeta, abstractmethod


class AbstractLogger(metaclass=ABCMeta):
    """Log Generator events

    An abstract logger to implement, by defining the write() and overwrite() methods.
    """

    @abstractmethod
    def write(self, msg):
        """Write a message

        To implement. Do not forget to add a newline ;)
        """

        raise NotImplementedError()


    def overwrite(self, msg):
        """Overwrite the preceding message

        Usefull for interactive shells.
        By default, use write(). To implement.
        """

        self.write(msg)


    def drawProgressBar(self, ratio):
        return (
            '['
            + int(ratio * 50) * '-'
            + (int(ratio) < 1) * '>'
            + (50 - int(ratio * 50)) * ' '
            + ']'
        )


    def onProcessusResume(self, event):
        if event.event_name == 'grading.progress':
            self.count_ia = len(event.grading)

    def onProcessusStart(self, event):
        self.write('Processus {} starts!'.format(event.processus_id))
        self.write(
            'Processus parameters: {} populations of {} individuals are doing to be generated.'
            .format(event.generations, event.pop_length)
        )
        self.write(
            'Selection parameters: selects {}% of the population whose {}% are random.'
            .format(self._percent(event.proportion), self._percent(event.chance))
        )

    def onProcessusDone(self, event):
        self.write('Processus {} is done!'.format(event.processus_id))

    def onCreationStart(self, event):
        self.write('- Creates the initial population...')

    def onCreationDone(self, event):
        self.overwrite('- Initial population created.')

    def onGenerationStart(self, event):
        self.write('- Starts generation {}:'.format(event.generation_id))

    def onGenerationDone(self, event):
        self.write('    Generation {} is done.'.format(event.generation_id))

    def onSelectionStart(self, event):
        self.write('    Starts selection.')

    def onSelectionDone(self, event):
        self.write('    Selection done.')

    def onGradingStart(self, event):
        self.write('    Start grading...')

        self.count_ia = 0

    def onGradingProgress(self, event):
        self.count_ia += 1

        self.overwrite('    Grading: {} IA {} gets a score of {}.'.format(
            self.drawProgressBar(self.count_ia / event.pop_length),
            event.individual.id, event.graduation.score
        ))

    def onGradingDone(self, event):
        self.overwrite('    Grading done.')

    def onBreedingStart(self, event):
        self.write('    Starts breeding.')

        self.count_ia = 0

    def onBreedingProgress(self, event):
        self.count_ia += 1

        self.overwrite('    Breeding: {} {} + {} -> {}.'.format(
            self.drawProgressBar(self.count_ia / event.pop_length),
            event.parents[0].id, event.parents[1].id, event.offspring.id
        ))

    def onBreedingDone(self, event):
        self.overwrite('    Breeding done. ')


    def _percent(self, ratio):
        return int(100 * ratio)
```

```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from shutil import get_terminal_size

from .AbstractLogger import AbstractLogger
from lib.inherit_docstring import inherit_docstring
from src.meta.ABCInheritableDocstringsMeta import ABCInheritableDocstringsMeta


class ConsoleLogger(AbstractLogger, metaclass=ABCInheritableDocstringsMeta):
    """Log Generator events into a file """

    def __init__(self):
        self.first_line = False
        self.last_line = False

    def onProcessusResume(self, event):
        self.first_line = True
        super().onProcessusResume(event)

    def onProcessusStart(self, event):
        self.first_line = True
        super().onProcessusStart(event)

    def onProcessusDone(self, event):
        self.last_line = True
        super().onProcessusDone(event)


    @inherit_docstring
    def write(self, msg):
        msg = '  ' + msg
        length = get_terminal_size()[0]
        msg = msg[:length]
        print(('\n' if not self.first_line else '') + msg, end=('' if not self.last_line else '\n'), flush=True)
        self.first_line = False


    @inherit_docstring
    def overwrite(self, msg):
        msg = '  ' + msg
        length = get_terminal_size()[0]
        msg = msg[:length]
        print('\r' + msg + (length-len(msg)) * ' ', end='', flush=True)
```

06/14/17 05:37:10 /home/zz/Documents/TIPE/src/Logger/FileLogger.py

```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from .AbstractLogger import AbstractLogger
from lib.inherit_docstring import inherit_docstring
from src.meta.ABCInheritableDocstringsMeta import ABCInheritableDocstringsMeta
from src.Writer.PathManager import PathManager


class FileLogger(AbstractLogger, metaclass=ABCInheritableDocstringsMeta):
    """Log Generator events into a file """


    def onProcessusResume(self, event):
        self.processus_id = event.processus_id
        super().onProcessusResume(event)

    def onProcessusStart(self, event):
        self.processus_id = event.processus_id
        super().onProcessusStart(event)


    @inherit_docstring
    def write(self, msg):
        with PathManager.getPath(self.processus_id).with_name('log').open('a') as f:
            f.write(msg + '\n')


    @inherit_docstring
    def overwrite(self, msg):
        with PathManager.getPath(self.processus_id).with_name('log').open('r') as f:
           lines = f.readlines()
        with PathManager.getPath(self.processus_id).with_name('log').open('w') as f:
            f.writelines([item for item in lines[:-1]])
            f.write(msg + '\n')
```

Étape 7 : L'application utilisable en ligne de commande

- */src/app.py*