

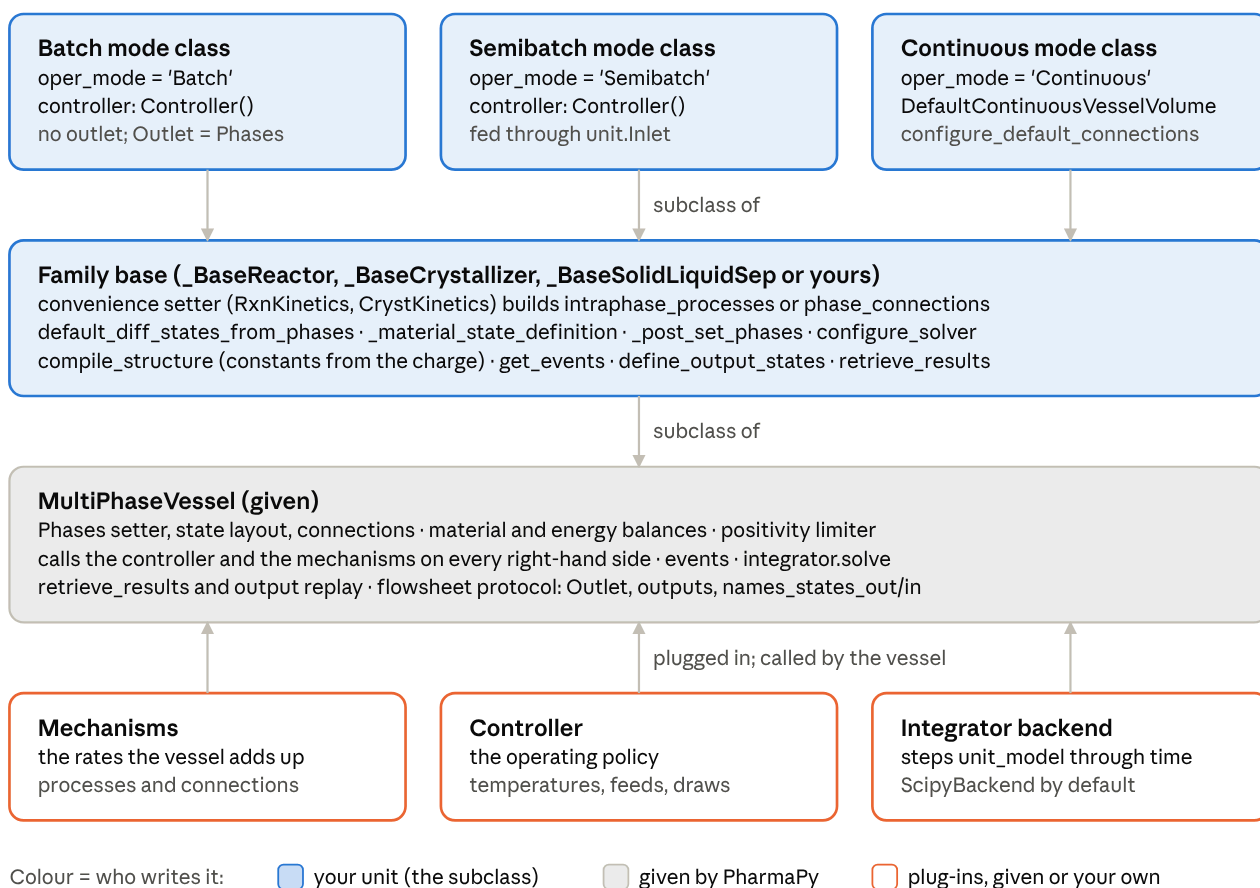
Writing your own PharmaPy unit operation

Oct 7, 2026 · @Nagy Group

At a glance

Reactors, crystallizers and the filter are all built the same way. A thin **mode class** sets the operating mode, the default controller and the outlet. It sits on a **family base** that wires the physics. That base sits on `MultiPhaseVessel`, which already does the balances, streams, events, solving, results and flowsheet hand-over.

A new unit adds wiring; `MultiPhaseVessel` already does the rest



the layers of a unit operation · what you write, what is given, what plugs in

A new unit operation is therefore mostly a family base: which phases are integrated, which mechanisms run where, what the charge fixes, and what stops the run. Usually you add the three short mode classes as well.

Life cycle of a unit

Every unit operation goes through the same seven stages. `MultiPhaseVessel` (`MultiPhaseVessel.py:28`) does the work at each stage, and at each one it calls a method a subclass can override. Mechanisms, controllers and integrators plug in at the stages marked below; their own API is in the companion doc [Mechanism, integrator and controller API](#).

#	The user writes	What <code>MultiPhaseVessel</code> does	Hooks a subclass overrides
1. Build	<code>unit = MyUnit(**kwargs)</code>	<code>__init__</code> (:42) sets a default <code>Controller()</code> and <code>ScipyBackend()</code> , empty connections and state collections, and <code>elapsed_time = 0</code> . It accepts <code>integrator</code> , <code>controller</code> , <code>isothermal</code> , <code>adiabatic</code> , <code>h_conv</code> , <code>diam</code> , <code>area_base</code> , <code>ht_mode</code> , <code>basis='mass_j'</code> , <code>temp_ref</code> , <code>state_events</code> , <code>emit_events</code> and <code>positivity_horizon</code>	<code>__init__</code> : set your own attributes, put defaults in with <code>kwargs.setdefault(...)</code> , then call <code>super().__init__(**kwargs)</code>
2. Charge	<code>unit.Phases = phase or [liquid, solid]</code>	The setter (:132) turns a stream into a phase (refused for 'Batch') and, on a flowsheet hand-over, merges by phase family. Then <code>_post_set_phases()</code> (:225) runs <code>define_material_states()</code> (one <code>_material_state_definition()</code> per phase), <code>default_diff_states_from_phases()</code> , <code>configure_default_connections()</code> , and <code>nomenclature()</code> , which runs <code>define_solver_states()</code> and <code>define_output_states()</code>	<code>_post_set_phases</code> (call <code>super()</code> first), <code>_material_state_definition</code> , <code>default_diff_states_from_phases</code> , <code>configure_default_connections</code> , <code>define_output_states</code>
3. Physics	<code>unit.intraphase_processes = [...]</code> , <code>unit.phase_connections = [...]</code> , or a convenience setter such as <code>RxnKinetics</code>	Stores them and calls <code>nomenclature(overwrite=True)</code> , so the mechanisms' states and outputs are registered	A property setter that builds processes or connections from something friendlier (<code>Reactors_Refactored.py:22</code> , <code>Crystallizers_Refactored.py:22</code>), or build them yourself in <code>_post_set_phases</code>
4. Feed	<code>unit.Inlet = stream</code>	One <code>StreamConnection</code> per stream, each phase mapped to the vessel phase of the same family and index (:278)	Rarely; set <code>inlet_connections</code> for unusual mappings

#	The user writes	What MultiPhaseVessel does	Hooks a subclass overrides
5. Solve	<code>unit.solve_unit(runtime=...) or time_grid=...</code>	<code>compile_structure()</code> (:1285) compiles the state layout and pushes <code>positivity_horizon</code> to the mechanisms. It gives each phase connection without a mechanism a <code>DirectTransfer</code> , and calls <code>compile_events()</code> , which calls <code>get_events()</code> . Then <code>integrator.solve(self, ...)</code> calls <code>reset()</code> , <code>create_solver_init_states()</code> and <code>configure_solver()</code> , then <code>unit_model(t, y)</code> at every step	<code>compile_structure</code> (call <code>super()</code>), then compute constants from the charge), <code>get_events</code> (extend <code>super()</code> 's list), <code>configure_solver</code> , <code>reset</code> , plus the balance hooks in the next section
6. Results	Reads <code>unit.result</code>	<code>retrieve_results(time, states)</code> (:3064) replays every saved time on a deep copy of the unit (<code>create_pseudo</code>) to evaluate each output state. It puts the final state back on the phases, stores <code>outlet_conditions</code> , moves <code>elapsed_time</code> and builds <code>unit.result</code> . Result names are <code><state>_<phase><index></code> , such as <code>mass_j_liquid0</code> , plus unit-level names such as <code>global_temp</code> and <code>vessel_vol</code>	<code>retrieve_results</code> : call <code>super()</code> , then post-process, as <code>Filter</code> does (<code>SolidLiquidSep_Refactored.py:452</code>)
7. Hand-over	<code>SimulationExec(...).SolveFlowsheet(...)</code>	Reads <code>oper_mode</code> , <code>Outlet</code> , <code>outputs</code> , <code>names_states_out</code> and <code>names_states_in</code> , then sets the next unit's <code>Phases</code> or <code>Inlet</code> (see Flowsheets below)	<code>Outlet</code> , <code>default_output</code> , and rarely <code>outputs</code>

Assign `Phases` after construction. Passing `Phases=` to the constructor fails today with `AttributeError: ... has no attribute 'phase_states'`, because the constructor assigns it before the state collections exist.

The subclass API

A unit operation is mostly wiring. Rates belong in a **mechanism**, the operating policy in a **controller**, and integration in a **backend**. The subclass decides:

- which phases hold an integrated inventory;
- which mechanisms run, and on which phases;
- which outlets exist;
- which constants follow from the charge;
- what stops the run;
- what extra results to report.

Nothing is abstract: a bare `class MyUnit(MultiPhaseVessel): oper_mode = 'Batch'` already solves. Override only what differs.

Class attributes

Attribute	Default	Set it to
<code>oper_mode</code>	<code>None</code>	<code>'Batch'</code> , <code>'Semibatch'</code> or <code>'Continuous'</code> . Required for flowsheets: <code>Connections</code> , <code>CheckModule</code> , <code>Outlet</code> , <code>is_continuous</code> and the <code>Phases</code> setter all branch on it. With <code>None</code> , <code>SolveFlowsheet</code> stops in <code>CheckModule</code> with <code>KeyError: None</code> . Declare it on the class, not in <code>__init__</code>
<code>dilutes_intensive_states</code>	<code>False</code>	<code>True</code> when a phase-carried mechanism stores its state per unit volume, as crystallizers do. The vessel then calls each mechanism's <code>get_dilution_contributions</code>
<code>default_output</code>	<code>None</code>	The key a flowsheet picks when <code>Outlet</code> returns a dict (the Filter's is <code>'cake'</code>)
<code>connection_level_operating_names</code>	<code>('vol_flow',)</code>	Rarely changed: operating keys that describe a whole stream rather than one phase

Hooks for layout and wiring (stage 2: charge)

Method	Default	Override when
<code>_post_set_phases(self)</code>	Material states, then default states, connections and nomenclature	You must wire things that need the phases: build mechanisms and <code>intraphase_processes</code> , or re-point a mechanism at a new liquid. Call <code>super()</code> first. It runs again on every flowsheet hand-over

Method	Default	Override when
<code>_material_state_definition(self)</code> returns <code>StateVariable</code>	<code>StateVariable(name=self.basis, dim=num_species, units, index=name_species)</code>	You must change flags on every phase inventory, such as <code>limit_negative_inventory=False</code> when a terminal event is the guard (SolidLiquidSep_Refactored.py:89)
<code>default_diff_states_from_phases(self)</code>	Every phase's <code>mass_j</code> is integrated	Some phase's inventory is not a species mass. Crystallizer: only <code>liquid0</code> , because the solid's inventory is the population (Crystallizers_Refactored.py:105). Filter: liquid only. DeliquoringStep: none
<code>configure_default_connections(self)</code>	No outlet	The unit discharges a flow. Return at once if <code>self.outlet_connections</code> is already set; otherwise build <code>StreamConnection(stream=self.Phases.to_stream(), phase_mappings=[PhaseMapping(source_phaseref=ref, sink_phaseref=ref), ...])</code> . Map only the phases that leave: the Filter maps the liquid alone
<code>define_output_states(self, overwrite=False)</code>	Total mass, <code>vessel_vol</code> , <code>outlet_vol_flow</code> , <code>global_temp</code> , liquid <code>mole_conc</code> , and the mechanisms' outputs	You want an extra result, or must drop one that cannot be computed. Call <code>super()</code> , then <code>self.output_state_collection.add(StateVariable(name, dim, units, state_type='post', compute_value=f), overwrite)</code> , or <code>.states.pop(key)</code>

An output's `compute_value` is called as `f(state_var, time, completed_state, context, resolved_inlets=None, resolved_outlets=None, operating_conditions=None)` for every saved time after the solve. `context` is a deep copy of the unit used for the replay, so write `f` as a `@staticmethod` that reads from `context`, not from `self`.

Hooks for the solve (stage 5)

Method	Default	Override when
<code>compile_structure(self)</code>	Layouts, buffers, events	Constants follow from the charge (Darcy constants, cake geometry). Call <code>super()</code> , then compute them. Runs at the top of every <code>solve_unit</code>
<code>configure_solver(self)</code>	Nothing	The system is large and sparse: <code>self.integrator.set_linear_solver('krylov')</code> , as crystallizers and DeliquoringStep do

Method	Default	Override when
<code>get_events(self)</code> returns a list of <code>StateEvent</code>	The controller's events, the mechanisms' events and the constructor's <code>state_events</code>	The unit has its own stopping or switching condition. <code>events = list(super().get_events())</code> , append yours, return. Unlike the controller's and mechanism's versions it takes no <code>unit</code> argument
<code>reset(self)</code>	Resets the controller, <code>elapsed_time</code> and the mechanisms, and restores the initial state	You keep per-run state on the unit. Call <code>super()</code>
<code>retrieve_results(self, time, solver_states)</code>	Builds <code>self.result</code> and returns it	You post-process the final state. Call <code>super()</code> and keep its return value

Hooks inside the balances (every right-hand side)

Method	Default	Override when
<code>get_heat_transfer_area(self)</code>	<code>4*V_liquid/diam + area_base</code> (raises if <code>diam</code> is 0)	The geometry is not a stirred tank. The Filter returns its filter area
<code>get_heat_transfer_temperature(self)</code>	First phase's temperature	Heat is exchanged by a different phase
<code>add_mixing_energy_terms / add_shaftwork_energy_terms(self, contributions, time, completed_state)</code>	Add 0	Add W to <code>contributions['mixing']</code> or <code>contributions['shaftwork']</code>
<code>compute_requested_phase_outlet_flow(self, vessel_phase, total_outlet_flow, connection)</code>	Splits a connection's flow over its phases by volume	Phases leave in another ratio (a settler, a decanter)
<code>compute_actual_phase_outlet_flow(self, vessel_phase, requested_flow)</code>	<code>max(requested_flow, 0)</code>	A physical limit applies, such as hydraulics or weir height

A convenience setter

This is the pattern `RxnKinetics` and `CrystKinetics` follow. The getter raises and explains, and the setter builds the processes or connections and refreshes the layout:

```

@property
def MyKinetics(self):
    raise AttributeError('MyKinetics is a convenience initializer only; edit
self.intraphase_processes instead.')

@MyKinetics.setter
def MyKinetics(self, kinetics):
    self.intraphase_processes =
[IntraPhaseProcess(phaseref=PhaseRef('liquid', 0),

mechanism=MyMechanism(kinetics))]
    # the intraphase_processes setter already calls
self.nomenclature(overwrite=True)

```

What a subclass can read

The phases:

- `self.Phases`;
- `self.Phases.get_phase_from_ref(PhaseRef('liquid', 0))`;
- `self.Phases.Liquids` and `self.Phases.Solids`;
- `self.phase_states.get_phase(ref)`, which is the live phase during a solve.

The pieces plugged into the unit: `self.controller`, `self.integrator` and `self.Utility`.

Run state and results: `self.elapsed_time`, `self.result` and `self.outlet_conditions`.

Inside a right-hand side or an event function, the current values are in `completed_state[StateKey(name, phaseref)]`. For example, `completed_state[StateKey('mass_j', PhaseRef('liquid', 0))]` gives the liquid's species masses in kg.

How the existing units do it

Every refactored family has the same two layers, and a new unit should copy them:

- **A family base** (`_BaseReactor`, `_BaseCrystallizer`, `_BaseSolidLiquidSep`) holds the physics wiring.
- **Thin mode classes** (`Batch...`, `SemiBatch...`, `Continuous...`) set `oper_mode`, a default controller and, for continuous units, the outlet.

Family bases: the physics wiring

Family base	What it overrides	Why
<code>_BaseReactor</code> (Reactors_Refactored.py:6)	<code>RxnKinetics</code> setter: one <code>IntraPhaseProcess</code> with a <code>ReactionMechanism</code> per liquid. Asserts the unit is not both isothermal and adiabatic	Reactions happen inside one phase, so an intraphase process is the whole wiring. Every phase keeps its default integrated <code>mass_j</code>
<code>_BaseCrystallizer</code> (Crystallizers_Refactored.py:6)	<code>dilutes_intensive_states = True</code> . The <code>CrystKinetics</code> setter builds <code>phase_connections</code> from a <code>ReversibleTransferMechanism</code> , <code>.forward</code> liquid→solid and <code>.reverse</code> solid→liquid. <code>default_diff_states_from_phases</code> : only <code>liquid0</code> 's mass. <code>configure_solver</code> : Krylov. <code>_post_set_phases</code> : re-points the population balance at the live liquid	The solid's inventory is the population, a mechanism state, not a species mass. The population is per m ³ of slurry, so a feed dilutes it. A flowsheet hand-over replaces the liquid object

Family base	What it overrides	Why
<code>_BaseSolidLiquidSep</code> (<code>SolidLiquidSep_Refactored.py:49</code>)	<code>default_diff_states_from_phases :</code> liquid only. <code>_material_state_definition :</code> <code>limit_negative_inventory=False .</code> <code>get_heat_transfer_area :</code> the filter area	The cake is drained, never consumed. A terminal event, not the positivity limiter, stops the drainage

Mode classes: operating mode, controller, outlet

Mode	oper_mode	Default controller	Outlet	Feed
Batch	'Batch'	<code>Controller()</code> (does nothing)	None. <code>Outlet</code> returns the final <code>Phases</code>	None. A flowsheet hands material over as the next charge. A continuous upstream is refused: put a holding vessel between them
Semibatch	'Semibatch'	<code>Controller()</code>	None	<code>unit.Inlet = stream</code> ; the volume grows
Continuous	'Continuous'	<code>DefaultContinuousVesselVolume()</code> (holds the volume with the outlet), built inside <code>__init__</code> so instances never share one	<code>configure_default_connections :</code> one outlet mapping every phase (<code>Reactors_Refactored.py:79</code>)	<code>unit.Inlet = stream</code>

The continuous classes take `controller=None` and build the default inside `__init__`, not as a default argument. A default argument is evaluated once, so every unit in the session would share one controller and its `target_volume`.

Units that need more than the pattern

Unit	What it adds
Filter (SolidLiquidSep_Refactored.py:159)	The constructor defaults are <code>isothermal=True</code> and a <code>FiltrationController</code> that sets the filtrate draw from Darcy's law (an operating condition, so the model stays an ODE on every backend). <code>compile_structure</code> computes the cake constants from the charge. <code>configure_default_connections</code> maps the liquid only. <code>get_events</code> adds a terminal "filtrate exhausted" event. <code>Outlet</code> returns <code>{'cake': ..., 'filtrate': ...}</code> with <code>default_output = 'cake'</code> . <code>retrieve_results</code> rescales the cake's population after drainage
DeliquoringStep (:671)	<code>default_diff_states_from_phases</code> integrates no phase mass at all. <code>_post_set_phases</code> builds a <code>DeliquoringMechanism</code> (the saturation and concentration profiles are mechanism states) and attaches it as an intraphase process. <code>define_output_states</code> drops <code>mole_conc</code> , which needs the missing <code>mass_j</code> . <code>compile_structure</code> computes the constants and initial profiles

The Filter shows a choice worth copying. Its physics, Darcy's law, lives in plain methods on the unit, and a controller turns it into an outlet flow. That way the vessel's own liquid balance does the integrating and no new state is needed.

Worked example: a batch evaporator (tested)

This is a new unit that PharmaPy does not have. It boils solvent off a batch charge at a rate set by the heat duty, and stops when a solute reaches a target mass fraction. It uses five of the hooks above:

- a class attribute (`oper_mode`);
- constructor defaults;

- `_post_set_phases`, to attach its mechanism;
- `define_output_states`, for an extra result;
- `get_events`, for a terminal stop.

```
import numpy as np
from PharmaPy.MultiPhaseVessel import MultiPhaseVessel
from PharmaPy.Mechanisms import Mechanism
from PharmaPy.DataClasses import (IntraPhaseProcess, PhaseRef, StateEvent,
                                   StateKey,
                                   StateVariable, TransferResult)

LIQUID = PhaseRef('liquid', 0)

class SolventBoilOff(Mechanism):
    """Removes one species from the liquid at a fixed mass rate (it leaves as
    vapour)."""

    def __init__(self, rate, species_index):
        super().__init__()
        self.rate, self.species_index = rate, species_index

    def get_overrides(self):
        return {}

    def get_solver_state_rates(self, process, phase, time, completed_state):
        rates = np.zeros(phase.num_species)
        rates[self.species_index] = -self.rate # kg/s
        return TransferResult(state_rates={StateKey('mass_j',
process.phaseref): rates},
                               aux={'phase': phase, 'process': process},
                               net_mass_rate=self.rate)

class BatchEvaporator(MultiPhaseVessel):
    """Boils solvent off a batch charge until a solute reaches a target mass
    fraction."""

    oper_mode = 'Batch' #
    Connections, SimExec, CheckModule read it

    def __init__(self, duty, latent_heat, target_species, target_frac,
```

```

solvent='solvent', **kwargs):
    # your own settings first: the hooks below read them
    self.duty, self.latent_heat = duty, latent_heat          # W,
J/kg
    self.target_species, self.target_frac = target_species, target_frac
    self.solvent = solvent
    kwargs.setdefault('isothermal', True)                  #
boiling: T stays put
    super().__init__(**kwargs)

    @property
    def boil_off_rate(self):
        return self.duty / self.latent_heat                # kg/s

    # --- hooks -----
    def _post_set_phases(self):
        super()._post_set_phases()                          #
material states, connections, nomenclature
        liquid = self.Phases.get_phase_from_ref(LIQUID)
        boil_off = SolventBoilOff(self.boil_off_rate,
liquid.name_species.index(self.solvent))
        self.intraphase_processes = [IntraPhaseProcess(phaseref=LIQUID,
mechanism=boil_off)]

    def define_output_states(self, overwrite=False):
        super().define_output_states(overwrite)
        self.output_state_collection.add(
            StateVariable(name='evaporated', dim=1, units='kg',
state_type='post',
                        compute_value=self.compute_evaporated), overwrite)

    @staticmethod
    def compute_evaporated(state_var, time, completed_state, context,
**kwargs):
        return context.boil_off_rate * time

    def get_events(self):
        events = list(super().get_events())                # keep
the controller's and mechanisms'

    def target_reached(time, completed_state, unit):
        # written on masses, not on the fraction: it falls once through
zero and stays below

```

```

        mass = completed_state[StateKey('mass_j', LIQUID)]
        i = unit.Phases.name_species.index(unit.target_species)
        return mass.sum() - mass[i] / unit.target_frac

        events.append(StateEvent(name='target_reached',
function=target_reached,
                                direction=-1, terminal=True, source=self))

        return events

```

On its own

```

ev = BatchEvaporator(duty=2000, latent_heat=4e5, target_species='C',
target_frac=0.25)
ev.Phases = LiquidPhase(path, temp=333.15, mass=1.0, mass_frac=[0, 0, 0.1, 0,
0.9], name_solv='solvent')
ev.solve_unit(runtime=600, verbose=False)
ev.result.time[-1], ev.result.evaporated[-1], ev.result.mass_j_liquid0[-1]

```

The charge is 1 kg at 10 wt% C. Boiling off 5 g/s until C is 25 wt% removes 0.6 kg, which takes 120 s. The run stops at $t = 120.000$ s with $w_C = 0.25000$. `result.evaporated` reads 0.6000 kg, and the 0.1 kg of C is untouched.

In a flowsheet

```

flst = SimulationExec(path, flowsheet='R01 --> EV01')
flst.R01 = BatchReactor(integrator=ScipyBackend(), isothermal=True)
flst.R01.Phases = LiquidPhase(path, temp=333.15, vol=1e-3, mole_conc=[1, 1,
0, 0, 0], name_solv='solvent')
flst.R01.RxnKinetics = RxnKinetics(path, rxn_list=['A + B --> C'], k_params=
[5e-3], ea_params=[40e3], temp_ref=333.15)
flst.EV01 = BatchEvaporator(duty=2000, latent_heat=4e5, target_species='C',
target_frac=0.25)
flst.EV01.Phases = LiquidPhase(path, temp=333.15, mass=1.0, mass_frac=[0, 0,
0, 0, 1], name_solv='solvent') # placeholder
flst.SolveFlowsheet(kwargs_run={'R01': {'runtime': 3600, 'verbose': False},
                                'EV01': {'runtime': 3600, 'verbose': False}},
verbose=False)

```

The evaporator receives exactly the reactor's product: 0.89555 kg, holding 0.18976 kg of C. It stops at 27.3 s with $w_C = 0.2500$, which matches the hand calculation (0.13651 kg at 5 g/s). The placeholder charge is replaced on hand-over. `_post_set_phases` runs again and

re-attaches the mechanism to the new liquid.

What it took

1. **The first version of the event was wrong, and it failed silently.** It was written as `target_frac - m_C/m_total`. That ratio blows up and flips sign again once the solvent runs out. Alone, the solver stepped from before 120 s to after 200 s, saw the same sign at both ends, and ran to 600 s with a negative solvent mass. Written on masses, the event is linear and is found exactly.
2. **Nothing stops the boil-off on its own.** The vessel's positivity limiter throttles outlet and crossphase transfers only. An intraphase mechanism that removes material must be stopped by an event or guard itself, as `ReactionMechanism` does with `positivity_horizon`.

Flowsheets: connecting to other units

A subclass inherits everything `SimulationExec` needs. The base class carries a translation layer for the flowsheet code (`MultiPhaseVessel.py:386-551`), and `Connections` recognises any class whose MRO contains `MultiPhaseVessel` as a refactored unit. The evaporator above ran behind a reactor without one line of flowsheet code. What a new unit has to get right is below.

What `SolveFlowsheet` does with each unit

1. `check_modeling_objects(unit, name)` (`CheckModule.py:18`) looks up `unit.oper_mode` in `data/minimum_modeling_objects.json`:
 - `'Batch'` needs `Phases`;
 - `'Semibatch'` and `'Continuous'` need `Phases` and `Inlet` (a non-empty `inlet_connections` counts);
 - an unset `oper_mode` raises `KeyError: None` here.
2. `unit.solve_unit(**kwargs_run[name])`, then `unit.flatten_states()` (a no-op on the vessel).
3. A `Connection(unit, next_unit)` (`Connections.py:284`) moves the product on:
 - **FeedConnection** takes `unit.Outlet`. If that is a dict, it takes `Outlet[unit.default_output]`. It attaches `unit.outputs` as the trajectory, over the whole run if `unit.is_continuous`, otherwise at the final time.
 - **ConvertUnits** maps `unit.names_states_out` onto `next_unit.names_states_in`.

- **PassPhases** sets the destination by its `oper_mode`, as in the table below.

Destination <code>oper_mode</code>	Receives the material as	Notes
'Batch'	<code>next_unit.Phases = matter</code>	Merged by phase family into the configured phases, so a configured solid with its mechanism survives a liquid-only hand-over. <code>_post_set_phases</code> runs again. A continuous source is refused with a <code>TypeError</code> : put a semibatch unit between them as a holding tank
'Semibatch'	<code>Inlet</code> if the source is continuous, <code>Phases</code> otherwise	A semibatch destination fed by a batch unit treats the material as a charge
'Continuous'	<code>next_unit.Inlet = matter</code>	

What the base class already provides

Member	What it gives the flowsheet
<code>Outlet</code>	Before a solve, the outlet connection's stream. After a solve, the resolved outlet stream. For a unit with no outlet connection (batch, semibatch), the final <code>Phases</code>
<code>outputs</code>	<code>{'mole_conc', <population>, 'temp', 'vol_flow' or 'vol'}</code> , read from <code>result.mole_conc_liquid0</code> , <code>result.<population>_solid0</code> , <code>result.global_temp</code> , and <code>result.outlet_vol_flow</code> or <code>result.vessel_vol</code> . Missing entries are skipped
<code>names_states_out</code> / <code>names_states_in</code>	The same names, as lists. An inlet says <code>vol_flow</code> even on a semibatch unit

Member	What it gives the flowsheet
<code>is_continuous</code> , <code>states_di</code> , <code>flatten_states</code> , <code>material_from_upstream</code> , <code>bipartite</code> , <code>names_upstream</code>	Bookkeeping the flowsheet reads or sets; leave them alone

What a new unit may need to add

- **Two products:** return a dict from `Outlet` and set `default_output` to the one a flowsheet takes by default. The Filter returns `{'cake': ..., 'filtrate': ...}` with `default_output = 'cake'`.
- **A solid that carries a population:** give its mechanism a `flow_state_name` (FVM: `distrib`). `outputs` and `names_states_out` find it there.
- **A different liquid basis:** if the unit has no liquid `mass_j`, so there is no `mole_conc_liquid0` (`DeliquoringStep`), `outputs` simply omits it. Make sure the downstream unit does not need it.
- **Parameters belong in the constructor, not in `kwargs_run`.** `kwargs_run[name]` goes to `solve_unit`, which forwards unknown keywords to the integrator, where they land in the solver options. The Filter takes `deltaP` in its constructor for this reason.

Placeholder charges

In a flowsheet, a downstream batch unit is usually given a placeholder `Phases` before the run, so its mechanisms and kinetics can be attached. That is what the workshop notebooks and the evaporator example do. The hand-over replaces the placeholder phase by phase family. Make sure everything your unit wires to a phase is re-wired in `_post_set_phases`: the crystallizers re-point the population balance at the new liquid, and the evaporator re-attaches its mechanism.

Checklist and pitfalls

Each item below either fails silently or fails far from its cause. The ones marked (*tested*) were reproduced while writing this doc.

The class

- ☐ `oper_mode` is set **on the class** to `'Batch'`, `'Semibatch'` or `'Continuous'`. Without it, `SolveFlowsheet` stops with `KeyError: None in CheckModule (tested)`.
- ☐ `__init__` takes `**kwargs` and forwards them whole to `super().__init__(**kwargs)`. Repeating the base signature means every parameter later added to `MultiPhaseVessel` silently stops reaching your unit.
- ☐ Defaults go in with `kwargs.setdefault(...)`, so a user can still override them.
- ☐ A default controller is built inside `__init__`, never as a default argument value, which every instance would share.
- ☐ Users assign `Phases` after construction. `MyUnit(Phases=...)` fails today with `AttributeError: ... 'phase_states' (tested)`.

States and wiring

- ☐ Every phase inventory marked differential has something that supplies its rate. Otherwise override `default_diff_states_from_phases` so that phase is not integrated. A solid tracked by a population is not integrated as `mass_j`.
- ☐ Every output you keep can be computed from the states you have. Drop `mole_conc` if there is no liquid `mass_j`, as `DeliquoringStep` does.
- ☐ Output `compute_value` functions are `@staticmethod`s that read from `context`, the replay copy, not from `self`.
- ☐ Everything wired to a phase is (re)built in `_post_set_phases`, after calling `super()`. It runs again whenever a flowsheet hands material over.
- ☐ Continuous outlets: `configure_default_connections` returns early when `outlet_connections` is already set, and maps only the phases that actually leave.
- ☐ Constants that depend on the charge are computed in `compile_structure` after `super()`, not in `__init__`. The charge may arrive later, from upstream.

Stopping and limits

- ☐ A unit-level event function changes sign **once** and stays changed. Write it on masses or volumes, not on a ratio whose denominator can reach zero. A ratio can flip back between two solver steps, and the stop is then missed without an error (*tested: the evaporator ran past its target and drove the solvent negative*).
- ☐ `get_events(self)` extends `super().get_events()`, so the controller's and mechanisms' events are kept. It takes no `unit` argument.

- ☐ Anything an intraphase mechanism removes is bounded, by a terminal event or by the mechanism itself. The vessel's positivity limiter only throttles outlet and crossphase transfers (MultiPhaseVessel.py:2118).
- ☐ When a terminal event is the real guard, as for the Filter, the limiter is switched off for that inventory (`limit_negative_inventory=False`), or it will slow the process down before the event fires.
- ☐ Energy: a unit with no stirred-tank geometry either defaults to `isothermal=True` or overrides `get_heat_transfer_area` . The default raises when `diam` is 0.

Using and testing it

- ☐ Model parameters go in the constructor. `kwargs_run` and `solve_unit` keywords reach the integrator.
- ☐ Build a new unit per simulation. `solve_unit` on a solved unit continues from `elapsed_time` .
- ☐ Test the unit on its own against a hand calculation, then inside a flowsheet, and check that the downstream unit received exactly what the upstream one produced. The evaporator matched both (120.000 s alone; 0.89555 kg handed over and 27.3 s in the flowsheet).
- ☐ If the unit needs a new mechanism or controller, check those against the companion doc [Mechanism, integrator and controller API](#) as well.