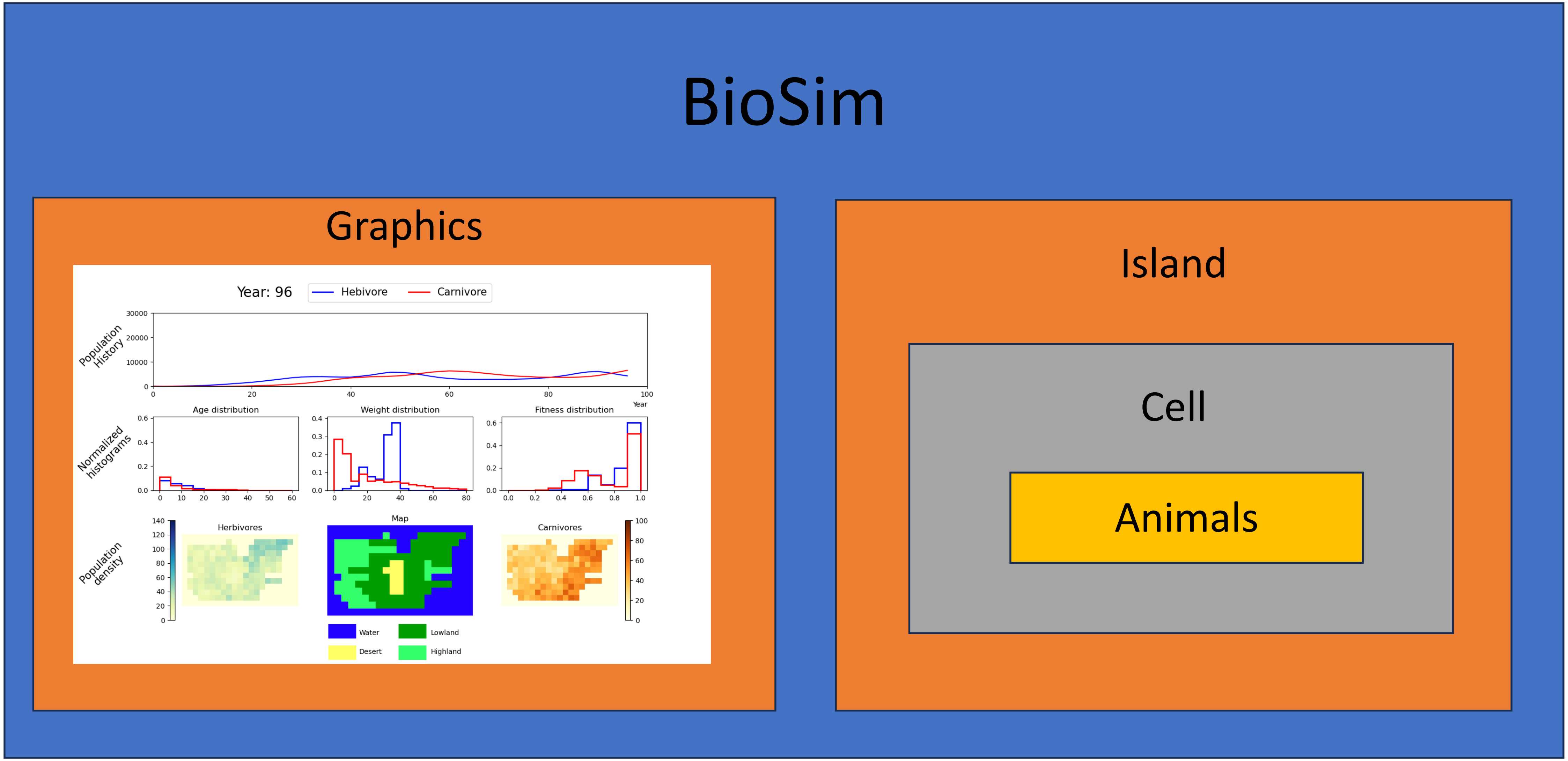
BioSim

With Kim Son Ly and Mathias Mollatt

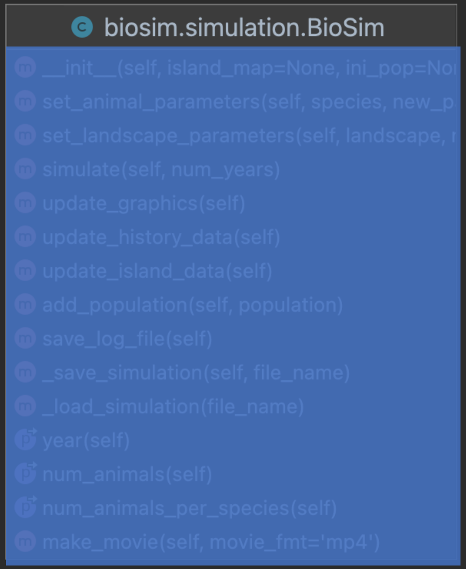
Usage

```python
# set up map
geogr = """\
           WWW
           WLW
           WWW"""

# set up initial populations
ini_herbs = [{'loc': (2, 2),
              'pop': [{'species': 'Herbivore',
                       'age': 5,
                       'weight': 20}
                      for _ in range(50)]}]
ini_carns = [{'loc': (2, 2),
              'pop': [{'species': 'Carnivore',
                       'age': 5,
                       'weight': 20}
                      for _ in range(20)]}]


sim = BioSim(geogr, ini_herbs, img_dir='results')  # Start instance of BioSim
sim.simulate(50)  # Run simulation for 50 years with visualisation
sim.add_population(ini_carns)  # Add carnivores
sim.simulate(250)  # Run simulation for 250 years with visualisation

sim.make_movie()  # Makes video of the visualisation
```

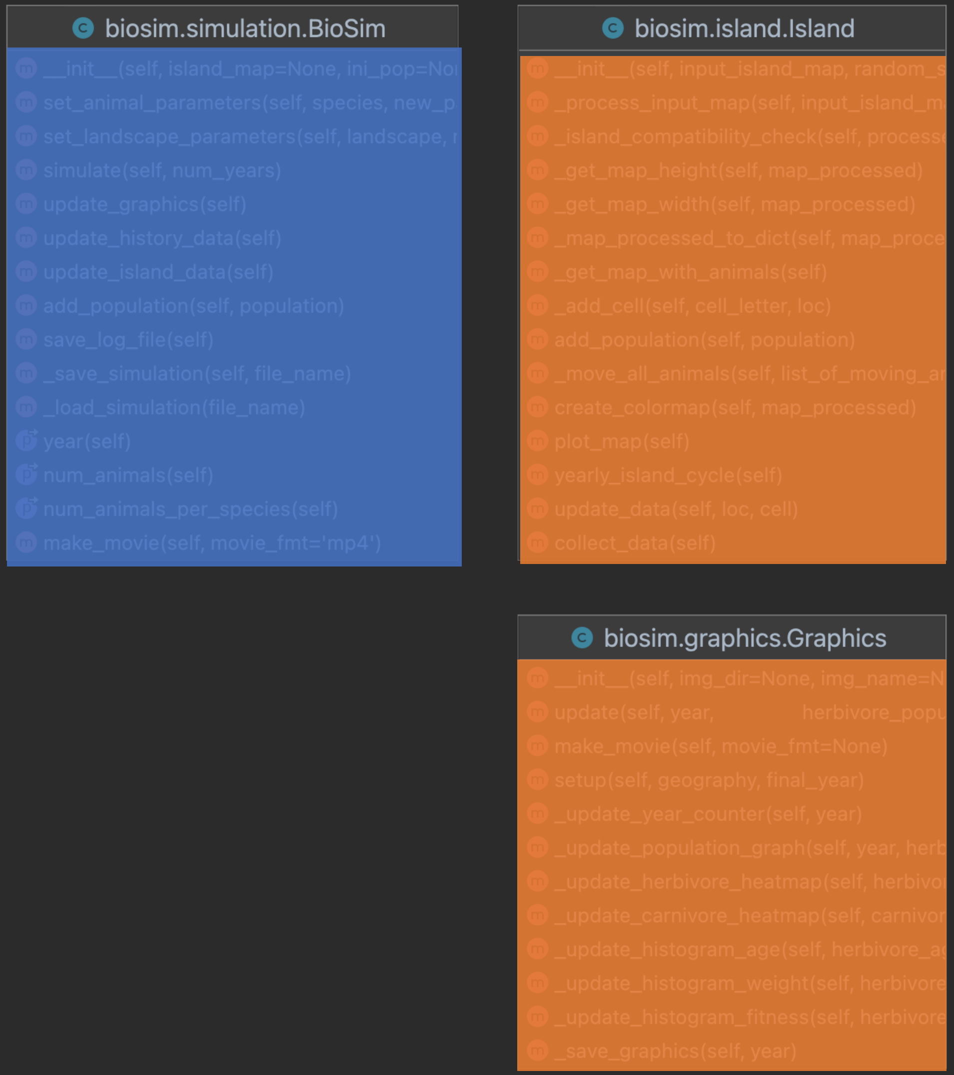
Structure: Overview



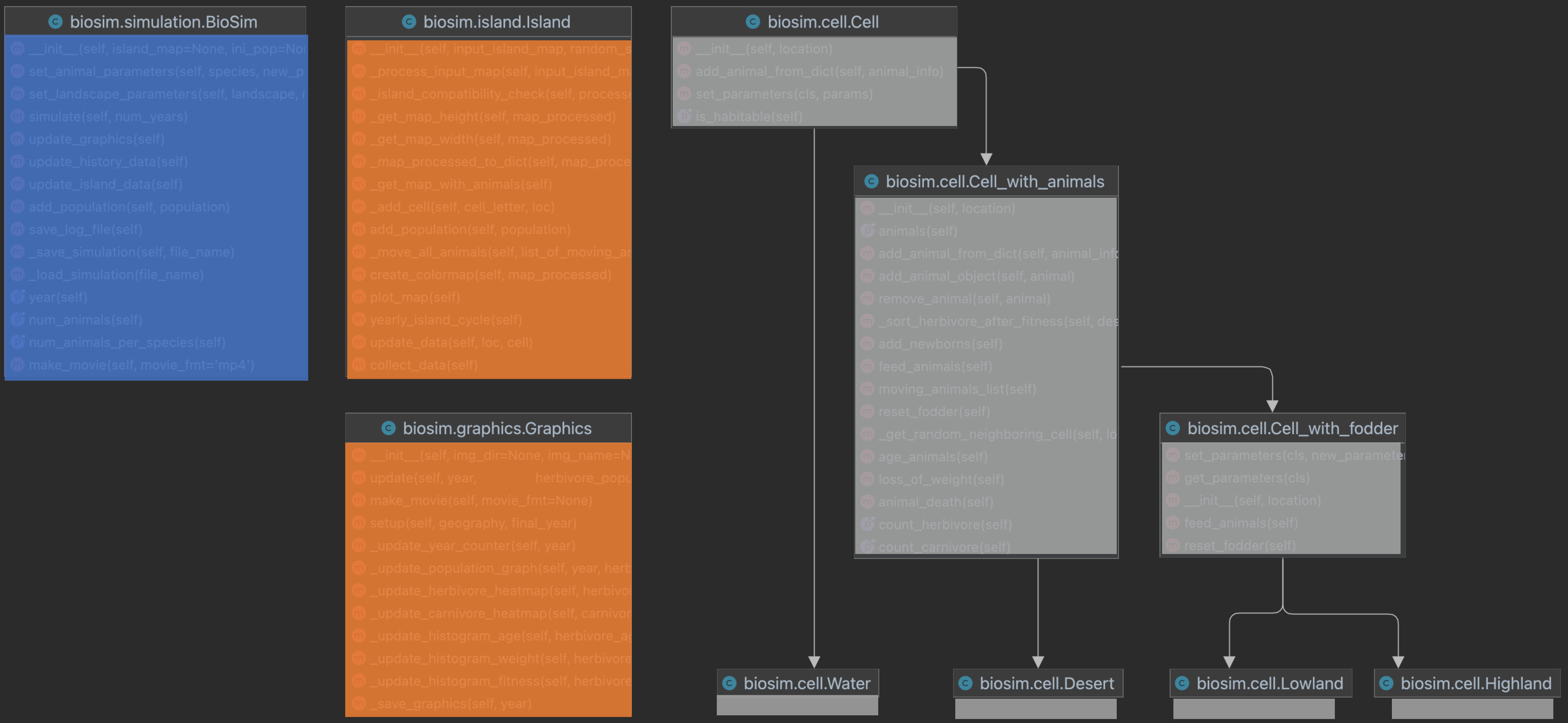
Structure: Classes



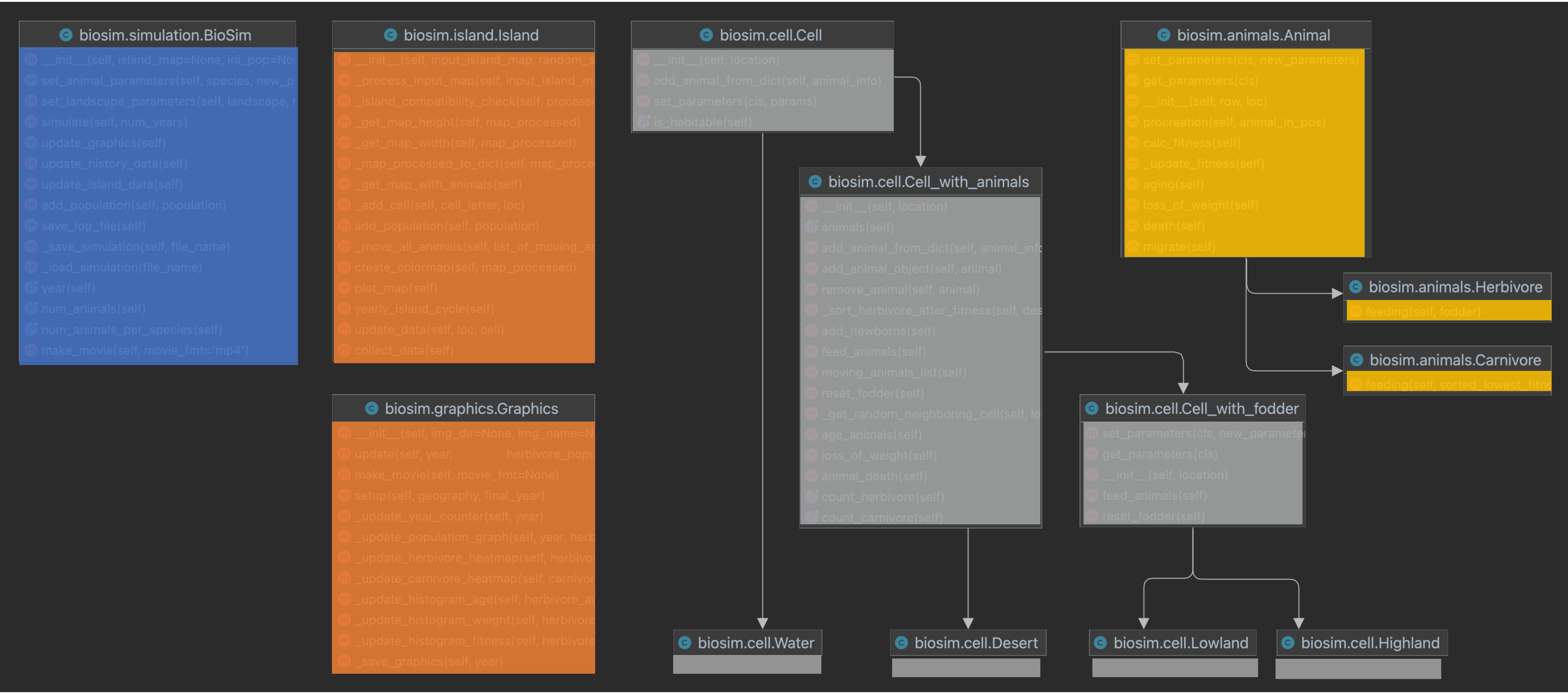
Structure: Classes



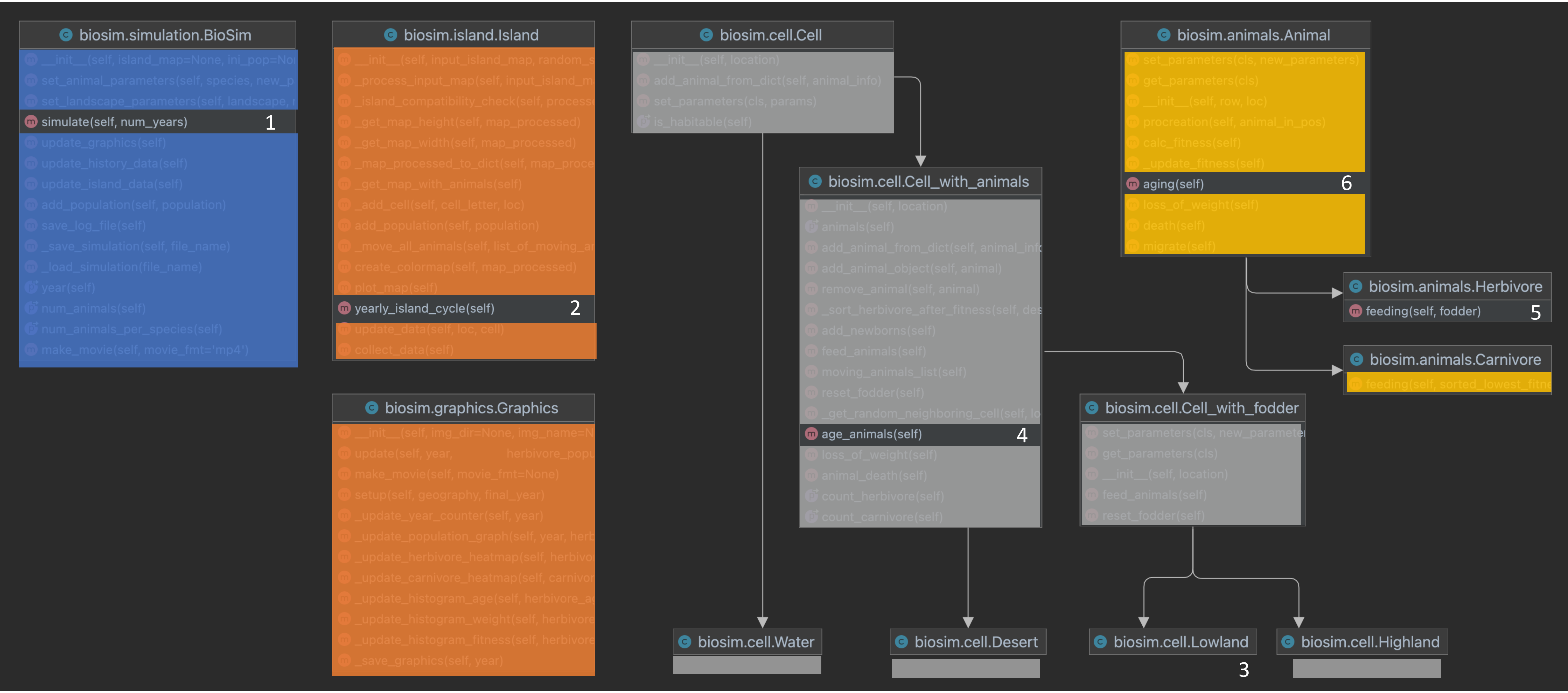
Structure: Classes



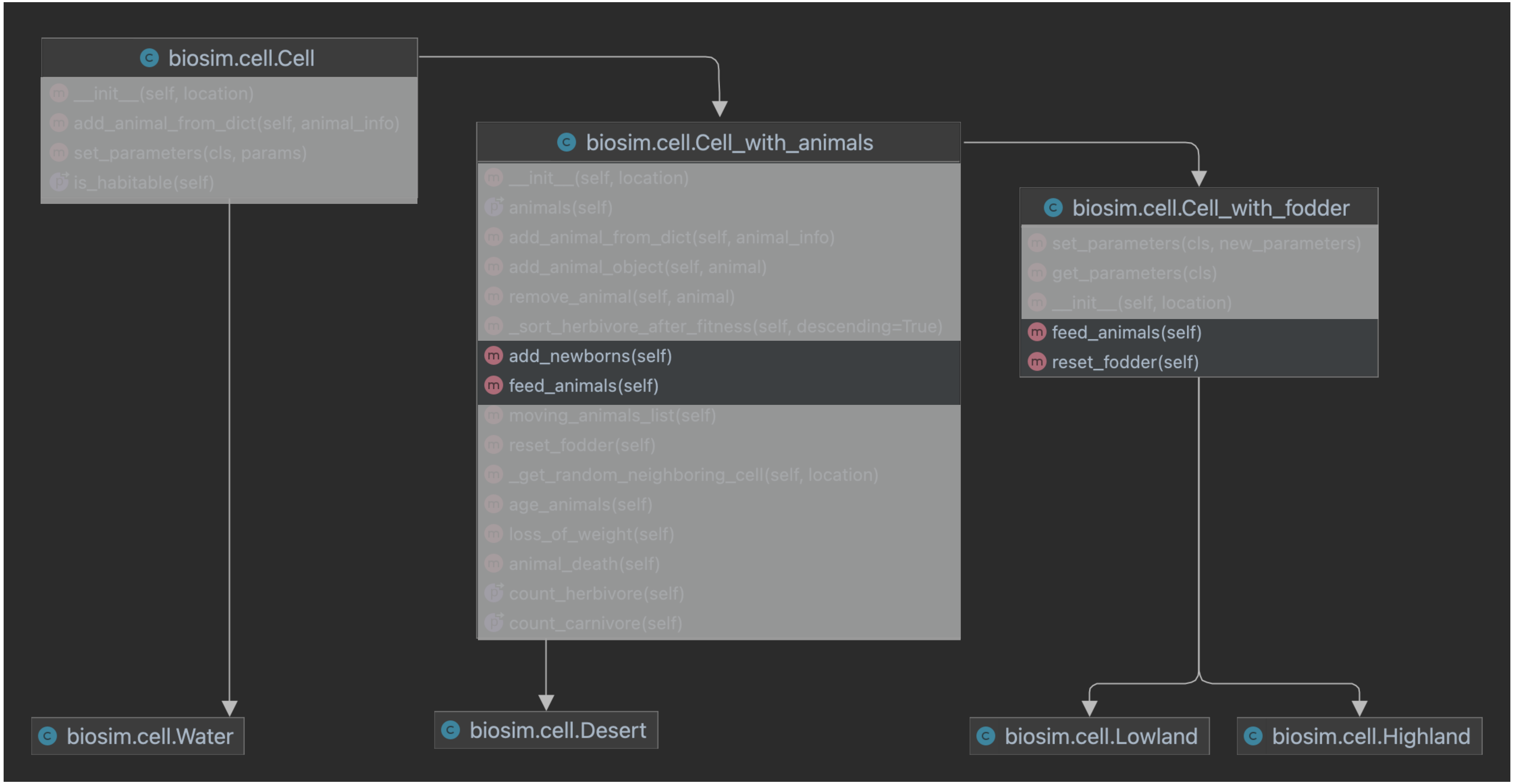
Structure: Classes



Structure: Methods and fields



Cell subclasses



Cell subclasses

Cell_with_animals

```python
def feed_animals(self):
    """..."""
    # skip if there is no herbivores in the cell
    if self.count_herbivore > 0:
        self._sort_herbivore_after_fitness(descending=False)
        random.shuffle(self.fauna["Carnivore"])
        for animal in self.fauna["Carnivore"]:...
```

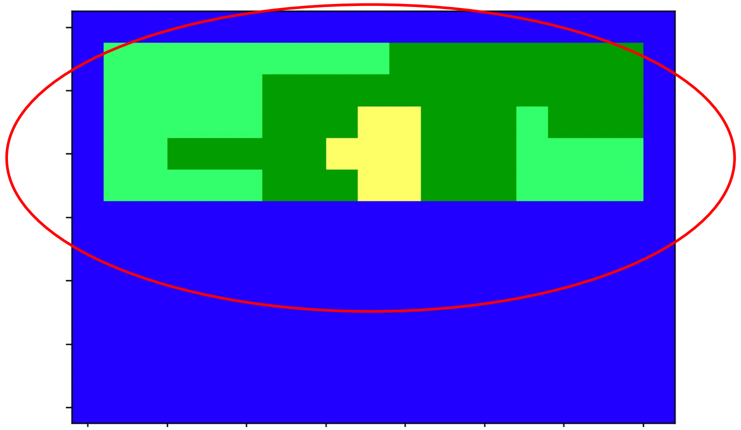
Cell_with_fodder

```python
def feed_animals(self):
    """..."""
    self._sort_herbivore_after_fitness()
    for animal in self.fauna["Herbivore"]:...

    super().feed_animals()
```

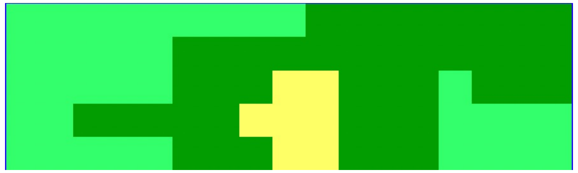
Solutions: Optimalisation

```python
def _get_map_with_animals(self):
    """
    Creates a map(dict) with only habitable cells

    Returns
    -------
    dict: map with animals
    """
    map_with_animals = {loc: cell for loc, cell in self.map.items() if cell.is_habitable}

    return map_with_animals
```

Original map



Map that is used



Solutions: Animal removal

```python
def death(self):
    r"""..."""
    probability_of_death = self.params["omega"] * (1 - self.fitness)
    if self.weight <= 0:
        self.alive = False
    elif random.random() < probability_of_death:
        self.alive = False
```

```python
def animal_death(self):
    """..."""
    for species, animal_list in self.fauna.items():
        for animal in animal_list:
            animal.death()
        # Remove dead animals
        self.fauna[species] = [animal for animal in animal_list if animal.alive]
```

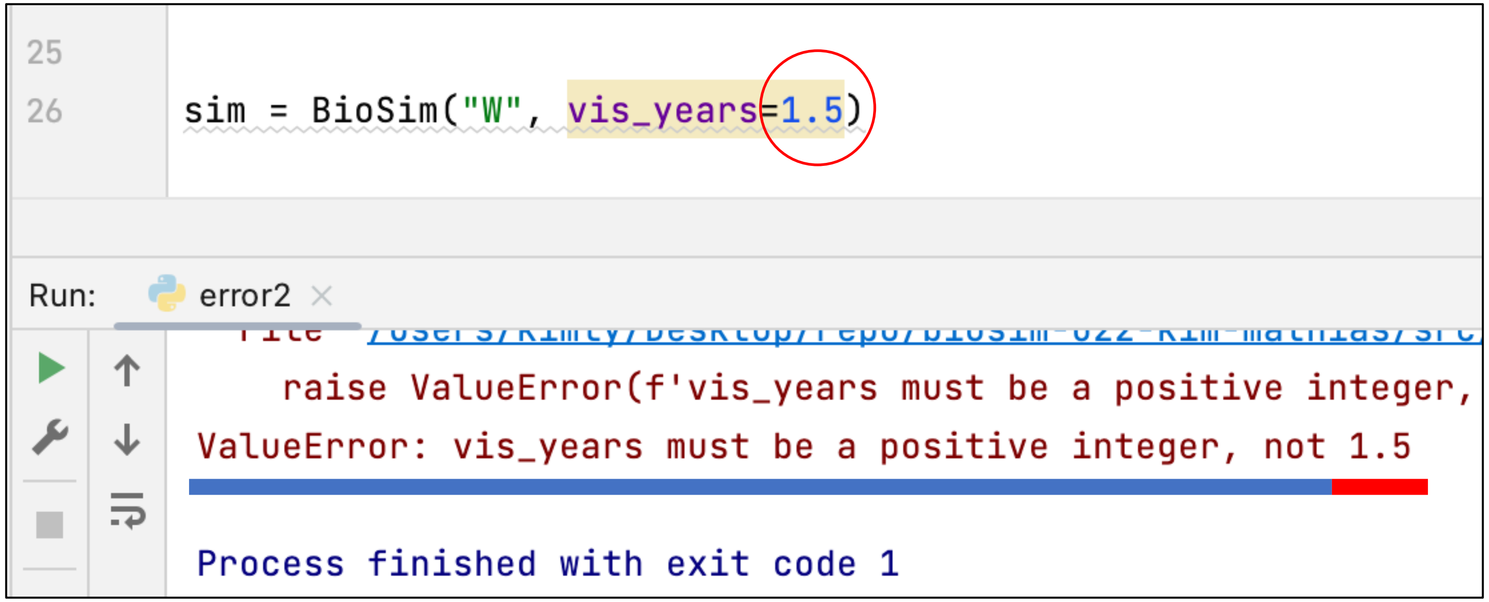
Error handling: Invalid border

```
geography = """\
            WDW
            WLW
            WWW"""

sim = BioSim(geography)
```

error1

```
    raise ValueError("Geography must be surrounded by water.")
ValueError: Geography must be surrounded by water.
```

Error handling: Invalid param



Error handling: Invalid animal placement

```python
ini_herbs = [{'loc': (1, 1),
              'pop': [{'species': 'Herbivore',
                       'age': 5.5,
                       'weight': 20}]}]

sim = BioSim("W", ini_pop=ini_herbs, vis_years=0)
```

error3

```
    raise ValueError(f"Cannot add animal to {type(self)} cell at loc: {self.locati
ValueError: Cannot add animal to <class 'biosim.cell.Water'> cell at loc: (1, 1)
```

Unit tests: Probability of procreation

```python
@pytest.mark.parametrize("animal", [std_herb(), std_carn()])
def test_procreation_prob(animal):
    """
    The outcome of procreation follows a binomial distribution. This test if the p-value of the binomial test is
    larger than 0.01, which is the significance level.
    """
    num_trials = 10000
    animal_in_cell = 2
    probability_of_procreation = min(1, animal.params["gamma"] * animal.fitness * animal_in_cell)

    num_babies = 0
    for i in range(num_trials):
        baby = animal.procreation(animal_in_cell)
        animal.weight = 60
        if baby is not None:
            num_babies += 1

    assert stats.binom_test(num_babies, num_trials, probability_of_procreation) > .01
```

Tests: Migration

```python
@pytest.mark.parametrize("species", ["Herbivore", "Carnivore"])
def test_migration_integration(species):
    """Test migration integration test.
    This forces the animals to move each year.
    The test is checking if the animals are following the chekkerboard pattern."""
    geography = """..."""

    sim = BioSim(island_map=geography, seed=23456,
                 cmax_animals={"Herbivore": 1, 'Carnivore': 1},
                 ymax_animals=100, vis_years=0)
    sim.set_animal_parameters(species, {'mu': 100000, 'eta': 0})  # Ensures that th

    ini_pop = [{...}]

    sim.add_population(population=ini_pop)
    map = sim.island.habital_map
```

```python
for year in range(1, 8):
    sim.simulate(num_years=1)
    for loc, cell in map.items():
        x = loc[0]
        y = loc[1]

        # Check if the animals are following the checkerboard pattern
        if year % 2 == 0:
            if x % 2 == 0:
                if y % 2 != 0:
                    assert cell.animals == []
            elif x % 2 != 0:
                if y % 2 == 0:
                    assert cell.animals == []
        elif year % 2 != 0:
            if x % 2 == 0:
                if y % 2 == 0:
                    assert cell.animals == []
            elif x % 2 != 0:
                if y % 2 != 0:
                    assert cell.animals == []
```

Tests: yearly cycle

```python
def test_yearly_cycle(mocker):
    """
    Test yearly cycle by choosing some methods in animal class and counting them manually.
    Then comparing them to how many times the program calls the methods.
    """
    # Keep count of methods called
    mocker.spy(Herbivore, 'calc_fitness')
    mocker.spy(Herbivore, 'procreation')
    mocker.spy(Herbivore, 'feeding')
    mocker.spy(Herbivore, 'migrate')
    mocker.spy(Herbivore, 'aging')
    mocker.spy(Herbivore, 'loss_of_weight')
    mocker.spy(Herbivore, 'death')

    mocker.spy(Carnivore, 'calc_fitness')
    mocker.spy(Carnivore, 'procreation')
    mocker.spy(Carnivore, 'feeding')
    mocker.spy(Carnivore, 'migrate')
    mocker.spy(Carnivore, 'aging')
    mocker.spy(Carnivore, 'loss_of_weight')
    mocker.spy(Carnivore, 'death')

    geogr = """..."""
    sim = BioSim(island_map=geogr, vis_years=0)

    # Adding 1 herbivore
    ini_herbs = [{...}]
    sim.add_population(ini_herbs)
    sim.simulate(1)

    # adding 1 carnivore
    ini_carns = [{...}]
    sim.add_population(ini_carns)

    sim.simulate(1)
```

```python
    result = {"h_calc_fit": Herbivore.calc_fitness.call_count,
              "h_procreation": Herbivore.procreation.call_count,
              "h_feeding": Herbivore.feeding.call_count,
              "h_migrate": Herbivore.migrate.call_count,
              "h_aging": Herbivore.aging.call_count,
              "h_loss_of_weight": Herbivore.loss_of_weight.call_count,
              "h_death": Herbivore.death.call_count,
              "c_calc_fit": Carnivore.calc_fitness.call_count,
              "c_procreation": Carnivore.procreation.call_count,
              "c_feeding": Carnivore.feeding.call_count,
              "c_migrate": Carnivore.migrate.call_count,
              "c_aging": Carnivore.aging.call_count,
              "c_loss_of_weight": Carnivore.loss_of_weight.call_count,
              "c_death": Carnivore.death.call_count}

    # Manually counted method calls:
    expect = {"h_calc_fit": 5,
              "h_procreation": 2,
              "h_feeding": 2,
              "h_migrate": 2,
              "h_aging": 2,
              "h_loss_of_weight": 2,
              "h_death": 2,
              "c_calc_fit": 2,
              "c_procreation": 1,
              "c_feeding": 0,
              "c_migrate": 1,
              "c_aging": 1,
              "c_loss_of_weight": 1,
              "c_death": 1}

    assert result == expect
```