```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'] ])
```