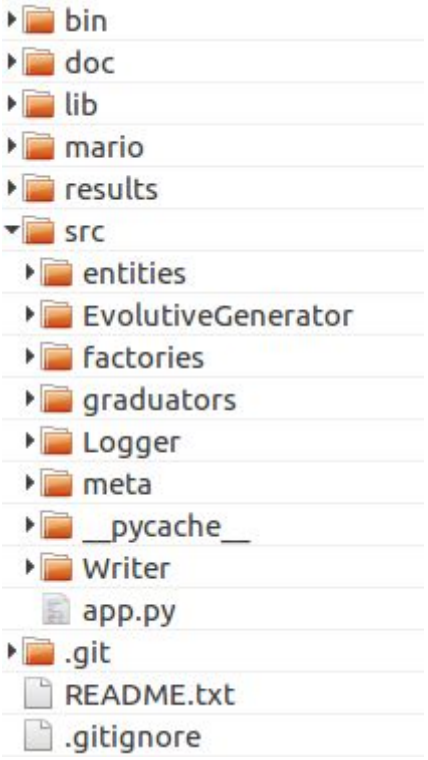
Annexe

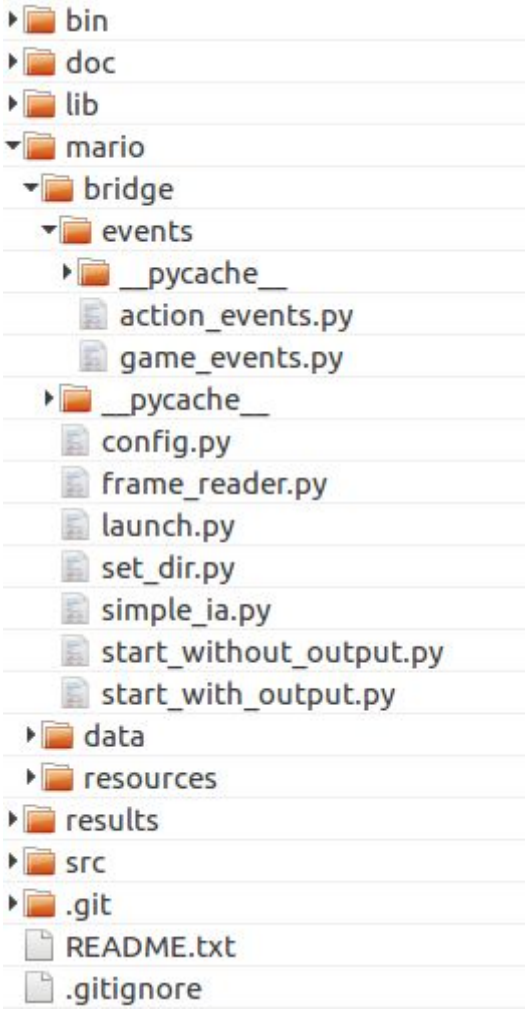
Arborescence globale du projet



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

Étape 1 : Adapter le jeu

Dossier /mario/bridge

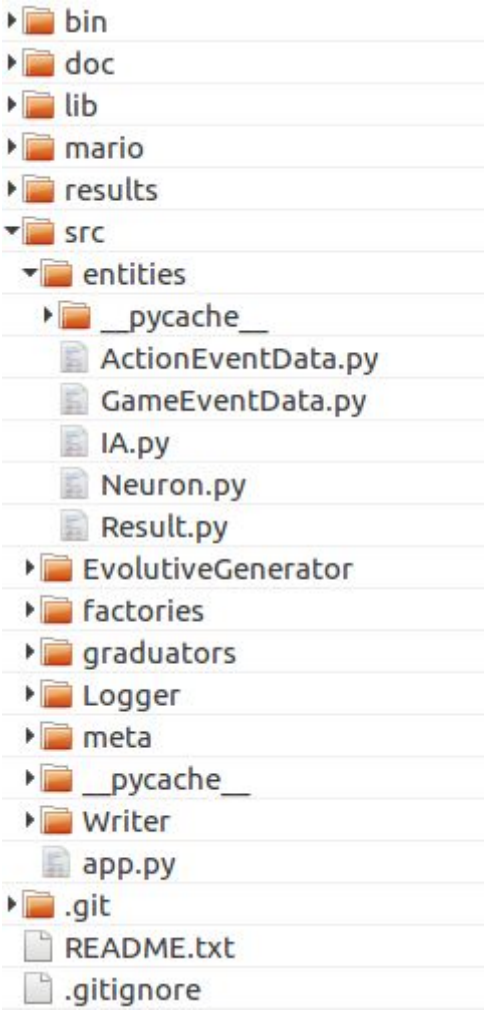


- Modifications du jeu non incluses ici (5000 lignes de codes dans 29 fichiers).
- *FrameReader* : composant chargé de lire ce qui arrive en jeu et de le traduire en évènements compréhensibles par les intelligences artificielles.

```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



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

```
 1  #!/usr/bin/python3.4
 2  # -*-coding:Utf-8 -*
 3
 4  from abc import ABCMeta
 5
 6
 7  class GeneticElement(metaclass=ABCMeta):
 8      """Carry the genetic information
 9
10      This is an abstract class to inherit.
11      A genetic element carries one or several genetic informations or contains
12      other genetic elements.
13
14      Evolution logic is handled by an external GeneticElementFactory.
15      """
16
17      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, ia, 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)
```

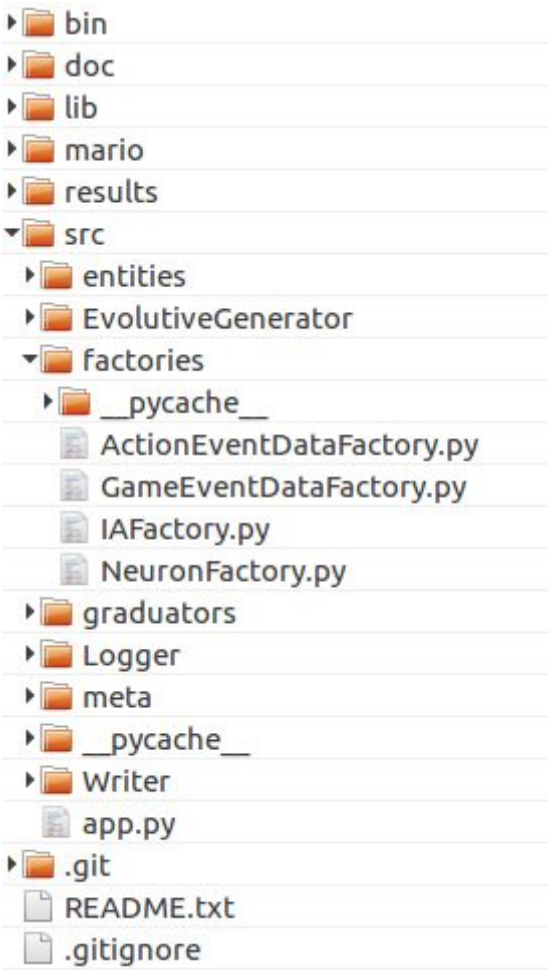
```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

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



- *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_ia):
        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
```

```
#!/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)
```