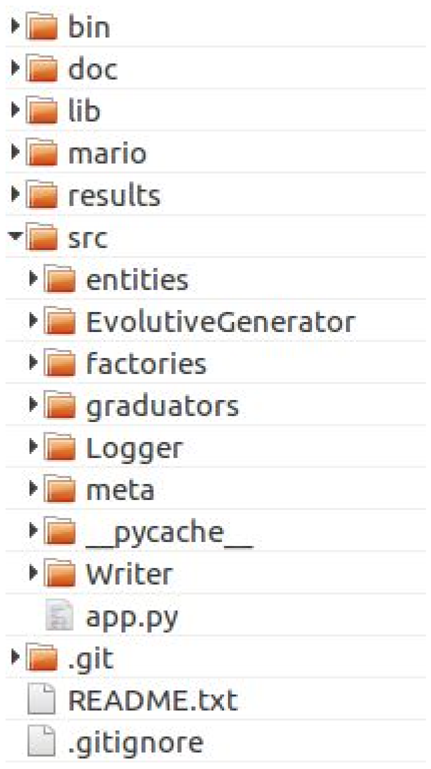
Annexe

Arborescence globale du projet

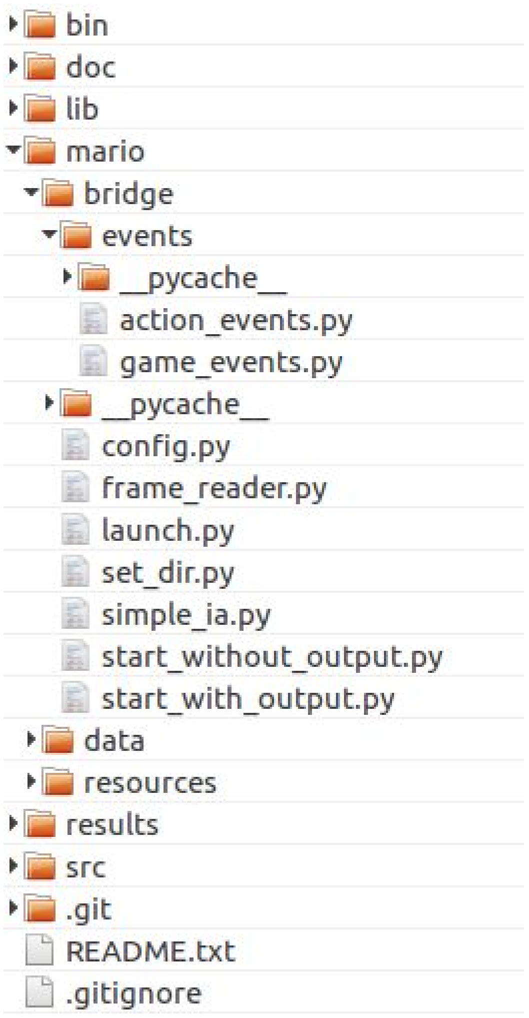


```
▸ bin
▸ doc
▸ lib
▸ mario
▸ results
▾ src
    ▸ entities
    ▸ EvolutiveGenerator
    ▸ factories
    ▸ graduators
    ▸ Logger
    ▸ meta
    ▸ __pycache__
    ▸ Writer
      app.py
▸ .git
  README.txt
  .gitignore
```

Les codes sources présents dans cette annexe ont été rassemblés par thèmes.

Étape 1 : Adapter le jeu

<u>Dossier /mario/bridge</u>



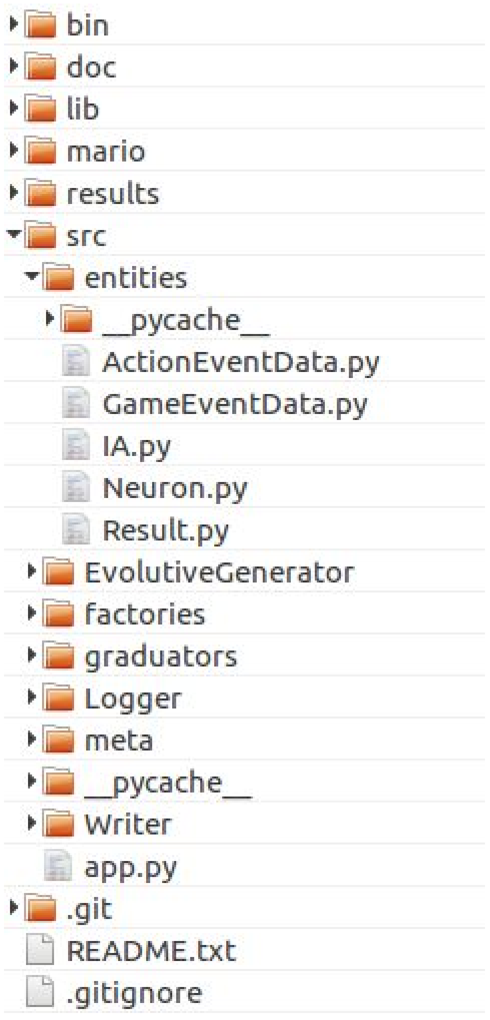
```
bin
doc
lib
mario
  bridge
    events
      __pycache__
      action_events.py
      game_events.py
    __pycache__
    config.py
    frame_reader.py
    launch.py
    set_dir.py
    simple_ia.py
    start_without_output.py
    start_with_output.py
  data
  resources
results
src
.git
README.txt
.gitignore
```

- Modifications du jeu non incluses ici (5000 lignes de codes dans 29 fichiers).
- *FrameReader* : composant chargé de lire ce qui arrive en jeu et de le traduire en évènements compréhensibles par les intelligences artificielles.

```python
import pygame as pg
from lib.inject_arguments import inject_arguments

from mario.bridge.events.game_events import *


class FrameReader:
    """Read the Frame event and make other game events """

    @inject_arguments
    def __init__(self, event_dispatcher):
        self.event_dispatcher.listen('game.frame', self.handle_frame)
        self.frame = None

    @inject_arguments
    def handle_frame(self, frame):
        """Handle Frame event"""

        self.build_events('game.block', Block,
            ['brick_group', 'coin_box_group', 'ground_group', 'pipe_group', 'step_group'], frame)
        self.build_events('game.enemy', Enemy, ['enemy_group'], frame)
        self.build_events('game.powerup', Powerup, ['powerup_group'], frame)
        self.build_events('game.coin', Coin, ['coin_group'], frame)


    def build_events(self, event_name, event_class, groups, frame):
        """Build DetectedComponent game event for each displayed sprite of the groups  """

        sprites = []
        for group in groups:
            sprites.extend(frame.sprite_groups[group].sprites())
        viewport_sprite = ViewportSprite(frame.viewport)
        displayed_sprites = pg.sprite.spritecollide(viewport_sprite, sprites, False)

        # Make the events and dispatch
        for block in displayed_sprites:
            self.event_dispatcher.dispatch(event_name, event_class(block.rect, frame.mario.rect, frame.current_frame))
            # print('game.block', Block(block.rect, frame.mario.rect, frame.current_frame))

        #debug
        pg.display.set_caption("Displayed blocks = " + str(len(displayed_sprites)))


class ViewportSprite(pg.sprite.Sprite):
    """A false sprite containing viewport """

    def __init__(self, viewport_rect):
        pg.sprite.Sprite.__init__(self)
        self.rect = viewport_rect
```

Étape 2 : Modéliser les intelligences avec des *GeneticElement*

Dossier /src/entities



```
▸ bin
▸ doc
▸ lib
▸ mario
▸ results
▾ src
  ▾ entities
    ▸ __pycache__
      ActionEventData.py
      GameEventData.py
      IA.py
      Neuron.py
      Result.py
  ▸ EvolutiveGenerator
  ▸ factories
  ▸ graduators
  ▸ Logger
  ▸ meta
  ▸ __pycache__
  ▸ Writer
    app.py
▸ .git
  README.txt
  .gitignore
```

- *GeneticElement*
- *IA*
- *Neuron*
- *GameEvent*
- *ActionEvent*

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

from abc import ABCMeta


class GeneticElement(metaclass=ABCMeta):
    """Carry the genetic information

    This is an abstract class to inherit.
    A genetic element carries one or several genetic informations or contains
    other genetic elements.

    Evolution logic is handled by an external GeneticElementFactory.
    """

    pass
```

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

from lib.inject_arguments import inject_arguments
from lib.XMLRepr import XMLRepr

from src.EvolutiveGenerator.GeneticElement import GeneticElement


class IA(GeneticElement, XMLRepr):
    """An IA"""

    @inject_arguments
    def __init__(self, id, neurons=list()):
        """Init the IA

        Expects:
            id to be a integer, unique among IA's of a processus
            neurons to be a list of Neuron
        """

        if not type(neurons) is list:
            raise ValueError('Neurons should be a list. ')


    def reprJSON(self):
        return self.__dict__

    def __repr__(self):
        return super().__repr__(displaySequencesNames=False)
```

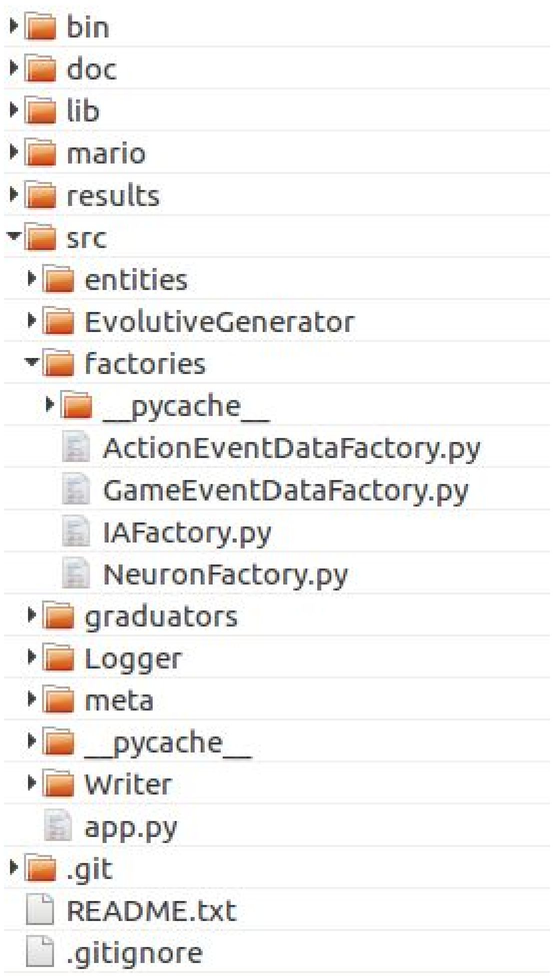
```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from lib.inject_arguments import inject_arguments
from lib.XMLRepr import XMLRepr

from src.EvolutiveGenerator.GeneticElement import GeneticElement
from src.entities.GameEventData import GameEventData
from src.entities.ActionEventData import ActionEventData


class Neuron(GeneticElement, XMLRepr):
    """A link between an game event and an action event """

    @inject_arguments
    def __init__(self, game_event_data, action_event_data):
        """Init the neuron

        Expects:
            game_event_data to be a GameEventData or a tuple (event_name, coor)
            action_event_data to be a ActionEventData or a tuple (action_class, duration)
        """

        if type(game_event_data) is tuple:
            self.game_event_data = GameEventData(*game_event_data)
        if type(action_event_data) is tuple:
            self.action_event_data = ActionEventData(*action_event_data)


    def event_dispatcher():
        doc = "The event_dispatcher property. "

        def fget(self):
            return self._event_dispatcher

        def fset(self, event_dispatcher):
            """Register the neuron to the event_dispatcher """
            if hasattr(self, 'listener_id'):
                del self.event_dispatcher

            self._event_dispatcher = event_dispatcher
            self.listener_id = self._event_dispatcher.listen(self.game_event_data.event_name, self.onEvent)

        def fdel(self):
            """Detach the listener"""
            if hasattr(self, 'listener_id'):
                self._event_dispatcher.detach(self.listener_id)
                del self.listener_id

            del self._event_dispatcher
        return locals()
    event_dispatcher = property(**event_dispatcher())

    def __del__(self):
        if hasattr(self, 'event_dispatcher'):
            del self.event_dispatcher


    def onEvent(self, event):
        if self.game_event_data.checkCoor(event):
            self.event_dispatcher.dispatch('action', self.action_event_data.buildAction(event))


    def reprJSON(self):
        return {
            'game_event_data': self.game_event_data,
            'action_event_data': self.action_event_data
        }

    def __repr__(self):
        return super().__repr__(['game_event_data', 'action_event_data'])
```

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

from lib.inject_arguments import inject_arguments
from lib.XMLRepr import XMLRepr

from src.EvolutiveGenerator.GeneticElement import GeneticElement


class GameEventData(GeneticElement, XMLRepr):
    """An game event data, part of a neuron """

    @inject_arguments
    def __init__(self, event_name, coor):
        pass


    def checkCoor(self, event):
        return self.coor['x'] >= event.left and self.coor['x'] <= event.right \
            and self.coor['y'] >= event.top and self.coor['y'] <= event.bottom


    def reprJSON(self):
        return self.__dict__

    def __repr__(self):
        return super().__repr__(
            attributes=['event_name', 'coor'],
            __dict__={'event_name': self.event_name, 'coor': (self.coor['x'], self.coor['y'])}
        )
```

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

from lib.inject_arguments import inject_arguments
from lib.XMLRepr import XMLRepr

from src.EvolutiveGenerator.GeneticElement import GeneticElement


class ActionEventData(GeneticElement, XMLRepr):
    """An action event data, part of a neuron """

    @inject_arguments
    def __init__(self, action_class, duration):
        pass


    def buildAction(self, event):
        return self.action_class(self.duration, event.current_frame)


    def reprJSON(self):
        return {
            'action_class': self.action_class.__name__,
            'duration': self.duration
        }

    def __repr__(self):
        return super().__repr__(__dict__=self.reprJSON())
```

Étape 3 : Manipuler les *GeneticElement* avec les *GeneticElementFactory*

Dossier /src/factories



```
▸ bin
▸ doc
▸ lib
▸ mario
▸ results
▾ src
    ▸ entities
    ▸ EvolutiveGenerator
    ▾ factories
        ▸ __pycache__
          ActionEventDataFactory.py
          GameEventDataFactory.py
          IAFactory.py
          NeuronFactory.py
    ▸ graduators
    ▸ Logger
    ▸ meta
    ▸ __pycache__
    ▸ Writer
      app.py
▸ .git
  README.txt
  .gitignore
```

- *GeneticElementFactory*
- *IAFactory*
- *NeuronFactory*
- *GameEventFactory*
- *ActionEventFactory*

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

from abc import ABCMeta, abstractmethod


class GeneticElementFactory (metaclass=ABCMeta):
    """Handle the evolution logic of a GeneticElement

    This is an abstract class to inherit.
    This is a static class.
    It brings the evolution logic of the GeneticElement through the following
    class methods:
        +create() -> GeneticElement
        +mutate(GeneticElement)
        +combine(GeneticElement, GeneticElement) -> GeneticElement
        +breed(GeneticElement, GeneticElement) -> GeneticElement
    Evolution logic may typically use recursive process over children of elements.
    """

    @property
    @abstractmethod
    def genetic_element_class (self):
        """The GeneticElement based class """

        raise NotImplementedError


    @staticmethod
    @abstractmethod
    def create(parent = None, children = [], cascade = True):
        """Create a GeneticElement from void

        An essential element of the generation proccess.
        This is a static method which has to be implemented.

        return GeneticElement
        """

        raise NotImplementedError


    @staticmethod
    @abstractmethod
    def mutate(element):
        """Operates a genetic mutation

        This is a static method which has to be implemented.

        This is rather designed for internal use, see generate() instead.
        """

        raise NotImplementedError


    @staticmethod
    def combine(element1, element2):
        """Form a new GeneticElement, combination of two ones

        Combine two GeneticElement to form an offspring.
        This is a static method which has to be implemented.

        This is rather designed for internal use, see generate() instead.

        Expects:
            element1, element2 to be GeneticElement's

        return GeneticElement
        """

        raise NotImplementedError


    @classmethod
    def breed(cls, element1, element2):
        """Generate a new GeneticElement, final offspring of two ones

        Call combine() then mutate().
        This is a class method.

        Expects:
            element1, element2 to be a GeneticElement's

        return GeneticElement
        """

        new_element  = cls.combine(element1, element2)
        cls.mutate(new_element )
        return new_element
```

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

from random import randint, random
from math import ceil
from copy import deepcopy

from lib.inherit_docstring import inherit_docstring
from src.meta.ABCInheritableDocstringsMeta import ABCInheritableDocstringsMeta
from src.EvolutiveGenerator.GeneticElementFactory import GeneticElementFactory
from src.entities.IA import IA
from src.factories.NeuronFactory import NeuronFactory


randindex = lambda it: randint(0, len(it)-1)
def randindex_safe(it):
    if len(it) < 3:
        raise ValueError("Iterable should have a least 3 elements. ")
    return randint(1, len(it)-2)


class IAFactory(GeneticElementFactory, metaclass=ABCInheritableDocstringsMeta):
    """IA factory"""

    @property
    @inherit_docstring
    def genetic_element_class(self):
        return IA

    last_ia_id = -1

    @classmethod
    def onProcessusStart(cls, event):
        cls.last_ia_id = -1

    @classmethod
    def newIaId(cls):
        cls.last_ia_id += 1
        return cls.last_ia_id

    @classmethod
    def updateIaId(cls, ia_id):
        cls.last_ia_id = max(cls.last_ia_id, ia_id)


    @classmethod
    @inherit_docstring
    def create(cls):
        neurons = list()
        for i in range(3 + randint(0, 3)):
            neurons.append(NeuronFactory.create())
        return IA(cls.newIaId(), neurons)


    @staticmethod
    @inherit_docstring
    def mutate(element):
        if random() < .2:
            element.neurons.insert(randindex(element.neurons), NeuronFactory.create())
        if random() < .1 and len(element.neurons) > 3:
            element.neurons.pop(randindex(element.neurons))
        for neuron in element.neurons:
            if random() < .2:
                NeuronFactory.mutate(neuron)


    @classmethod
    @inherit_docstring
    def combine(cls, element1, element2):
        neurons = element1.neurons[:randindex_safe(element1.neurons)] + element2.neurons[randindex_safe(element2.neurons):]

        # Ensure you have a least 3 neurons
        if len(neurons) < 3:
            return cls.combine(element1, element2)

        # Duplicate neurons instead of reuse ones
        neurons = [deepcopy(neuron) for neuron in neurons]
        return IA(cls.newIaId(), neurons)


    @classmethod
    def hydrate(cls, data):
        cls.updateIaId(data['id'])

        return IA(data['id'], [ NeuronFactory.hydrate(neuron_data) for neuron_data in data['neurons'] ])
```

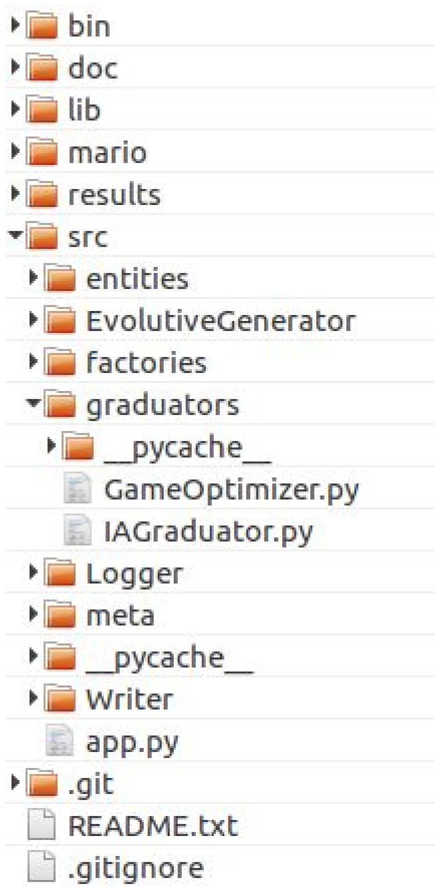
```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from lib.inherit_docstring  import inherit_docstring
from random import randint

from src.meta.ABCInheritableDocstringsMeta  import ABCInheritableDocstringsMeta
from src.EvolutiveGenerator .GeneticElementFactory  import GeneticElementFactory
from src.entities.Neuron import Neuron
from src.factories.GameEventDataFactory  import GameEventDataFactory
from src.factories.ActionEventDataFactory  import ActionEventDataFactory


class NeuronFactory (GeneticElementFactory, metaclass=ABCInheritableDocstringsMeta):
    """Neuron factory"""

    @property
    @inherit_docstring
    def genetic_element_class (self):
        return Neuron


    @staticmethod
    @inherit_docstring
    def create():
        return Neuron(GameEventDataFactory .create(), ActionEventDataFactory .create())


    @staticmethod
    @inherit_docstring
    def mutate(element):
        if randint(0, 1):
            GameEventDataFactory .mutate(element.game_event_data )
        else:
            ActionEventDataFactory .mutate(element.action_event_data )


    @staticmethod
    def hydrate(data):
        return Neuron(
            GameEventDataFactory .hydrate(data['game_event_data ']),
            ActionEventDataFactory .hydrate(data['action_event_data '])
        )
```

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

from lib.inherit_docstring import inherit_docstring
from random import choice, randint

from src.meta.ABCInheritableDocstringsMeta import ABCInheritableDocstringsMeta
from mario.data.constants import SCREEN_HEIGHT, SCREEN_WIDTH, GROUND_HEIGHT
from src.EvolutiveGenerator.GeneticElementFactory import GeneticElementFactory
from src.entities.GameEventData import GameEventData


class GameEventDataFactory(GeneticElementFactory, metaclass=ABCInheritableDocstringsMeta):
    """GameEventData factory"""

    @property
    @inherit_docstring
    def genetic_element_class(self):
        return GameEventData

    GAME_EVENT_NAMES = ('game.block', 'game.enemy', 'game.powerup', 'game.coin')

    MIN_X = -int(SCREEN_WIDTH / 2) # max left
    MAX_X = SCREEN_WIDTH # max right
    MIN_Y = -GROUND_HEIGHT # max top
    MAX_Y = SCREEN_HEIGHT # max bottom


    @classmethod
    @inherit_docstring
    def create(cls):
        return GameEventData(cls.createEventName(), cls.createCoor())

    @classmethod
    @inherit_docstring
    def mutate(cls, element):
        if randint(0, 1):
            element.event_name = cls.createEventName()
        else:
            element.coor = cls.mutateCoor(element.coor)


    @staticmethod
    def hydrate(data):
        return GameEventData(**data)


    @classmethod
    def createEventName(cls):
        if randint(0, 9):
            return cls.GAME_EVENT_NAMES[0]
        return choice(cls.GAME_EVENT_NAMES[1:])

    @classmethod
    def createCoor(cls):
        return {
            'x': randint(cls.MIN_X, cls.MAX_X),
            'y': randint(cls.MIN_Y, cls.MAX_Y)
        }

    @classmethod
    def mutateCoor(cls, coor):
        coor['x'] += randint(-100, 100)
        coor['y'] += randint(-100, 100)

        if coor['x'] < cls.MIN_X:
            coor['x'] = cls.MIN_X
        if coor['x'] > cls.MAX_X:
            coor['x'] = cls.MAX_X
        if coor['y'] < cls.MIN_Y:
            coor['y'] = cls.MIN_Y
        if coor['y'] > cls.MAX_Y:
            coor['y'] = cls.MAX_Y
        return coor
```

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

from lib.inherit_docstring  import inherit_docstring
from lib.choices import choices
from lib.gauss_int import gauss_int
from random import randint

from src.meta.ABCInheritableDocstringsMeta  import ABCInheritableDocstringsMeta
from mario.bridge.events.action_events  import Jump, Left, Right, Down, Fire
from src.EvolutiveGenerator .GeneticElementFactory  import GeneticElementFactory
from src.entities.ActionEventData  import ActionEventData


class ActionEventDataFactory (GeneticElementFactory, metaclass=ABCInheritableDocstringsMeta):
    """ActionEventData factory """

    @property
    @inherit_docstring
    def genetic_element_class (self):
        return ActionEventData

    ACTION_CLASSES  = (Jump, Left, Right, Down, Fire)


    @classmethod
    @inherit_docstring
    def create(cls):
        action_class  = cls.createActionClass ()
        return ActionEventData (action_class , cls.createDuration (action_class ))

    @classmethod
    @inherit_docstring
    def mutate(cls, element):
        if randint(0, 1):
            element .action_class  = cls.createActionClass ()
        else:
            element .duration  = cls.createDuration (element .action_class )


    @classmethod
    def hydrate(cls, data):
        for action_class  in cls.ACTION_CLASSES :
            if action_class .__name__  == data['action_class ']:
                return ActionEventData (action_class , data['duration '])
        return ValueError("Action class  {} doesn't exist. ".format(data['action_class ']))


    @classmethod
    def createActionClass (cls):
        return choices(cls.ACTION_CLASSES , weights=[35, 10, 35, 10, 10])[0]

    @staticmethod
    def createDuration (action_class):
        if action_class  == Jump:
            return gauss_int (2, 38)
        else:
            return randint (0, 25)
```

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.

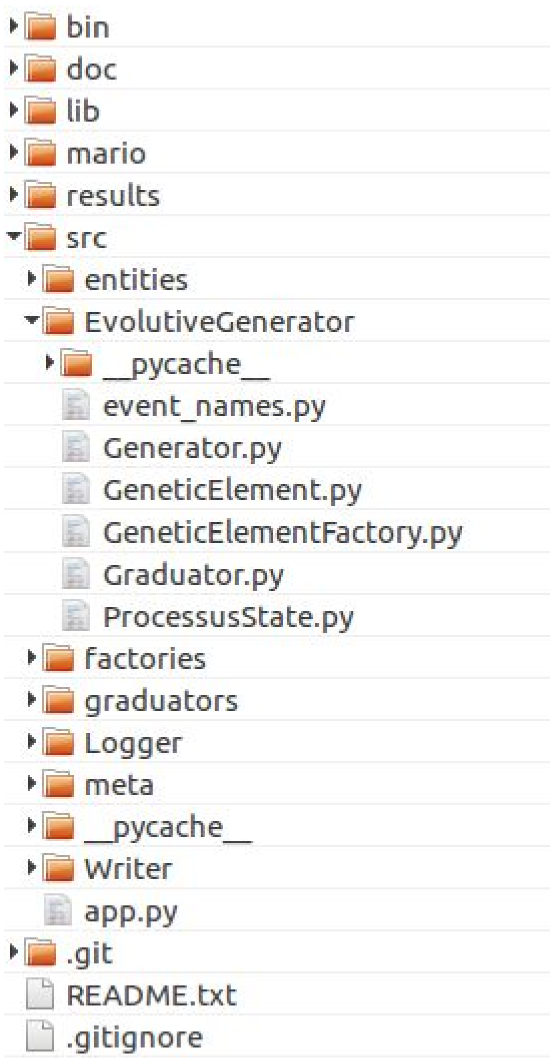
```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

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
```

```python
#!/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)
```

```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



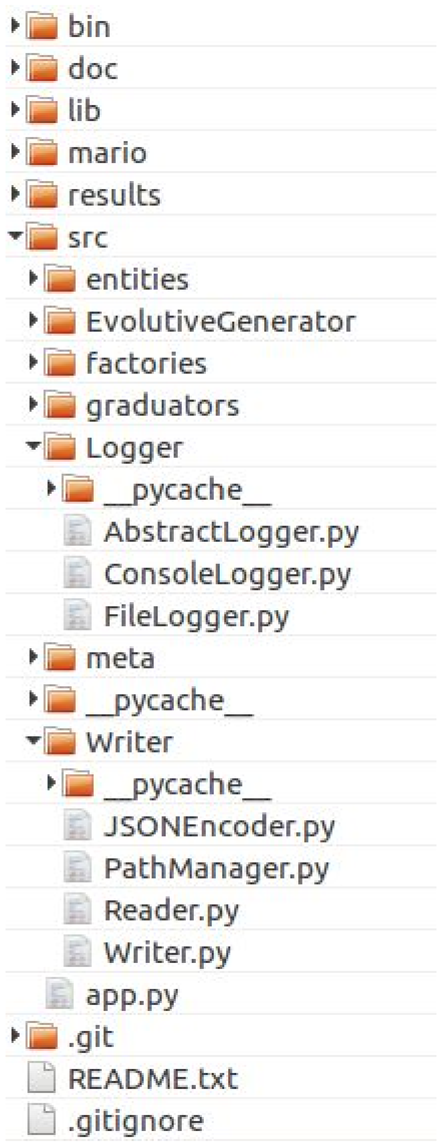
```
bin
doc
lib
mario
results
src
    entities
    EvolutiveGenerator
        __pycache__
        event_names.py
        Generator.py
        GeneticElement.py
        GeneticElementFactory.py
        Graduator.py
        ProcessusState.py
    factories
    graduators
    Logger
    meta
    __pycache__
    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):
        """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*

```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
```

```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)
```

```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')
```

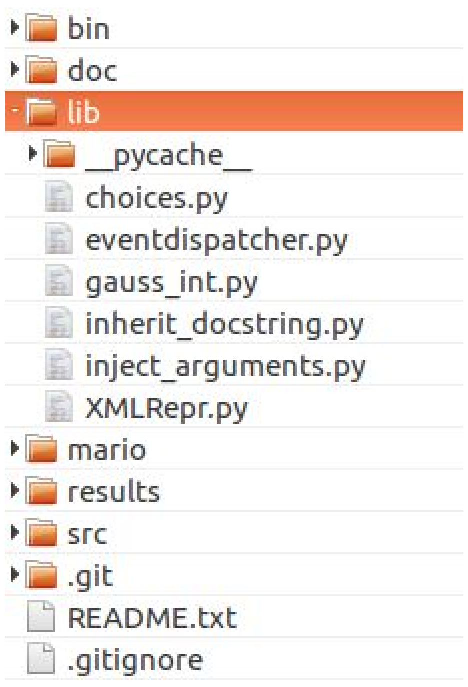
Étape 7 : L'application utilisable en ligne de commande

- */src/app.py*

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

from argparse import ArgumentParser
from math import inf
from time import time

from lib.eventdispatcher import EventDispatcher
from mario.bridge.frame_reader import FrameReader
from mario.bridge.config import Config
from .EvolutiveGenerator.Generator import Generator
from .factories.IAFactory import IAFactory
from .graduators.IAGraduator import IAGraduator
from .graduators.GameOptimizer import GameOptimizer
from .Writer.Writer import Writer
from .Logger.FileLogger import FileLogger
from .Logger.ConsoleLogger import ConsoleLogger
from .Writer.PathManager import PathManager
from .Writer.Reader import Reader


def instanciateGenerator(show):
    event_dispatcher = EventDispatcher()
    FrameReader(event_dispatcher)
    GameOptimizer(event_dispatcher)
    return Generator(IAFactory, IAGraduator(event_dispatcher, show), [Writer(), FileLogger(), ConsoleLogger()],
        lambda state: True in [score.percent >= 1. for score, individual in state.grading]
    )

def checkProcessusExists(processus_id):
    if not Reader.processusExists(processus_id):
        raise ValueError("Processus with id={} doesn't exist.".format(processus_id))

def new(args):
    """New processus"""
    population = instanciateGenerator(args.show).process(
        PathManager.newProcessusId(), args.generations, args.pop_length, args.proportion, args.chance
    )

def resume(args):
    """Resume a processus"""
    checkProcessusExists(args.processus_id)
    population = instanciateGenerator(args.show).resume(Reader.getProcessusState(args.processus_id))

def play(args):
    """Play the best individual of a processus' last generation """
    checkProcessusExists(args.processus_id)
    # Get IA
    if args.ia_id is None:
        ia, generation_id = Reader.getBestIa(args.processus_id, args.generation_id)
        print('The best AI is {}.'.format(ia.id), flush=True)
    else:
        ia, generation_id = Reader.getIa(args.processus_id, args.ia_id)
    # Play IA
    event_dispatcher = EventDispatcher()
    FrameReader(event_dispatcher)
    graduator = IAGraduator(event_dispatcher, show=True)
    if args.as_grading:
        print(
            "Attention : Malgré que le visionnage présenté soit le plus proche possible des conditions d'évaluation, des aléas subsistent. "
            "Si vous cherchez à visionner une performance difficile à reproduire, n'hésitez pas à rééssayer plusieurs fois. "
        , flush=True)
        GameOptimizer(event_dispatcher)
        graduator.grade(ia, generation_id)
    else:
        graduator.gradeIAWithConfig(ia, Config(True, event_dispatcher))

def print_data(args):
    checkProcessusExists(args.processus_id)

    data = Reader.getData(args.processus_id)
    txt1 = 'Générations,Scores des intelligences'
    for generation_id, grading in data:
        txt1 += '\n' + str(generation_id)
        for result, ia_id in grading:
            txt1 += ',' + str(result['score'])
    txt2 = 'Générations,Scores des intelligences'
    for generation_id, grading in data:
        txt2 += '\n' + str(generation_id)
        for result, ia_id in grading:
            txt2 += ',' + str(result['max_x'])

    path1 = PathManager.getPath(args.processus_id, read_only=True).parent / 'data' / (str(time()) + '.score.csv')
    path2 = PathManager.getPath(args.processus_id, read_only=True).parent / 'data' / (str(time()) + '.distance.csv')
    PathManager.makeDir(path1.parent)
    path1.write_text(txt1)
    path2.write_text(txt2)


# Build parser
parser = ArgumentParser()
subparsers = parser.add_subparsers()

new_parser = subparsers.add_parser('new')
new_parser.add_argument('pop_length', type=int)
new_parser.add_argument('--generations', default=inf, type=int)
new_parser.add_argument('--proportion', default=0.5, type=float)
new_parser.add_argument('--chance', default=0, type=float)
new_parser.add_argument('--show', dest='show', action='store_true')
new_parser.set_defaults(command=new, show=False)

resume_parser = subparsers.add_parser('resume')
resume_parser.add_argument('processus_id', type=int)
resume_parser.add_argument('--show', dest='show', action='store_true')
resume_parser.set_defaults(command=resume, show=False)

play_parser = subparsers.add_parser('play')
play_parser.add_argument('processus_id', type=int)
play_parser.add_argument('--generation_id', type=int)
play_parser.add_argument('--ia_id', type=int)
play_parser.add_argument('--as_grading', dest='as_grading', action='store_true')
play_parser.set_defaults(command=play, as_grading=False)

print_parser = subparsers.add_parser('print')
print_parser.add_argument('processus_id', type=int)
print_parser.set_defaults(command=print_data)

# Parse arguments
args = parser.parse_args()
if hasattr(args, 'command'):
    args.command(args)
else:
    print('No command given, use --help')
```

Bibliothèques utilisées

Dossier /lib



```
bin
doc
lib
    __pycache__
    choices.py
    eventdispatcher.py
    gauss_int.py
    inherit_docstring.py
    inject_arguments.py
    XMLRepr.py
mario
results
src
.git
README.txt
.gitignore
```

- *EventDispatcher* (créé par moi sur d'autres projets)
- *XMLRepr* (créé par moi pour l'occasion)
- *inject_arguments* (créé par moi pour l'occasion)
- *inherit_doctring* (pris sur Internet)
- *gauss_int et choices* (créé par moi pour l'occasion)

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


# The MIT License (MIT)
#
# Copyright (c) 2015-2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#


import re


class EventDispatcher:
    '''
    A simple event dispatcher

    Author: Rémi Blaise (alias Zzortell) "http://php-zzortell.rhcloud.com/"

    '''


    def __init__(self, propagation=False):
        '''
        Init the event dispatcher

        Parameter:
        {bool} propagation = False If dispatching an event should also dispatch its parents

        '''

        self.propagation = propagation
        self.listeners = {}


    def listen(self, name, listener, priority=0):
        '''
        Add an event listener

        If name is 'all', the listener will listen all events.

        Parameters:
        {str}          name              The name of the event
        {function}     listener          The event listener
        {int}        priority = 0     The priority of the listener

        Return: {tuple} id The ID of the listener

        '''

        # Register listener
        if name not in self.listeners:
            self.listeners[name] = {}
        if priority not in self.listeners[name]:
            self.listeners[name][priority] = []
        self.listeners[name][priority].append(listener)

        return (name, priority, listener)


    def on(self, name):
        '''Inscribe given listener, to use as decorator '''
        def decorator(function):
            self.listen(name, function)
            return function
        return decorator


    def detach(self, id):
        '''
        Detach an event listener

        Parameter:
        {tuple} id The ID of the listener

        '''

        name, priority, listener  = id
        self.listeners[name][priority].remove(listener)


    def dispatch(self, name, event=None, propagation=None):
        '''
        Dispatch an event

        If propagation is set, dispatch all the parent events.

        Parameters:
        {str}         name                 The name of the event
        {object}     event = None         The event to dispatch
        {bool}         propagation = None     Override self.propagation

        '''

        if name == 'all':
            raise ValueError("'all' is a reserved keyword, not an event name.  ")
        propagation = propagation if propagation is not None else self.propagation

        # Get existing keys among ('all', name)
        names = []
        if 'all' in self.listeners:
            names.append('all')
        if name in self.listeners:
            names.append(name)

        # Get sorted list of priorities
        priorities = set()
        for name in names:
            priorities = priorities.union(set(self.listeners[name].keys()))
        priorities = list(priorities)
        priorities.sort()

        # Iterate over priorities
        for priority in priorities:
            # Get listeners
            listeners = []
            for name in names:
                if priority in self.listeners[name]:
                    listeners.extend(self.listeners[name][priority])

            # Iterate over listeners
            for listener in listeners:
                listener(event)

        # If propagation dispatch the parent event
        if propagation:
            parent_name = self.getParent(name)
            if parent_name:
                self.dispatch(parent_name, event)


    def getParent(self, name):
        '''
        Get the name of the parent event

        Used if the propagation option is True.
        The event name has to match the format "parent.event".

        Parameters:
        {str} name The name of the event

        Return: {str}     The name of the parent event
                None     If the event has no parent

        '''

        if re.search(r'^(?:\w+\.)*\w+$', name) is None:
            raise AssertionError("The event name has to match with r'^(?:\w+\.)*\w+$'.  ")

        if re.search(r'\.', name):
            return re.search(r'^((?:\w+\.)*)\w+$', name).group(1)[:-1]
        else:
            return None
```

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

# The MIT License (MIT)
#
# Copyright (c) 2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.


from textwrap import indent
from operator import itemgetter , attrgetter


class XMLRepr:
    """
    Awesome XML representation base class

    Inherit to have an XML-like repr of instances.

    __repr__ expects:
        attributes to be a list of attribute names to filter and order.
        __dict__ to be a dict, substitute of self.__dict__
        displayChildrenNames to be a bool.
        displaySequencesNames to be a bool.
        indent_prefix to be a string.

    Features:
        - Use class name as tag name.
        - Use non-XMLRepr non-XMLRepr-containing-sequence attributes as
          attributes: names are used as names and values as values.
        - Use XMLRepr attributes as children, printed as it.
        - Use sequence attributes containing exclusively XMLRepr items as
          children: name is used as tag name and items as children.
        - If attributes is not given, order attributes and children by asc.
        - If displayChildrenNames is set True, children are preceded by their
          attr name. Ex: <brick>: <AwesomeBrick id=0 content='Red mushroom'/>
        - Filter attributes with the attribute names given by attributes parameter.
          Futhermore, it indicates the order of attributes.
        - Substitute self.__dict__ by __dict__.
        - If displaySequencesNames is set False, sequences' children are displayed
          without wrapping.

    Example:
        class MyAwesomeClass(XMLRepr):
            def __init__(self):
                self.color = 'pink'
                self.checked = True
                self.brick = AwesomeBrick(0)
                self.bricks = [AwesomeBrick(1), AwesomeBrick(2)]
        class AwesomeBrick(XMLRepr):
            def __init__(self, id):
                self.content = 'Red mushroom'
                self.id = id

        awesome_object = MyAwesomeClass()
        print(awesome_object)

    Output:
        <MyAwesomeClass color='pink' checked=True>
            <AwesomeBrick id=0 content='Red mushroom'/>
            <bricks>
                <AwesomeBrick id=1 content='Red mushroom'/>
                <AwesomeBrick id=2 content='Red mushroom'/>
            </bricks>
        </MyAwesomeClass>
    """

    def __repr__(self,
            attributes = None, __dict__ = None,
            displayChildrenNames = False, displaySequencesNames = True,
            indent_prefix = '    '
        ):
        if __dict__ is None:
            __dict__ = self.__dict__
        if attributes is None:
            attributes_and_children  = __dict__.items()
        else:
            attributes_and_children  = [(attr, __dict__[attr]) for attr in attributes]
        attributeList  = []
        children  = []
        sequences  = []
        for name, value in attributes_and_children :
            if isinstance(value, XMLRepr):
                if displayChildrenNames:
                    children.append((name, value))
                else:
                    children.append(value)
            elif hasattr(value, '__iter__') and all(isinstance(item, XMLRepr) for item in value):
                sequences.append((name, value))
            else:
                attributeList.append((name, value))

        if attributes is None:
            attributeList.sort(key=itemgetter(0))
            if displayChildrenNames:
                children.sort(key=itemgetter(0))
            else:
                children.sort(key=attrgetter('__class__.__name__'))
            sequences.sort(key=itemgetter(0))

        def formatAttributes(attributeList):
            formatted_attributes = ''
            for name, value in attributeList:
                formatted_attributes += '{}={} '.format(name, repr(value))
            return formatted_attributes.rstrip(' ')

        def formatChildren(children):
            formatted_children = ''
            for value in children:
                formatted_children += '{}\n'.format(repr(value))
            return indent(formatted_children, indent_prefix)

        def formatChildrenWithNames(children):
            formatted_children = ''
            for name, value in children:
                formatted_children += '<{}>: {}\n'.format(name, repr(value))
            return indent(formatted_children, indent_prefix)

        def formatSequences(sequences):
            formatted_sequences = ''
            for name, seq in sequences:
                formatted_sequences += formatChildren(seq)
            return formatted_sequences

        def formatSequencesWithNames(sequences):
            formatted_sequences = ''
            for name, seq in sequences:
                formatted_sequences += '<{0}>\n{1}</{0}>\n'.format(name, formatChildren(seq))
            return indent(formatted_sequences, indent_prefix)

        if children or sequences:
            return '<{0} {1}>\n{2}{3}</{0}>'.format(
                self.__class__.__name__,
                formatAttributes(attributeList),
                formatChildrenWithNames(children) if displayChildrenNames \
                else formatChildren(children),
                formatSequencesWithNames(sequences) if displaySequencesNames \
                else formatSequences(sequences)
            )

        return '<{0} {1}/>'.format(
            self.__class__.__name__,
            formatAttributes(attributeList)
        )


if __name__ == '__main__':
    class MyAwesomeClass(XMLRepr):
        def __init__(self):
            self.color = 'pink'
            self.checked = True
            self.brick = AwesomeBrick(0)
            self.awesome = SuperAwesomeBrick(42)
            self.bricks = [AwesomeBrick(1), AwesomeBrick(2)]
    class AwesomeBrick(XMLRepr):
        def __init__(self, id):
            self.content = 'Red mushroom'
            self.id = id
    class SuperAwesomeBrick(AwesomeBrick):
        pass

    awesome_object = MyAwesomeClass()
    print(69*'-')
    print(awesome_object)

    class DisplayNamesAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(displayChildrenNames=True, indent_prefix='    ')
    print(DisplayNamesAwesomeClass())

    class FilterAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(attributes=['color', 'bricks'], indent_prefix='\t')
    print(FilterAwesomeClass())

    class SubstituteAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(__dict__={'color': 'blood'}, indent_prefix='\t')
    print(SubstituteAwesomeClass())

    class WithoutSequencesNamesAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(displaySequencesNames=False)
    print(WithoutSequencesNamesAwesomeClass ())
    print(69*'-')
```

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


# The MIT License (MIT)
#
# Copyright (c) 2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.


def inject_arguments (in_function):
    """Inject arguments of a method as attributes

    To use as decorator.
    """

    def out_function (*args, **kwargs):
        _self  = args[0]

        # Get all of argument's names of the in_function
        all_names  = in_function .__code__ .co_varnames [1:in_function .__code__ .co_argcount ]

        ## Add default values for non-specified arguments
        defaults  = in_function .__defaults__
        if defaults:
            _self.__dict__ .update (zip (all_names [-len (defaults ):], defaults ))

        ## Add kwargs
        _self.__dict__ .update (kwargs )

        ## Add args
        # Get only the names that don't belong to kwargs
        names  = [n for n in all_names  if not n in kwargs ]
        # Match argument names with values
        _self.__dict__ .update (zip (names , args [1:]))

        return  in_function (*args , **kwargs )

    return  out_function


if __name__=='__main__':
    import unittest

    class ArgumentInjectionTest (unittest.TestCase):
        def test(self):
            class Test:
                @inject_arguments
                def __init__(self, name, surname, default = 'lol'):
                    pass

            t  = Test('mickey', surname='mouse')
            self.assertEqual ('mickey', t.name)
            self.assertEqual ('mouse', t.surname)
            self.assertEqual ('lol', t.default)

        def test_defaultAlone (self):
            class Test:
                @inject_arguments
                def __init__(self, default='lol'):
                    pass

            t  = Test('given')
            self.assertEqual ('given', t.default)

        def test_inheritance (self):
            class A():
                @inject_arguments
                def __init__(self, a1):
                    pass

            class B(A):
                @inject_arguments
                def __init__(self, b1 = None, b2 = None, *args, **kwargs):
                    super().__init__(*args, **kwargs)

            b  = B(0, 1, 2)
            self.assertEqual (0, b.b1)
            self.assertEqual (1, b.b2)
            self.assertEqual (2, b.a1)

        def test_defaultInheritance (self):
```

```python
        class Test:
            @inject_arguments
            def __init__(self, default='lol'):
                pass

        class Child(Test):
            @inject_arguments
            def __init__(self, minus = None, malus = None, *args, **kwargs):
                super().__init__(*args, **kwargs)

        c = Child(1, -1)
        self.assertEqual(1, c.minus)
        self.assertEqual(-1, c.malus)
        self.assertEqual('lol', c.default)

        c = Child(1, -1, 'hey')
        self.assertEqual(1, c.minus)
        self.assertEqual(-1, c.malus)
        self.assertEqual('hey', c.default)

    def test_giveLastDefaultArgument(self):
        class TestLastGivenDefault:
            @inject_arguments
            def __init__(self, default1=1, default2=2):
                pass

        t = TestLastGivenDefault(default2=3)
        self.assertEqual(1, t.default1)
        self.assertEqual(3, t.default2)

unittest.main()
```

```python
"""
Inherit docstrings

Found here: http://code.activestate.com/recipes/578587-inherit-method-docstrings-without-breaking-decorat/

Simple Use:
    1) Import this module
    2) Inherit metaclass InheritableDocstrings
    3) Apply decorator inherit_docstring

Example:
    from lib.inherit_docstring import InheritableDocstrings, inherit_docstring

    class Animal:
        def move_to(self, dest):
            '''Move to *dest*'''
            pass

    class Bird(Animal, metaclass=InheritableDocstrings):
        @inherit_docstring
        def move_to(self, dest):
            self._fly_to(dest)

    assert Animal.move_to.__doc__ == Bird.move_to.__doc__

"""


from functools import partial

# Replace this with actual implementation from
# http://code.activestate.com/recipes/577748-calculate-the-mro-of-a-class/
# (though this will work for simple cases)
def mro(*bases):
    return bases[0].__mro__

# This definition is only used to assist static code analyzers
def inherit_docstring(fn):
    '''Copy docstring for method from superclass

    For this decorator to work, the class has to use the `InheritableDocstrings`
    metaclass.
    '''
    raise RuntimeError('Decorator can only be used in classes '
                       'using the `InheritableDocstrings` metaclass')

def _inherit_docstring(mro, fn):
    '''Decorator to set docstring for *fn* from *mro*'''

    if fn.__doc__ is not None:
        raise RuntimeError('Function already has docstring')

    # Search for docstring in superclass
    for cls in mro:
        super_fn = getattr(cls, fn.__name__, None)
        if super_fn is None:
            continue
        fn.__doc__ = super_fn.__doc__
        break
    else:
        raise RuntimeError("Can't inherit docstring for %s: method does not "
                           "exist in superclass" % fn.__name__)

    return fn

class InheritableDocstrings(type):
    @classmethod
    def __prepare__(cls, name, bases, **kwds):
        classdict = super().__prepare__(name, bases, *kwds)

        # Inject decorators into class namespace
        classdict['inherit_docstring'] = partial(_inherit_docstring, mro(*bases))

        return classdict

    def __new__(cls, name, bases, classdict):

        # Decorator may not exist in class dict if the class (metaclass
        # instance) was constructed with an explicit call to `type`.
        # (cf http://bugs.python.org/issue18334)
        if 'inherit_docstring' in classdict:

            # Make sure that class definition hasn't messed with decorators
            copy_impl = getattr(classdict['inherit_docstring'], 'func', None)
            if copy_impl is not _inherit_docstring:
                raise RuntimeError('No inherit_docstring attribute may be created '
                                   'in classes using the InheritableDocstrings metaclass')

            # Delete decorators from class namespace
            del classdict['inherit_docstring']

        return super().__new__(cls, name, bases, classdict)
```

```python
from math import floor
from random import gauss

def gauss_int(a, b):
    n = b + 1
    while n > b or n < a:
        n = floor(gauss(b, (b-a)))
    return n

if __name__ == '__main__':
    count = [0] * 39
    for i in range(1000000):
        count[gauss_int(0, 38)] += 1
    print(count)


----------------------------------------------------------------------------

"""This is the standard Python 3.6 implementation of choices """

from random import random
import itertools as _itertools
import bisect as _bisect

def choices(population, weights=None, *, cum_weights=None, k=1):
    """Return a k sized list of population elements chosen with replacement.

    If the relative weights or cumulative weights are not specified,
    the selections are made with equal probability.

    """
    if cum_weights is None:
        if weights is None:
            _int = int
            total = len(population)
            return [population[_int(random() * total)] for i in range(k)]
        cum_weights = list(_itertools.accumulate(weights))
    elif weights is not None:
        raise TypeError('Cannot specify both weights and cumulative weights ')
    if len(cum_weights) != len(population):
        raise ValueError('The number of weights does not match the population ')
    bisect = _bisect.bisect
    total = cum_weights[-1]
    return [population[bisect(cum_weights, random() * total)] for i in range(k)]
```