Soil and Crop Redesign

How to make more independent and usable code

Clay J. Morrow

Introduction

- Problems overview
- Redesign details: module structure, organization
- General coding goals
- Example code changes
- Conclusions and discussion

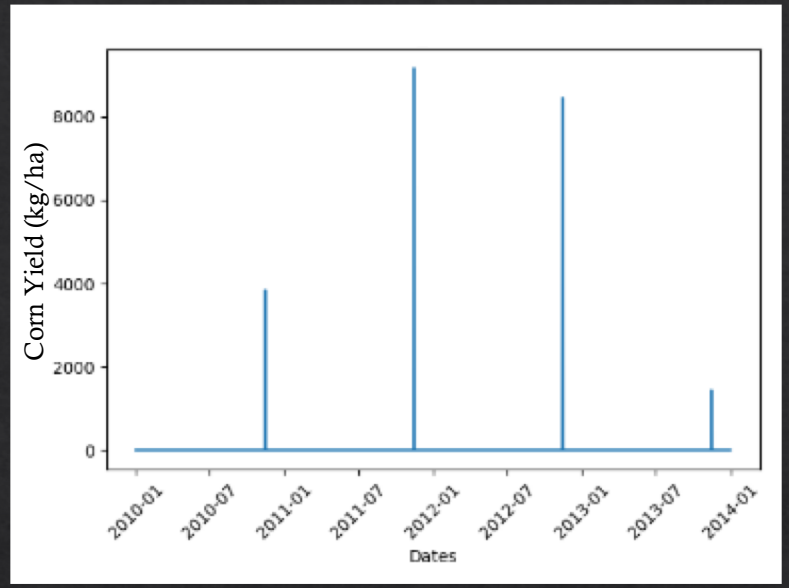
- Disclaimer: I am **not** a trained software engineer*

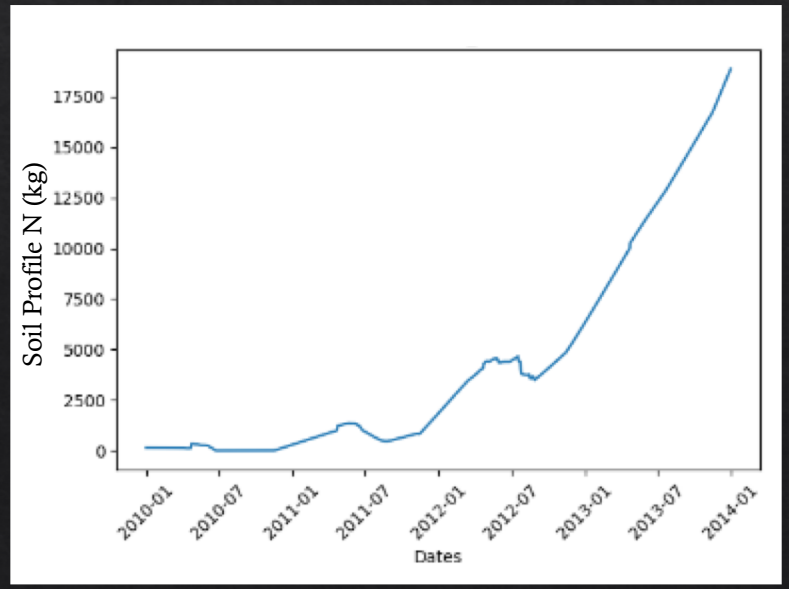
Problems

And how to fix them

Problems: major bugs

- Inconsistent Crop yields/biomass production
- Planting events ignore input specifications
- Nitrogen accumulates in soil without amendments
- Unrealistic Growth cycles
- More?





Problems: structure

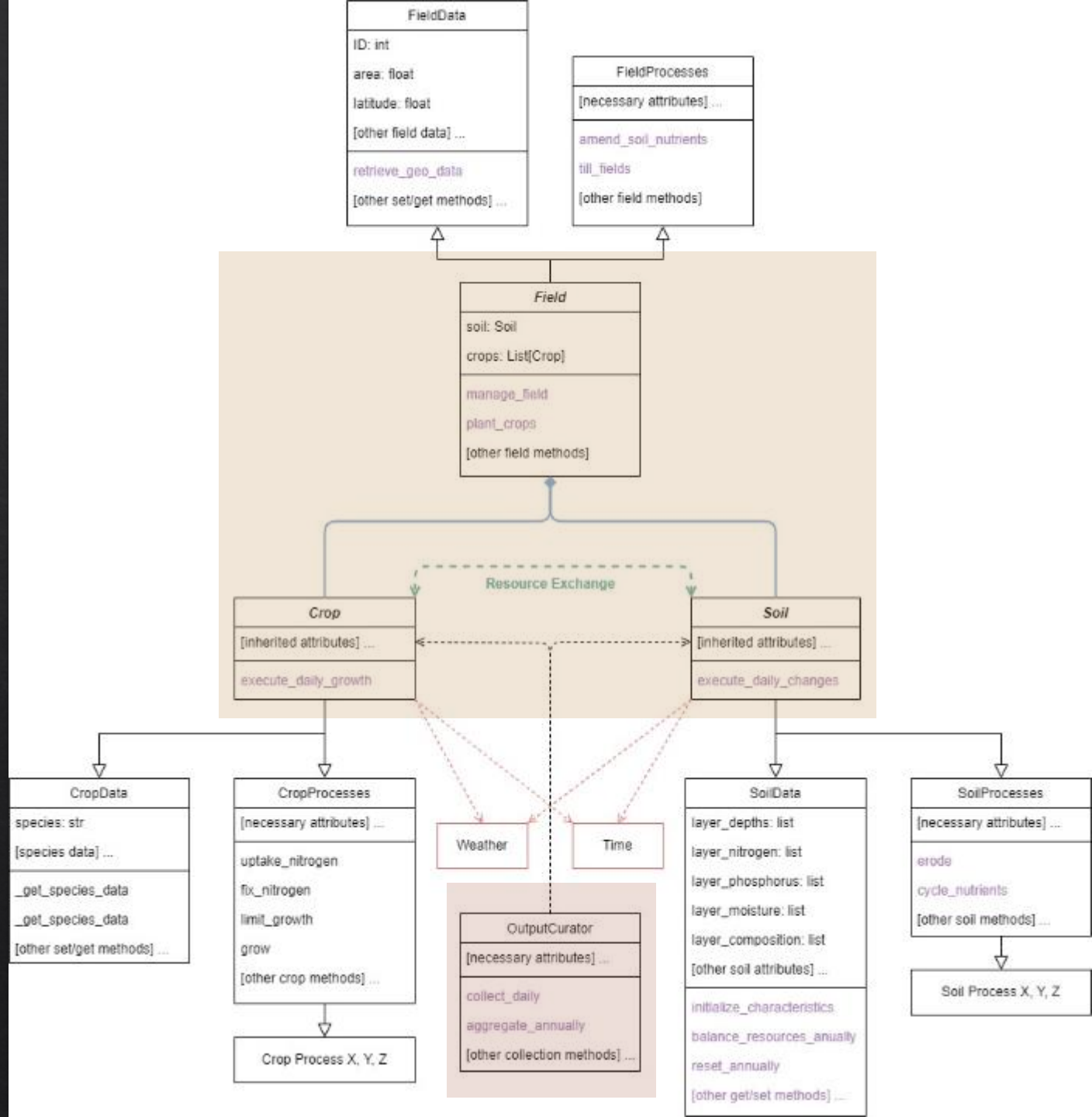
- Tests unimplemented
- Lack of isolation (difficult to test)
- Cryptic naming conventions
- Poor documentation
- Inconsistent across module
- “messy” code

Solution: redesign

- Redesign and restructuring needed
- Component/process validation
- Isolated and independent units
- Consistent structure
- Readable code
- Practice code-test-document
- Up-front investment

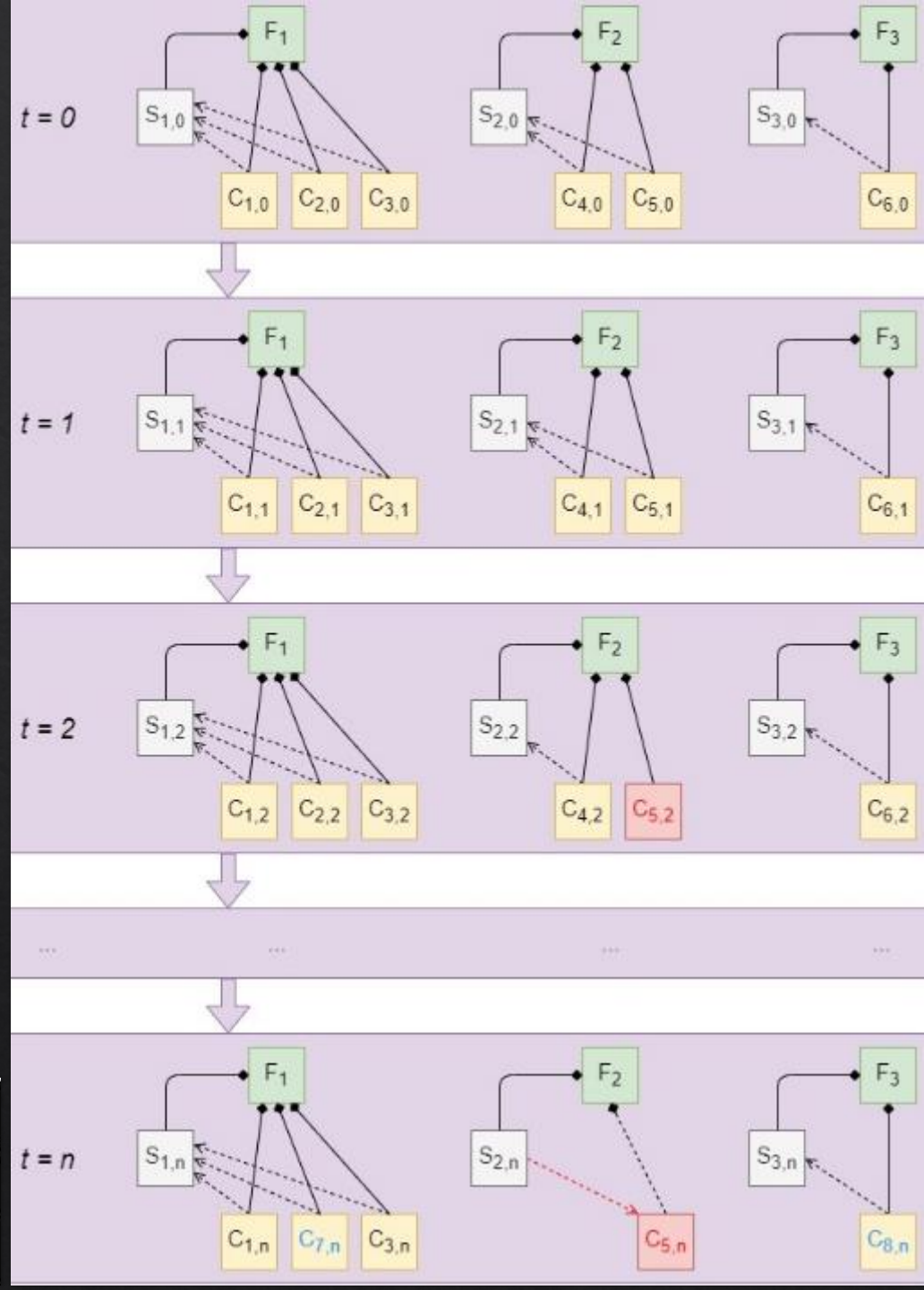
Basic Structure

- 3 main classes
  - Soil
  - Crop
  - Field
- Each inherit from subclasses
  - Data (1)
  - Processes (n)
- Secondary OutputCurator
  - collects, organizes, aggregates data
  - gives to output handler



Temporal structure (conceptual)

- Multiple fields simulated
- Variable number of crops per field
- Crops exchange resources with soil in their field
- Crops and soil change through time (daily)
- Crops can be harvested, killed, and replaced

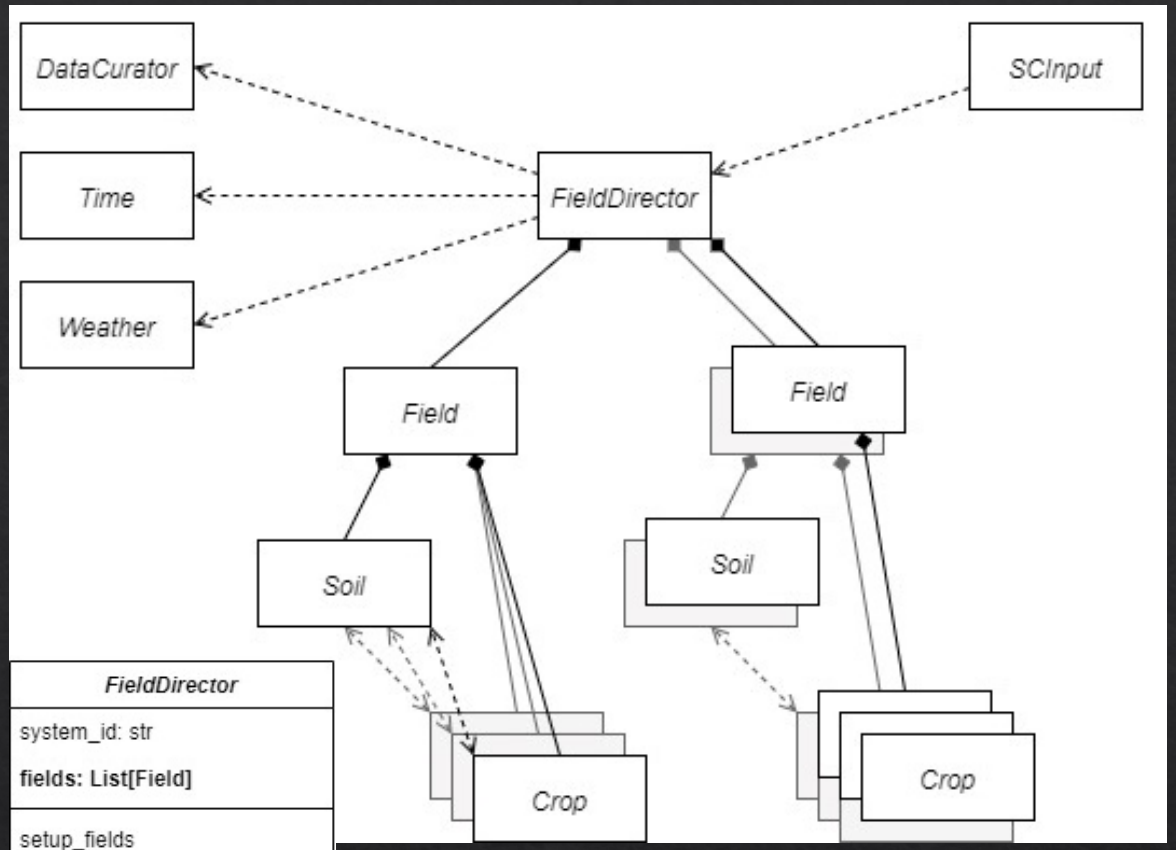


F_i	field *i*
$S_{i,t}$	soil profile *i* at time *t*
$C_{i,t}$	crop *i* at time *t*

FieldDirector Class

- Contains a collection of Field instances
- Drives all processes
- Iterates through fields and crops within a field
- Handles the dependence to Weather, and Time
- Pushes output to DataCurator

- Called by SimulationEngine._daily_simulation()



FieldDirector

system_id: str

fields: List[Field]

setup_fields

manage_fields

cycle_soil

plant_crops

grow_crops

harvest_crops

kill_crops

collect_data

read_input

cleanup_and_reset

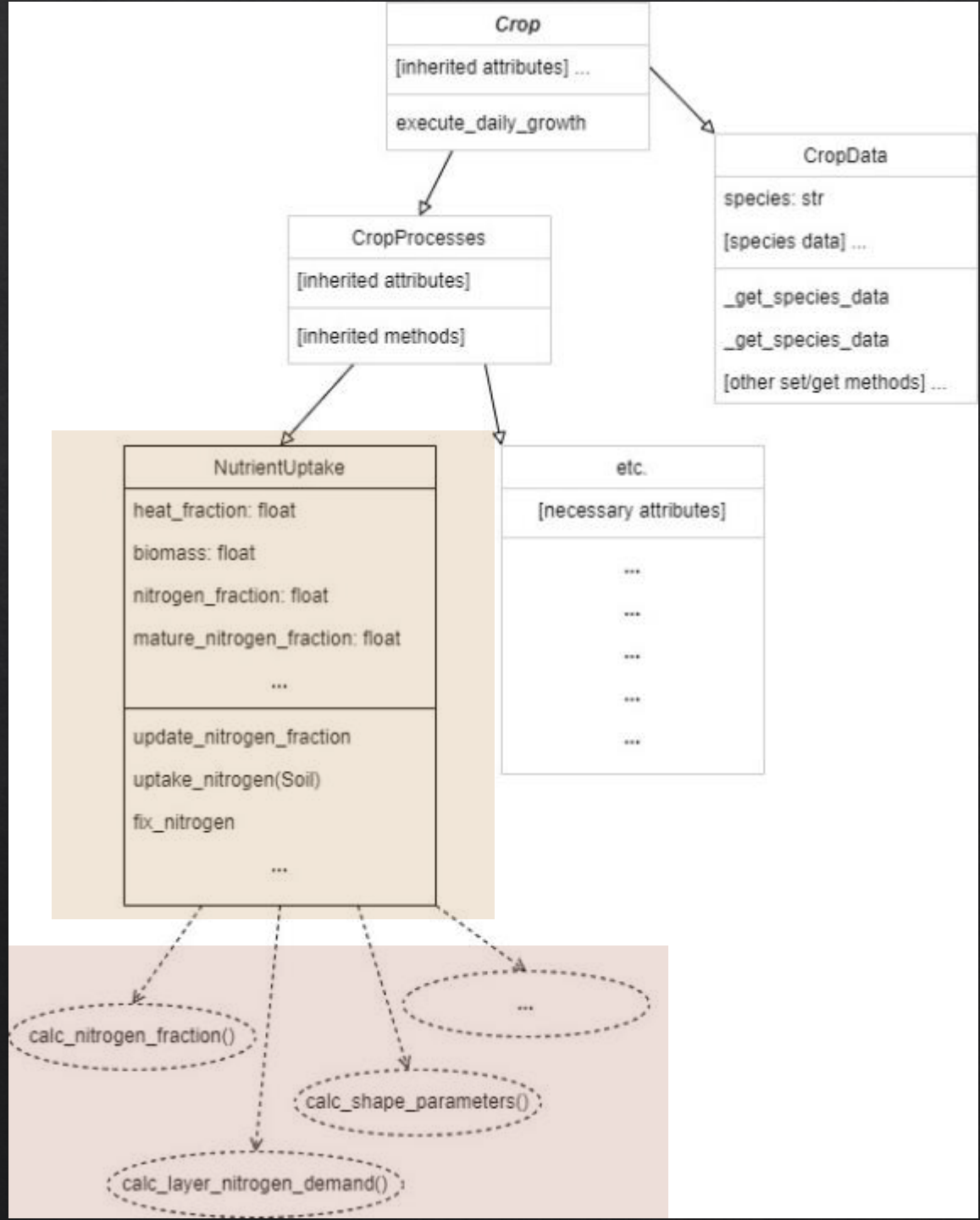
check_weather

check_time

run_daily_field_routines

Process Classes

- Main objects depend upon many processes
- Processes require multiple methods
- Separate file for each process
  - process class object
  - helper functions
- Process classes contain
  - necessary attributes for methods
  - member functions that update attributes
- External helper functions
  - input values, output values
  - used by member functions



Aspirations:

How the code should be

Methods and Functions

- Do one thing!
- Void vs value-returning
- Value parameters whenever possible
- Within a class or not?

Naming conventions

- Intuitive names
- Descriptive
- Plain English (when possible)
- Understandable >> short

```
def calc_gamma_reg(soil, crop, weather, time) -> None:
```

VS

```
def calc_growth_factor(water_stress: float, temperature_stress: float, nitrogen_stress: float,
                       phosphorus_stress: float) -> float:...
```

&

```
def determine_growth_factor(self) -> None:...
```

Testing

- Test everything!
- Verify calculations and equations
- Verify side-effects of void functions
- Extensive test conditions:
  - edge cases
  - logic coverage
  - with fixed parameter set, change variable *x*
  - reasonable/realistic values

```python
@pytest.mark.parametrize("temp,min,opt", [
    (1, 1, 1),  # all 1 (A)
    (1, 0, 0),  # air 1 (D)
    (0, 1, 0),  # min 1 (A)
    (0, 0, 1),  # opt 1 (A)
    (0, 0, 0),  # all 0 (A)
    (0.5, 0, 1),  # min < air < opt (B)
    (1, 0, 1),  # min < air = opt (B)
    (1.5, 0, 1),  # opt < air < 2*opt - min (C)
    (2, 0, 1),  # opt < air = 2*opt - min (C)
    (3, 0, 1),  # opt < air > 2*opt - min (D)
    (5.6, 12.2, 25.5),  # arbitrary (A)
    (15.8, 12.2, 25.5),  # arbitrary (B)
    (36.7, 12.2, 25.5),  # arbitrary (C)
    (39.9, 12.2, 25.5),  # arbitrary (D)
])
def test_calc_temperature_stress(temp, min, opt):...
```

Model Documentation

- “Pseudocode”
- Overall functionality
- Equations and explanations of processes
- Variable translation (code)
- Algorithms
- Separate from docstrings/Sphinx

Code Example:

Before and After

Example: growth constraints

- Growth factor:

$$\gamma_{reg} = 1 - \max\{s_n, \dots\}$$

- Nitrogen stress:

$$s_n = 1 - \frac{\phi_n}{\phi_n + e^{3.5 - 0.03\phi_n}}$$

- Stress scaling factor:

$$\phi_n = 200\left(\frac{N}{N_{opt}} - 0.05\right)$$

```python
# growth_constraints.py
def calc_gamma_reg(soil, crop, weather, time) -> None:
    w_stress = calc_w_stress(soil, crop)
    t_stress = calc_t_stress(crop, weather, time)
    n_stress = calc_n_stress(crop)
    p_stress = calc_p_stress(crop)
    crop.gamma_reg = 1.0 - max(w_stress, t_stress, n_stress, p_stress)

def calc_n_stress(crop) -> float:
    phi_n = calc_phi_N(crop)
    return 1 - phi_n / (phi_n + exp(3.5 - 0.03 * phi_n))

def calc_phi_N(crop) -> float:
    return 200 * ((crop.bio_N / crop.bio_N_opt) - 0.5)


def calc_p_stress(crop) -> float:...
def calc_phi_P(crop) -> float:...
def calc_w_stress(soil, crop) -> float:...
def calc_t_stress(crop, weather, time) -> float:...
```

Example: growth constraints

- Unintuitive names
- All functions take **classes** as arguments
- Common naming convention
- calc_gamma_reg returns nothing, updates crop
- calc_n_stress and calc_phi_N return values

```python
# growth_constraints.py
def calc_gamma_reg(soil, crop, weather, time) -> None:
    w_stress = calc_w_stress(soil, crop)
    t_stress = calc_t_stress(crop, weather, time)
    n_stress = calc_n_stress(crop)
    p_stress = calc_p_stress(crop)
    crop.gamma_reg = 1.0 - max(w_stress, t_stress, n_stress, p_stress)

def calc_n_stress(crop) -> float:
    phi_n = calc_phi_N(crop)
    return 1 - phi_n / (phi_n + exp(3.5 - 0.03 * phi_n))

def calc_phi_N(crop) -> float:
    return 200 * ((crop.bio_N / crop.bio_N_opt) - 0.5)


def calc_p_stress(crop) -> float:...
def calc_phi_P(crop) -> float:...
def calc_w_stress(soil, crop) -> float:...
def calc_t_stress(crop, weather, time) -> float:...
```

Example: Testing

- calc_n_stress depends upon calc_phi_N
  - Stress factor (ϕ_n) calculated internally
  - Nitrogen values needed (N, N_{opt})

```python
# growth_constraints.py
def calc_n_stress(crop) -> float:
    phi_n = calc_phi_N(crop)
    return 1 - phi_n / (phi_n + exp(3.5 - 0.03 * phi_n))

def calc_phi_N(crop) -> float:
    return 200 * ((crop.bio_N / crop.bio_N_opt) - 0.5)
```

Nitrogen stress:

$$s_n = 1 - \frac{\phi_n}{\phi_n + e^{3.5 - 0.03\phi_n}}$$

Stress factor:

$$\phi_n = 200\left(\frac{N}{N_{opt}} - 0.05\right)$$

- Initialize a crop with needed attributes, run function
- Calculate expectation (2 steps)
- Test for equality

```python
# test_growth_constraints.py
def test_calc_n_stress(nitro=0.5, opt_nitro=1):
    crop = BaseCrop()
    crop.bio_N = nitro
    crop.bio_N_opt = opt_nitro
    observe = calc_n_stress(crop)

    strs_fct = 200 * ((nitro / opt_nitro) - 0.5)
    expect = 1 - strs_fct / (strs_fct + exp(3.5 - 0.03 * strs_fct))

    assert observe == expect
```

Example: Testing

- calc_gamma_reg
  - Depends upon `calc_x_stress()` functions
  - Requires:
    - water uptake (`water_uptake`),
    - transpiration (`trans_max`),
    - nitrogen (`nitro`),
    - optimal nitrogen (`opt_nitro`),
    - phosphorus (`phos`),
    - optimal phosphorus (`opt_phos`)
  - It updates `gamma_reg` attribute

- Initialize classes, run function
- Calculate expectation (5 steps)
- Test for equality

```python
def test_calc_gamma_reg(water_uptake=85, trans_max=0.8, nitro=0.5,
                        opt_nitro=1.0, phos=0.3, opt_phos=0.5):
    crop = BaseCrop()
    crop.water_act_up = water_uptake  # calc_w_stress
    crop.bio_N = nitro
    crop.bio_N_opt = opt_nitro  # calc_n_stress

    crop.bio_P = phos  # calc_p_stress
    crop.bio_P_opt = opt_phos  # calc_p_stress

    soil = Soil()
    soil.trans_max = trans_max  # calc_w_stress

    time = Time()
    time.day = 1  # calc_t_stress
    time.year = 1  # calc_t_stress
    weather = Weather
    weather.T_avgs = [[20]]  # calc_t_stress

    calc_gamma_reg(soil, crop, weather, time)

    w_stress = calc_w_stress(soil, crop)
    t_stress = calc_t_stress(crop, weather, time)
    n_stress = calc_n_stress()
    p_stress = calc_p_stress()

    expect = 1 - max(w_stress, t_stress, n_stress, p_stress)

    assert crop.gamma_reg == expect
```

Example: a better way

```python
def calc_gamma_reg(soil, crop, weather, time) -> None:
    w_stress = calc_w_stress(soil, crop)
    t_stress = calc_t_stress(crop, weather, time)
    n_stress = calc_n_stress(crop)
    p_stress = calc_p_stress(crop)
    crop.gamma_reg = 1.0 - max(w_stress, t_stress, n_stress, p_stress)

def calc_n_stress(crop) -> float:
    phi_n = calc_phi_N(crop)
    return 1 - phi_n / (phi_n + exp(3.5 - 0.03 * phi_n))

def calc_phi_N(crop) -> float:
    return 200 * ((crop.bio_N / crop.bio_N_opt) - 0.5)


def calc_p_stress(crop) -> float:...
def calc_phi_P(crop) -> float:...
def calc_w_stress(soil, crop) -> float:...
def calc_t_stress(crop, weather, time) -> float:...
```

```python
# growth_constraints.py
class GrowthConstraints:
    def __init__(self):...

    def determine_growth_factor(self) -> None:
        self.growth_factor = calc_growth_factor(self.nitrogen_stress, self.phosphorus_stress,
                                                self.water_stress, self.temperature_stress)


def calc_growth_factor(nitrogen_stress, phosphorus_stress, water_stress, temperature_stress) -> float:
    return 1.0 - max(nitrogen_stress, phosphorus_stress, water_stress, temperature_stress)

def calc_nutrient_stress(scaling_factor) -> float:
    return 1.0 - scaling_factor / (scaling_factor + exp(3.5 - 0.03 * scaling_factor))

def calc_stress_scaling_factor(stored, optimal) -> float:
    return 200.0 * (stored / optimal) - 0.5
```

Example: better tests

```python
def test_calc_gamma_reg(water_uptake=85, trans_max=0.8, nitro=0.5,
                        opt_nitro=1.0, phos=0.3, opt_phos=0.5):
    crop = BaseCrop()
    crop.water_act_up = water_uptake  # calc_w_stress
    crop.bio_N = nitro
    crop.bio_N_opt = opt_nitro  # calc_n_stress

    crop.bio_P = phos  # calc_p_stress
    crop.bio_P_opt = opt_phos  # calc_p_stress

    soil = Soil()
    soil.trans_max = trans_max  # calc_w_stress

    time = Time()
    time.day = 1  # calc_t_stress
    time.year = 1  # calc_t_stress
    weather = Weather
    weather.T_avgs = [[20]]  # calc_t_stress

    calc_gamma_reg(soil, crop, weather, time)

    w_stress = calc_w_stress(soil, crop)
    t_stress = calc_t_stress(crop, weather, time)
    n_stress = calc_n_stress()
    p_stress = calc_p_stress()

    expect = 1 - max(w_stress, t_stress, n_stress, p_stress)

    assert crop.gamma_reg == expect
```

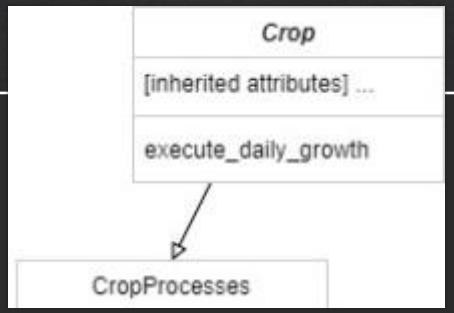
```python
# test_growth_constraints.py
def test_calc_n_stress(nitro=0.5, opt_nitro=1):
    crop = BaseCrop()
    crop.bio_N = nitro
    crop.bio_N_opt = opt_nitro
    observe = calc_n_stress(crop)

    strs_fct = 200 * ((nitro / opt_nitro) - 0.5)
    expect = 1 - strs_fct / (strs_fct + exp(3.5 - 0.03 * strs_fct))

    assert observe == expect
```

```python
# test_growth_constraints.py
def test_calc_growth_factor(nitrogen=0.41, phosphorus=0.2, water=0.3, temp=0.4):
    expect = 1 - max(nitrogen, phosphorus, water, temp)
    assert calc_growth_factor(nitrogen, phosphorus, water, temp) == expect


def test_calc_nutrient_stress(sclr=0.35):
    expect = 1 - sclr / (sclr + exp(3.5 - 0.3 * sclr))
    assert calc_nutrient_stress(sclr) == expect


def test_calc_stress_scaling_factor(str=0.5, opt=1.0):
    expect = 200 * (str/opt - 0.5)
    assert calc_stress_scaling_factor(str, opt) == expect
```

Example: process class

- Crop process
- Initialized with all necessary attributes to run methods
- Functions update attributes

- constrain_growth()
  - Main function, executes routines
  - uses attributes
  - Accepts parameter values
  - Initializing only this class for tests




```python
# growth_constraints.py
class GrowthConstraints:
    def __init__(self):
        self.water_uptake = 18
        self.nitrogen = 35
        self.optimal_nitrogen = 100
        self.phosphorus = 20
        self.optimal_phosphorus = 80
        self.minimum_temp = 15
        self.optimal_temp = 22

    def constrain_growth(self, max_transpiration: float, temperature: float) -> None:
        self.assess_water_stress(max_transpiration)
        self.assess_temp_stress(temperature)
        self.assess_nitrogen_stress()
        self.assess_phosphorus_stress()
        self.determine_growth_factor()

    def assess_water_stress(self, max_transpiration):...
    def assess_temp_stress(self, temperature):...
    def assess_nitrogen_stress(self):...
    def assess_phosphorus_stress(self):...
    def determine_growth_factor(self):...

def calc_growth_factor(nitrogen_stress, phosphorus_stress, water_stress, temperature_stress) -> float:...
def calc_nutrient_stress(scaling_factor) -> float:...
def calc_stress_scaling_factor(stored, optimal) -> float:...
def calc_water_stress(uptake, max_transpiration) -> float:...
def calc_temperature_stress(air_temperature, min_temperature, optimal_temperature) -> float:...
```

Example Summary

- Old
  - unintuitive names
  - do multiple things
  - functions depends on classes
  - functionality is repeated
  - class in separate file
  - wrapper function called externally
- New
  - better names
  - do one thing
  - accept value arguments
  - no more repeats
  - wrapper is member
  - flexible, testable, modular

Conclusions

- Good names make code easier to understand
- Functions should do one thing (calculate vs. update)
- Functions that don't update attributes (void) should return values
- Storing process functions in classes helps organize and isolate
- Isolated code is easier to test and use
- Good documentation makes everything easier

- Take ownership over the code base – if you see a mess, clean it!

Discussion

- Questions?
- Comments?
- Suggestions?
- Criticism?