**Привет, мы создаем ядро платформы, которая занимается обработкой данных путем движения этих данных от одного элемента к другому. В прикрепленном документе собран код текущего состояния и структура проекта. Важные комментарии:**

**1. Особенность архитектуры. Все объекту, которые обрабатывают (преобразуют) информацию представляют собой объекты класса Element или его наследники. Каждый такой объект представляется черным ящиком. У него есть входные порты (куда информация входит) и выходные порты, из которых информация выходит в обработанном виде. Также у них есть набор параметров (внутренних), которые могут быть настроены. Пример: элемент насос. У него будет входной порт, который должен содержать информацию о рабочем теле (температура, расход, давление, информация о среде) и выходной порт (содержит туже самую информацию что и входной), при этом у него есть набор внутренних параметров (рабочая характеристика, КПД и т.д.). Задача элемента - осуществить метод calculate на основе известных данных на входных, выходных портах и внутренних параметров. Аналогично таким элементом может быть нейронная модель: у нее есть 3 входных порта (train, validate, evaluate) и один выходной порт (ответ), внутренними параметрами могут быть архитектура нейронной сети, режим работы (обучение/инференс) и т.д.. Т.е. мы пытаемся создать универсальный инструмент пайплайна обработки данных.**

**2. В сердце величина (класс Value) - каждая содержит свою спецификацию (ValueClass), которая отдельно регистрируется в файле конфигерации. ValueClass - это описание физической (или любой другой величины) на базовом уровне - указывается к какому направлению физики относится, какая размерность, какое название. Например, параметр Value у насоса может быть "Температура на входе" с именем t1 (Но ValueClass у нее именно температура), а может быть у вохздуха "температура воздуха" с именем ta (но Но ValueClass у нее тоже температура). Эти ClassValue жестко регистрируются в файле конфигурации value\_classes и обрати внимание - имя величины из разных разделов физики может совпадать (давление в "Thermodynamics" и давление в "Mechanics").**

**3. Сами величины Value имеют статусы - важный атрибут, определяемый классом ValueStatus. Это сделано для организации соединений между элементами. Если в процессе вызова функции calculate объект типа element рассчитал какойто из параметров, то он назначит этому параметру статус CALCULTED, если пользователь перед расчетом задал какойто параметр, то ему установится статус FIXED, если величина на данный момент неизвестна, то у нее статус UNKNOWN, а если данные о величине пришли от соединенного с этим элементом другого элемента, то у нее установится статус DEPEND**

**4. Все элементы, величины, порты, регистрируются через id, реализация хранения элементов представлена в классе ObjectRepository.**

**5. Соединение элементов описывается классом Connection.py. Есть жесткое правило - выходные порты могут соединяться только с входными**

**6. Фабрика объектов реализована классом ElementFactory**

**Наша задача сейчас - АККУРАТНО переработать класс ElementFactory - очевидно он должен создавать объекты класса Element или наследников класса Element, информация о которых зарегестрирована в файлах конфигурации вот какие возможны сценарии:**

**1. Создается от базового класса Element (пример — dummy.json), тогда class\_path: null**

**2. Создается от унаследованного от Element класса (тогда в class\_path мы указываем наследника, пример pump.json)**

**Как видишь, у нас есть возможность назначать порты прямо в файле конфигурации, либо пользоваться уже существующими.**

**Важно учесть, что конструктор унаследованного класса может быть неизвестен фабрике классов. Как например ты видешь, что в pump.json «in\_ports» вообще содержит список.**

**Идея такая — в файле конфигурации есть набор обязательных полей:** "name", "author", "version", "description", "category", "class\_path", "in\_ports", "out\_ports", "parameters", "functions". Эиа информация, которая обрабатывается фабрикой классов и частично передается в конструктор элемента или наследника этого элемента. Тип данных в этих полях не меняется (т. е. in\_ports это всегда список, functions это всегда словарь и т.д.) Если в конфигурации есть другие поля, то они передаются в конструктор (предварительно обрабатываются фабрикой классов, о чем обьясню позже)

Далее, как ты видишь у элемента «pump.json» в in\_ports нет каких либо сигнатур, говорящих о порте (т. к. конструктор унаследованного класса использует это поле для других вещей). Вот тут нам надо обозначить сигнатуру, когда фабрике классов создавать класс Port, а когда просто передавать в конструктор информацию из конфигурации. Тоже самое для parameters, functions. А также если в конфигурации есть дополнительные поля, то пробежаться по ним и проверить, если сигнатуры для создания каких то объектов. Например, если в каком-то элементе унаследованоом нет от Element (т. е. У него class\_path не null), есть 2 дополнительных поля такого типа:

«new\_param»: [0, 23, {«Port»: {"name": "inlet1", "use\_port": "thermodinamyc\_port"}}, {«value»: { "param\_name": "G", "value\_class": "Thermodynamics.G", "min\_value": 0 }}]

«new\_param\_2»: {«value»: { "param\_name": "G2", "value\_class": "Thermodynamics.G", "min\_value": 0 }}

то фабрика классов должна будет передать в конструктор этого объекта дополнительно параметр new\_param=[float, Port, Value] и new\_param\_2 = Value.

Это естественно не касается объектов, которые являются объектами класса Element, т. е. class\_path: null

Важно: пример с подобной сигнатурой я придумал как пример! Предложи наиболее эффективный способ подсвечивания для фабрики классов, когда нужно создавать объекты (которыми мы можем оперировать: Value, Port) , а когда надо передавать в конструктор «сырую» не обработанную информацию. Важно, чтобы для объектов типа Element не усложнялся синтаксис создания файлов конфигурации, т. к. для них сохраняются четкие правила создания элементов.

Список задач:

1. Предложи полный код переработанной фабрики элементов

2. Создай файлы pump.json и dummy.json в соответствие с изменениями подхода к обработке конфигураций из пункта 1.

3. Посмотри на структуру проекта. Четко пропиши, каким образом необходимо импортировать модули, чтобы все работало внутри файлов проекта (может, где-то \_\_init.py\_\_ добавить, может в \_\_init.py\_\_ дописать код). Мне нужен четкий сформированный подход для развития проекта, чтобы все нормально и правильно импортировалось

**Файл Value.py**

import uuid

import numpy as np

from enum import Enum, auto

from typing import Dict, Iterator, List, Optional, Tuple, Union, Any, Collection, Type, Callable

import copy

from dataclasses import dataclass

class ValueStatus(Enum):

"""Статусы значений для отслеживания состояния параметра"""

UNKNOWN = auto()

DEPEND = auto() # Значение зависит от других параметров

CALCULATED = auto() # Рассчитано в процессе вычислений

FIXED = auto() # Фиксированное значение (не изменяется)

@classmethod

def from\_input(cls, status\_input: Union[str, 'ValueStatus']) -> 'ValueStatus':

"""Преобразует различные форматы в объект ValueStatus"""

if isinstance(status\_input, ValueStatus):

return status\_input

elif isinstance(status\_input, str):

normalized = status\_input.upper().strip()

for status in cls:

if status.name == normalized:

return status

raise ValueError(f"Неизвестный статус: {status\_input}. "

f"Допустимые значения: {', '.join([s.name for s in cls])}")

else:

raise TypeError(f"Неподдерживаемый тип статуса: {type(status\_input)}")

@dataclass

class ValueClass:

value\_name: str = None #Название величины (например, энтальпия, энтропия и т.д. - это не конкретное название параметра для данного элемента)

physics\_type: str = None #Физическое направление, к которому относится параметр (например, термодинамика, механика и т.д.)

dimension: str = None #Размерность

def \_\_eq\_\_(self, other: 'ValueClass'):

if not isinstance(other, ValueClass):

return NotImplemented

elif self.physics\_type == other.physics\_type and self.value\_name == other.value\_name and self.dimension == other.dimension:

return True

else:

return False

def \_\_ne\_\_(self, other: 'ValueClass'):

return not self.\_\_eq\_\_(other)

def combine\_dims(dim1: Optional[str], dim2: Optional[str], op: str) -> Optional[str]:

"""Комбинирует размерности для арифметических операций"""

# Обе размерности отсутствуют

if dim1 is None and dim2 is None:

return None

# Операции сложения/вычитания

if op in ['+', '-']:

if dim1 == dim2:

return dim1

return None

# Умножение

if op == '\*':

if dim1 is None:

return dim2

if dim2 is None:

return dim1

if dim1 == dim2:

return f"({dim1})^2"

return f"({dim1})\*({dim2})"

# Деление

if op == '/':

if dim1 is None:

return f"1/({dim2})" if dim2 else None

if dim2 is None:

return dim1

if dim1 == dim2:

return "1"

return f"({dim1})/({dim2})"

return None

class Value:

def \_\_init\_\_(self,

name: str,

value\_spec: ValueClass,

value: Any,

description: str = "",

status: ValueStatus = ValueStatus.UNKNOWN,

store\_prev: bool = True,

min\_value: Optional[Any] = None,

max\_value: Optional[Any] = None):

"""

Инициализация параметра

:param name: Имя параметра (идентификатор) - например, имя параметра h1 (а )

:param: value\_spec: Спецификация параметра (относится к классу ValueClass)

:param value: Значение параметра (любого типа)

:param description: Описание параметра

:param status: Исходный статус значения

:param store\_prev: Флаг сохранения предыдущих значений

:param min\_value: Минимальное значение, допустимое для данного параметра

:param max\_value: Максимальное значение. допустимое для данного параметра

"""

self.\_name = name

self.\_description = description

self.\_status = ValueStatus.from\_input(status)

self.\_store\_prev = store\_prev

self.\_prev\_val = None

self.\_prev\_status = None

self.\_min\_value = min\_value

self.\_max\_value = max\_value

self.\_value\_spec = value\_spec

# Установка значения с валидацией

self.\_val = None

self.value = value # Используем сеттер для валидации

# Сохраняем исходное значение как предыдущее

if self.\_store\_prev:

self.\_save\_previous()

def \_save\_previous(self):

"""Сохранение текущего состояния как предыдущего"""

self.\_prev\_val = self.\_try\_copy(self.\_val)

self.\_prev\_status = self.\_status

@property

def value(self) -> Any:

"""Текущее значение параметра"""

return self.\_val

@value.setter

def value(self, new\_value: Any):

"""Установка нового значения без изменения статуса"""

try:

self.update(new\_value)

except ValueError as e:

raise ValueError(f"Ошибка установки значения для {self.name}: {str(e)}")

@property

def dimension(self) -> Optional[str]:

"""Физическая размерность параметра"""

return self.\_value\_spec.dimension

@property

def physics\_type(self) -> Optional[str]:

"""К какой физической группе относится"""

return self.\_value\_spec.physics\_type

@property

def physics\_value\_name(self) -> str:

"""Физическое название величины"""

return self.\_value\_spec.value\_name

@property

def name(self) -> str:

"""Имя параметра"""

return self.\_name

@property

def description(self) -> str:

"""Описание параметра"""

return self.\_description

@property

def status(self) -> ValueStatus:

"""Текущий статус значения (только для чтения)"""

return self.\_status

@property

def value\_type(self) -> Type:

"""Тип хранимого значения"""

return type(self.\_val)

@property

def store\_prev(self) -> bool:

"""Флаг сохранения предыдущих значений"""

return self.\_store\_prev

@store\_prev.setter

def store\_prev(self, flag: bool):

"""Изменение флага сохранения истории"""

self.\_store\_prev = flag

if not flag:

self.\_prev\_val = None

self.\_prev\_status = None

elif self.\_prev\_val is None:

self.\_save\_previous()

@property

def previous\_value(self) -> Optional[Any]:

"""Предыдущее значение параметра (если доступно)"""

return self.\_prev\_val

@property

def previous\_status(self) -> Optional[ValueStatus]:

"""Предыдущий статус параметра (если доступен)"""

return self.\_prev\_status

@property

def min\_value(self) -> Optional[Any]:

"""Минимально допустимое значение"""

return self.\_min\_value

@property

def max\_value(self) -> Optional[Any]:

"""Максимально допустимое значение"""

return self.\_max\_value

def \_validate\_value(self, value: Any):

"""Проверка значения на соответствие границам"""

if value is None:

return # None всегда разрешен

# Проверка минимального значения

if self.\_min\_value is not None:

try:

if np.any(value < self.\_min\_value):

raise ValueError(f"Значение {value} меньше минимального {self.\_min\_value}")

except TypeError:

# Для нестандартных типов используем оператор <

if value < self.\_min\_value:

raise ValueError(f"Значение {value} меньше минимального {self.\_min\_value}")

# Проверка максимального значения

if self.\_max\_value is not None:

try:

if np.any(value > self.\_max\_value):

raise ValueError(f"Значение {value} больше максимального {self.\_max\_value}")

except TypeError:

# Для нестандартных типов используем оператор >

if value > self.\_max\_value:

raise ValueError(f"Значение {value} больше максимального {self.\_max\_value}")

def update(self, new\_value: Any, new\_status: Optional[ValueStatus] = None):

"""

Обновление значения и статуса с валидацией

:param new\_value: Новое значение

:param new\_status: Новый статус

"""

# Валидация нового значения

self.\_validate\_value(new\_value)

# Сохраняем текущее состояние перед обновлением

if self.\_store\_prev:

self.\_save\_previous()

# Устанавливаем новые значения

self.\_val = new\_value

if new\_status is not None:

self.\_status = new\_status

def set\_bounds(self, min\_value: Optional[Any] = None, max\_value: Optional[Any] = None):

"""

Установка новых граничных значений с валидацией текущего значения

:param min\_value: Новое минимальное значение

:param max\_value: Новое максимальное значение

"""

# Сохраняем текущие границы

old\_min = self.\_min\_value

old\_max = self.\_max\_value

try:

# Временно устанавливаем новые границы

self.\_min\_value = min\_value

self.\_max\_value = max\_value

# Проверяем текущее значение на соответствие новым границам

self.\_validate\_value(self.\_val)

except ValueError:

# В случае ошибки восстанавливаем старые границы

self.\_min\_value = old\_min

self.\_max\_value = old\_max

raise

def get\_residual(self) -> Optional[Any]:

"""

Вычисление разницы между текущим и предыдущим значением

:return: Разница значений или None, если:

- предыдущее значение недоступно

- предыдущий статус UNKNOWN

- операция вычитания не поддерживается

"""

# Не вычисляем residual если предыдущее значение не сохранено

if self.\_prev\_val is None:

return None

# Не вычисляем residual для UNKNOWN статуса

if self.\_prev\_status == ValueStatus.UNKNOWN:

return None

try:

# Для numpy-массивов

if isinstance(self.\_val, np.ndarray):

return self.\_val - self.\_prev\_val

# Для тензоров TensorFlow

if hasattr(self.\_val, 'numpy') and hasattr(self.\_prev\_val, 'numpy'):

return self.\_val.numpy() - self.\_prev\_val.numpy()

# Для тензоров PyTorch

if (hasattr(self.\_val, 'detach') and hasattr(self.\_prev\_val, 'detach') and

hasattr(self.\_val, 'numpy') and hasattr(self.\_prev\_val, 'numpy')):

return self.\_val.detach().numpy() - self.\_prev\_val.detach().numpy()

# Общий случай для поддерживающих вычитание

if hasattr(self.\_val, '\_\_sub\_\_'):

return self.\_val - self.\_prev\_val

except (TypeError, ValueError):

pass

return None

def reset\_history(self):

"""Сброс истории предыдущих значений"""

self.\_prev\_val = None

self.\_prev\_status = None

if self.\_store\_prev:

self.\_save\_previous()

def get\_state(self) -> Tuple[Any, ValueStatus]:

"""Получение текущего состояния (значение + статус)"""

return self.\_val, self.\_status

@staticmethod

def \_try\_copy(obj: Any) -> Any:

"""Попытка создания копии объекта"""

try:

# Для numpy-массивов

if isinstance(obj, np.ndarray):

return obj.copy()

# Для тензоров TensorFlow/PyTorch

if hasattr(obj, 'numpy'):

return obj.numpy().copy()

if hasattr(obj, 'detach'):

return obj.detach().clone()

# Общий случай

return copy.deepcopy(obj)

except:

# Если копирование невозможно, возвращаем оригинал

return obj

# Добавляем логические операции сравнения

def \_check\_comparable(self, other: 'Value') -> None:

"""Проверка возможности сравнения двух значений"""

"""Проверка возможности сравнения двух значений"""

if self.dimension != other.dimension:

raise ValueError(f"Несовместимые размерности: {self.dimension} vs {other.dimension}")

if self.physics\_type != other.physics\_type:

raise ValueError(f'Несовместимые физические направления: {self.physics\_type} vs {other.physics\_type}')

if self.physics\_value\_name != other.physics\_value\_name:

raise ValueError(f'Несовместимые физические величины: {self.physics\_value\_name} vs {other.physics\_value\_name}')

if not hasattr(self.\_val, '\_\_eq\_\_') or not hasattr(other.value, '\_\_eq\_\_'):

raise TypeError("Значения не поддерживают операции сравнения")

def \_\_eq\_\_(self, other: 'Value') -> bool:

"""Оператор равенства == с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val == other.value

def \_\_ne\_\_(self, other: 'Value') -> bool:

"""Оператор неравенства != с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val != other.value

def \_\_lt\_\_(self, other: 'Value') -> bool:

"""Оператор меньше < с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val < other.value

def \_\_le\_\_(self, other: 'Value') -> bool:

"""Оператор меньше или равно <= с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val <= other.value

def \_\_gt\_\_(self, other: 'Value') -> bool:

"""Оператор больше > с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val > other.value

def \_\_ge\_\_(self, other: 'Value') -> bool:

"""Оператор больше или равно >= с проверкой размерности"""

if not isinstance(other, Value):

return NotImplemented

self.\_check\_comparable(other)

return self.\_val >= other.value

def compare(self, other: 'Value', operator: str) -> bool:

"""

Универсальный метод сравнения с указанием оператора

:param other: Другой объект Value для сравнения

:param operator: Оператор сравнения ('==', '!=', '<', '<=', '>', '>=')

:return: Результат сравнения

"""

operators = {

'==': self.\_\_eq\_\_,

'!=': self.\_\_ne\_\_,

'<': self.\_\_lt\_\_,

'<=': self.\_\_le\_\_,

'>': self.\_\_gt\_\_,

'>=': self.\_\_ge\_\_

}

if operator not in operators:

raise ValueError(f"Неподдерживаемый оператор: {operator}")

return operators[operator](other)

def \_\_repr\_\_(self) -> str:

"""Обновленное строковое представление с информацией о вызываемости"""

call\_info = ""

if callable(self.\_val):

sig = self.callable\_signature

call\_info = f", callable={sig}" if sig else ", callable=True"

return (f"Value(name={self.name!r}, dimension={self.dimension!r}, "

f"value={self.value}, status={self.status.name}{call\_info})")

def \_\_iter\_\_(self) -> Iterator[Union[Any, ValueStatus]]:

"""Позволяет распаковывать Value как (значение, статус)"""

yield self.value

yield self.status

def \_\_call\_\_(self, \*args, \*\*kwargs) -> Any:

"""

Универсальный вызов значения:

- Если значение является вызываемым объектом, вызывает его с аргументами

- Иначе возвращает само значение (игнорируя аргументы)

:param args: Позиционные аргументы для вызываемого объекта

:param kwargs: Именованные аргументы для вызываемого объекта

:return: Результат вызова или само значение

"""

if callable(self.\_val):

try:

# Пробуем вызвать объект

return self.\_val(\*args, \*\*kwargs)

except Exception as e:

raise RuntimeError(f"Ошибка при вызове значения '{self.name}': {str(e)}") from e

else:

# Для не-функций просто возвращаем значение

return self.\_val

def is\_callable(self) -> bool:

"""Проверяет, является ли хранимое значение вызываемым объектом"""

return callable(self.\_val)

@property

def callable\_signature(self) -> Optional[str]:

"""Возвращает строковое представление сигнатуры вызываемого объекта"""

if not callable(self.\_val):

return None

try:

# Для функций и методов

if hasattr(self.\_val, '\_\_name\_\_'):

name = self.\_val.\_\_name\_\_

else:

name = str(self.\_val)

# Пытаемся получить информацию о параметрах

import inspect

sig = inspect.signature(self.\_val)

return f"{name}{sig}"

except:

return f"Callable object: {type(self.\_val).\_\_name\_\_}"

@staticmethod

def from\_dict(data: Dict[str, Any]) -> 'Value':

"""

Создает экземпляр Value из словаря

:param data: Словарь с параметрами для создания объекта

:return: Экземпляр класса Value

Поддерживаемые ключи:

- 'value' (обязательный): хранимое значение

- 'dimension' (обязательный): размерность величины

- 'name' (обязательный): имя величины

- 'description': описание (по умолчанию "")

- 'status': статус (строка или ValueStatus, по умолчанию UNKNOWN)

- 'store\_prev': флаг сохранения истории (по умолчанию True)

- 'min\_value': минимальное значение (по умолчанию None)

- 'max\_value': максимальное значение (по умолчанию None)

"""

# Проверка обязательных параметров

required\_keys = ['value', 'name', 'value\_spec']

missing = [key for key in required\_keys if key not in data]

if missing:

raise ValueError(f"Отсутствуют обязательные ключи: {', '.join(missing)}")

# Извлечение параметров с установкой значений по умолчанию

value = data['value']

value\_spec = ValueClass(data['value\_spec']['value\_name'],

data['value\_spec'].get('physics\_type', None),

data['value\_spec'].get('dimension', None))

name = data['name']

description = data.get('description', "")

store\_prev = data.get('store\_prev', True)

min\_value = data.get('min\_value', None)

max\_value = data.get('max\_value', None)

# Обработка статуса

status\_data = data.get('status', ValueStatus.UNKNOWN)

if isinstance(status\_data, str):

status = ValueStatus.from\_input(status\_data)

elif isinstance(status\_data, ValueStatus):

status = status\_data

else:

raise TypeError(f"Неподдерживаемый тип для статуса: {type(status\_data)}")

return Value(

name=name,

value\_spec=value\_spec,

value=value,

description=description,

status=status,

store\_prev=store\_prev,

min\_value=min\_value,

max\_value=max\_value

)

def to\_dict(self, include\_private: bool = False) -> Dict[str, Any]:

"""

Преобразует объект Value в словарь

:param include\_private: Включать ли приватные атрибуты (id, предыдущие значения)

:return: Словарь с параметрами объекта

"""

data = {

'value': self.\_val,

'dimension': self.dimension,

'name': self.name,

'description': self.description,

'status': self.status.name,

'store\_prev': self.store\_prev,

'min\_value': self.min\_value,

'max\_value': self.max\_value,

'is\_callable': self.is\_callable()

}

if include\_private:

data.update({

'id': str(self.id),

'previous\_value': self.previous\_value,

'previous\_status': self.previous\_status.name if self.previous\_status else None

})

return data

def convert(self, new\_class: ValueClass):

self.\_value\_spec = new\_class

def \_numeric\_operation(self, other: Union['Value', Any], op: Callable, op\_str: str) -> 'Value':

# Конвертация обычных чисел в безразмерный Value

if not isinstance(other, Value):

other = Value(

name="constant",

value\_spec=ValueClass(None, None, None),

value=other,

status=ValueStatus.FIXED

)

# Проверка на callable

if self.is\_callable() or other.is\_callable():

raise TypeError("Арифметические операции запрещены для вызываемых объектов")

# Проверка числового типа

if not isinstance(self.value, (int, float, complex, np.number)) or \

not isinstance(other.value, (int, float, complex, np.number)):

raise TypeError("Операции разрешены только для числовых типов")

# Определение спецификации результата

new\_spec = self.\_determine\_result\_spec(other, op\_str)

# Выполнение операции

try:

new\_value = op(self.value, other.value)

except Exception as e:

raise TypeError(f"Операция не поддерживается: {e}")

# Создание нового объекта Value

return Value(

name=f"({self.name}{op\_str}{other.name})",

value\_spec=new\_spec,

value=new\_value,

status=ValueStatus.CALCULATED

)

def \_determine\_result\_spec(self, other: 'Value', op\_str: str) -> ValueClass:

"""Определяет спецификацию результата операции"""

# Сложение и вычитание

if op\_str in ['+', '-']:

if self.\_value\_spec == other.\_value\_spec:

return copy.deepcopy(self.\_value\_spec)

if self.dimension == other.dimension:

return ValueClass(None, None, self.dimension)

return ValueClass(None, None, None)

# Умножение

if op\_str == '\*':

physics\_type = (self.physics\_type if self.physics\_type == other.physics\_type

else None)

new\_dim = combine\_dims(self.dimension, other.dimension, '\*')

return ValueClass(None, physics\_type, new\_dim)

# Деление

if op\_str == '/':

physics\_type = (self.physics\_type if self.physics\_type == other.physics\_type

else None)

new\_dim = combine\_dims(self.dimension, other.dimension, '/')

return ValueClass(None, physics\_type, new\_dim)

# Возведение в степень

if op\_str == '\*\*':

# Степень должна быть безразмерной

if other.dimension is None and other.physics\_type is None:

if self.dimension:

try:

# Красивое форматирование для целых степеней

power = int(other.value) if other.value.is\_integer() else other.value

new\_dim = f"({self.dimension})^{power}"

except:

new\_dim = f"({self.dimension})^{other.value}"

else:

new\_dim = None

return ValueClass(None, self.physics\_type, new\_dim)

return ValueClass(None, None, None)

return ValueClass(None, None, None)

# Основные арифметические операции

def \_\_add\_\_(self, other) -> 'Value':

return self.\_numeric\_operation(other, lambda a, b: a + b, '+')

def \_\_sub\_\_(self, other) -> 'Value':

return self.\_numeric\_operation(other, lambda a, b: a - b, '-')

def \_\_mul\_\_(self, other) -> 'Value':

return self.\_numeric\_operation(other, lambda a, b: a \* b, '\*')

def \_\_truediv\_\_(self, other) -> 'Value':

return self.\_numeric\_operation(other, lambda a, b: a / b, '/')

def \_\_pow\_\_(self, power) -> 'Value':

return self.\_numeric\_operation(power, lambda a, b: a \*\* b, '\*\*')

# Обратные операции

def \_\_radd\_\_(self, other) -> 'Value':

return self.\_\_add\_\_(other)

def \_\_rsub\_\_(self, other) -> 'Value':

return Value("constant", ValueClass(None, None, None), other, ValueStatus.FIXED) - self

def \_\_rmul\_\_(self, other) -> 'Value':

return self.\_\_mul\_\_(other)

def \_\_rtruediv\_\_(self, other) -> 'Value':

return Value("constant", ValueClass(None, None, None), other, ValueStatus.FIXED) / self

def \_\_rpow\_\_(self, other) -> 'Value':

return Value("constant", ValueClass(None, None, None), other, ValueStatus.FIXED) \*\* self

# Унарные операции

def \_\_neg\_\_(self) -> 'Value':

if self.is\_callable():

raise TypeError("Унарный минус запрещен для вызываемых объектов")

return Value(

name=f"-{self.name}",

value\_spec=copy.deepcopy(self.\_value\_spec),

value=-self.value,

status=ValueStatus.CALCULATED

)

def \_\_abs\_\_(self) -> 'Value':

if self.is\_callable():

raise TypeError("Модуль запрещен для вызываемых объектов")

return Value(

name=f"|{self.name}|",

value\_spec=copy.deepcopy(self.\_value\_spec),

value=abs(self.value),

status=ValueStatus.CALCULATED

)

Файл Port.py

from Value import Value, ValueStatus

import uuid

from typing import Dict, Iterator, List, Optional, Tuple, Union, Any, Collection, Type

from ObjectRepository import ObjectRepository

class Port:

# Защищенные атрибуты, которые нельзя перезаписать

PROTECTED\_ATTRS = ['\_name', '\_values', 'PROTECTED\_ATTRS']

def \_\_init\_\_(self, name: str, \*values: Value):

# Безопасная инициализация атрибутов

super().\_\_setattr\_\_('\_name', name)

super().\_\_setattr\_\_('\_values', ObjectRepository(rep\_type='value', postfix=name))

# Добавление начальных значений

for value in values:

self.add\_value(value)

def add\_value(self, value: Value, type\_check: Optional[Type] = None):

"""Добавление величины в порт с проверкой типа"""

# Проверка типа значения

if type\_check and not isinstance(value.value, type\_check):

raise TypeError(f"Ожидается тип {type\_check}, получен {type(value.value)}")

# Регистрация величины (используется value.name как base\_name)

self.\_values.register(value)

@property

def name(self) -> str:

return self.\_name

@name.setter

def name(self, new\_name: str):

self.\_name = new\_name

def \_\_iter\_\_(self) -> Iterator[Tuple[uuid.UUID, Value]]:

return iter(self.\_values.items())

def \_\_getitem\_\_(self, key: Union[uuid.UUID, str, int]) -> Value:

return self.\_values[key]

def \_\_getattr\_\_(self, name: str) -> Tuple[Any, ValueStatus]:

"""Доступ к величине по имени атрибута"""

# Защита системных атрибутов

if name in self.PROTECTED\_ATTRS:

return super().\_\_getattribute\_\_(name)

if name in self.\_values:

value = self.\_values[name]

return value.value, value.status

raise AttributeError(f"Порт '{self.name}' не содержит величины '{name}'")

def \_\_setattr\_\_(self, name: str, value: Tuple[Any, Union[ValueStatus, str]]):

"""Установка значения величины"""

# Защита системных атрибутов

if name in self.PROTECTED\_ATTRS:

super().\_\_setattr\_\_(name, value)

return

if name not in self.\_values:

raise AttributeError(f"Величина '{name}' не найдена в порте")

value\_obj = self.\_values[name]

new\_val, new\_status = value

# Преобразование строкового статуса

if isinstance(new\_status, str):

new\_status = ValueStatus.from\_input(new\_status)

value\_obj.update(new\_val, new\_status)

def get\_value(self, identifier: Union[str, uuid.UUID]) -> Optional[Value]:

return self.\_values.get\_by\_name(identifier) if isinstance(identifier, str) else self.\_values.get\_by\_id(

identifier)

def get\_value\_state(self, identifier: Union[str, uuid.UUID]) -> Optional[Tuple[Any, ValueStatus]]:

value = self.get\_value(identifier)

return (value.value, value.status) if value else None

def set\_value\_state(self, identifier: Union[str, uuid.UUID], value: Any, status: Union[ValueStatus, str]):

value\_obj = self.get\_value(identifier)

if not value\_obj:

raise AttributeError(f"Величина '{identifier}' не найдена")

value\_obj.update(value, ValueStatus.from\_input(status))

def \_\_contains\_\_(self, value: Union[str, uuid.UUID, Value]) -> bool:

return value in self.\_values

def \_\_len\_\_(self) -> int:

return len(self.\_values)

def \_\_repr\_\_(self) -> str:

return f"Port(name={self.name}, values={list(self.\_values.registered\_base\_names)})"

def list\_by\_status(self, status: Union[ValueStatus, str, Collection[Union[ValueStatus, str]]]) -> List[str]:

# Преобразование в множество статусов

statuses = {status} if not isinstance(status, Collection) or isinstance(status, str) else status

status\_set = {ValueStatus.from\_input(s) for s in statuses}

return [name for name in self.\_values.registered\_base\_names

if self.\_values[name].status in status\_set]

def list\_known(self) -> List[str]:

return self.list\_by\_status([s for s in ValueStatus if s != ValueStatus.UNKNOWN])

def list\_unknown(self) -> List[str]:

return self.list\_by\_status(ValueStatus.UNKNOWN)

@property

def is\_calculated(self) -> bool:

return all(value.status != ValueStatus.UNKNOWN for \_, value in self.\_values.items())

def reset(self, reset\_fixed: bool = False):

for \_, value in self.\_values.items():

if value.status in (ValueStatus.CALCULATED, ValueStatus.DEPEND) or \

(reset\_fixed and value.status == ValueStatus.FIXED):

value.update(None, ValueStatus.UNKNOWN)

def reset\_by\_names(self, names: List[str], reset\_fixed: bool = False):

for name in names:

if name in self.\_values:

value = self.\_values[name]

if value.status in (ValueStatus.CALCULATED, ValueStatus.DEPEND) or \

(reset\_fixed and value.status == ValueStatus.FIXED):

value.update(None, ValueStatus.UNKNOWN)

def \_\_eq\_\_(self, other: object) -> bool:

if not isinstance(other, Port):

return NotImplemented

return (self.name == other.name and

{v.name: v.dimension for \_, v in self.\_values.items()} ==

{v.name: v.dimension for \_, v in other.\_values.items()})

def \_\_ne\_\_(self, other: object) -> bool:

return not self.\_\_eq\_\_(other)

# Новые методы

def update\_bulk(self, updates: Dict[str, Tuple[Any, Union[ValueStatus, str]]]):

"""Массовое обновление значений"""

for name, (val, status) in updates.items():

self.set\_value\_state(name, val, status)

def get\_all(self) -> Dict[str, Tuple[Any, ValueStatus]]:

"""Получение всех значений порта"""

return {name: self.\_\_getattr\_\_(name) for name in self.\_values.registered\_base\_names}

def as\_dict(self) -> Dict[str, Any]:

"""Сериализация порта в словарь"""

return {

"name": self.name,

"values": [value.to\_dict() for \_, value in self.\_values.items()]

}

@classmethod

def from\_dict(cls, data: Dict[str, Any]) -> "Port":

"""Десериализация порта из словаря"""

port = cls(data["name"])

for value\_data in data["values"]:

port.add\_value(Value.from\_dict(value\_data))

return port

Файл ObjectRepository.py

from typing import (Dict, List, Optional, Callable, Any,

Union, Tuple, TypeVar, Generic, Iterator)

import uuid

T = TypeVar('T')

class ObjectRepository(Generic[T]):

def \_\_init\_\_(self, rep\_type: str, postfix: Optional[str] = None):

self.repository: Dict[uuid.UUID, T] = {}

self.\_base\_name\_to\_id: Dict[str, uuid.UUID] = {}

self.\_id\_to\_base\_name: Dict[uuid.UUID, str] = {}

self.\_obj\_to\_id: Dict[int, uuid.UUID] = {}

self.\_obj\_list: List[Tuple[uuid.UUID, T]] = []

if rep\_type.lower() in ['value', 'port', 'element']:

self.\_repository\_type = rep\_type.lower()

else:

raise ValueError('Некорректно задан тип хранилища')

self.\_postfix = postfix

def \_generate\_full\_name(self, base\_name: str) -> str:

if self.\_postfix is not None:

return f'{base\_name}\_{self.\_postfix}'

else:

return base\_name

def \_extract\_base\_name(self, name: str) -> str:

if self.\_postfix and name.endswith(f"\_{self.\_postfix}"):

return name[:-(len(self.\_postfix) + 1)]

return name

def \_validate\_object\_type(self, obj: T) -> bool:

obj\_type = type(obj).\_\_name\_\_.lower()

return obj\_type == self.\_repository\_type

def \_validate\_base\_name(self, base\_name: str) -> None:

if '\_' in base\_name:

raise ValueError(f"Базовое имя '{base\_name}' содержит подчеркивание, что недопустимо")

if base\_name in self.\_base\_name\_to\_id:

raise ValueError(f"Базовое имя '{base\_name}' уже зарегистрировано")

def register(self, obj: T, obj\_id: Optional[uuid.UUID] = None):

if not self.\_validate\_object\_type(obj):

raise TypeError(f"Объект типа {type(obj).\_\_name\_\_} не поддерживается")

self.\_validate\_base\_name(obj.name)

if obj\_id is None:

obj\_id = uuid.uuid4()

elif obj\_id in self.repository:

raise ValueError(f"UUID {obj\_id} уже используется")

self.repository[obj\_id] = obj

self.\_base\_name\_to\_id[obj.name] = obj\_id

self.\_id\_to\_base\_name[obj\_id] = obj.name

self.\_obj\_to\_id[id(obj)] = obj\_id

self.\_obj\_list.append((obj\_id, obj))

def get\_by\_id(self, obj\_id: uuid.UUID) -> Optional[T]:

"""Возвращает объект по UUID"""

return self.repository.get(obj\_id)

def get\_by\_name(self, name: str) -> Optional[T]:

"""Возвращает объект по имени (полному или базовому)"""

base\_name = self.\_extract\_base\_name(name)

obj\_id = self.\_base\_name\_to\_id.get(base\_name)

return self.repository.get(obj\_id) if obj\_id else None

def get\_by\_object(self, obj: T) -> Optional[T]:

"""Возвращает зарегистрированную версию объекта"""

obj\_id = self.\_obj\_to\_id.get(id(obj))

return self.repository.get(obj\_id) if obj\_id else None

def get(self, identifier: Union[uuid.UUID, str]):

return self[identifier]

@property

def repository\_type(self) -> str:

return self.\_repository\_type

@property

def postfix(self) -> Optional[str]:

return self.\_postfix

@property

def size(self) -> int:

return len(self.repository)

@property

def registered\_ids(self) -> List[uuid.UUID]:

return list(self.repository.keys())

@property

def registered\_base\_names(self) -> List[str]:

return list(self.\_base\_name\_to\_id.keys())

@property

def registered\_full\_names(self) -> List[str]:

return [self.get\_full\_name(name) for name in self.registered\_base\_names]

def is\_postfix(self, postfix: str):

return postfix == self.\_postfix

def remove(self, identifier: Union[uuid.UUID, str]) -> None:

"""Удаляет объект по UUID или имени"""

if isinstance(identifier, uuid.UUID):

obj\_id = identifier

base\_name = self.\_id\_to\_base\_name.get(obj\_id)

else:

base\_name = self.\_extract\_base\_name(identifier)

obj\_id = self.\_base\_name\_to\_id.get(base\_name)

if not obj\_id or not base\_name:

return

# Удаление из всех индексов

obj = self.repository.pop(obj\_id)

self.\_base\_name\_to\_id.pop(base\_name)

self.\_id\_to\_base\_name.pop(obj\_id)

self.\_obj\_to\_id.pop(id(obj))

def \_\_contains\_\_(self, identifier: Union[uuid.UUID, str, T]) -> bool:

if isinstance(identifier, uuid.UUID):

return identifier in self.repository

if isinstance(identifier, str):

base\_name = self.\_extract\_base\_name(identifier)

return base\_name in self.\_base\_name\_to\_id

return id(identifier) in self.\_obj\_to\_id

def \_\_getitem\_\_(self, identifier: Union[uuid.UUID, str, int]) -> T:

if isinstance(identifier, uuid.UUID):

obj = self.get\_by\_id(identifier)

elif isinstance(identifier, str):

obj = self.get\_by\_name(identifier)

else:

try:

obj = self.\_obj\_list[identifier]

except IndexError:

obj = None

if obj is None:

raise KeyError(f"Объект не найден: {identifier}")

return obj

def \_\_iter\_\_(self) -> Iterator[Tuple[uuid.UUID, T]]:

return iter(self.repository.items())

def \_\_len\_\_(self) -> int:

return self.size

def values(self) -> List[T]:

return list(self.repository.values())

def items(self) -> List[Tuple[uuid.UUID, T]]:

return list(self.repository.items())

def find\_by\_value(self, value: Any) -> List[T]:

"""Находит объекты с указанным значением"""

return [obj for obj in self.repository.values() if getattr(obj, 'value', None) == value]

Файл Element.py

class Element:

PROTECTED\_ATTRS = [

'\_name', '\_description',

'\_in\_ports', '\_out\_ports', '\_parameters',

'\_calculate\_func', '\_update\_int\_conn\_func', '\_setup\_func',

'PROTECTED\_ATTRS'

]

def \_\_init\_\_(self,

name: str,

description: str,

in\_ports: List[Union[dict, Port]],

out\_ports: List[Union[dict, Port]],

parameters: List[Union[dict, Value]],

calculate\_func: Optional[Callable[['Element'], None]] = None,

update\_int\_conn\_func: Optional[Callable[['Element'], None]] = None,

setup\_func: Optional[Callable[['Element'], None]] = None):

super().\_\_setattr\_\_('\_name', name)

super().\_\_setattr\_\_('\_description', description)

super().\_\_setattr\_\_('\_in\_ports', ObjectRepository(rep\_type='port', postfix=name))

super().\_\_setattr\_\_('\_out\_ports', ObjectRepository(rep\_type='port', postfix=name))

super().\_\_setattr\_\_('\_parameters', ObjectRepository(rep\_type='value', postfix=name))

for port in in\_ports:

self.\_add\_port(port, is\_input=True)

for port in out\_ports:

self.\_add\_port(port, is\_input=False)

for param in parameters:

self.\_add\_parameter(param)

self.\_validate\_and\_set\_func('\_calculate\_func', calculate\_func)

self.\_validate\_and\_set\_func('\_update\_int\_conn\_func', update\_int\_conn\_func)

self.\_validate\_and\_set\_func('\_setup\_func', setup\_func)

if self.\_setup\_func:

self.\_setup\_func(self)

def \_add\_port(self, port\_data: Union[dict, Port], is\_input: bool):

port = port\_data if isinstance(port\_data, Port) else Port.from\_dict(port\_data)

repo = self.\_in\_ports if is\_input else self.\_out\_ports

repo.register(port)

def \_add\_parameter(self, param\_data: Union[dict, Value]):

param = param\_data if isinstance(param\_data, Value) else Value.from\_dict(param\_data)

self.\_parameters.register(param)

def \_validate\_and\_set\_func(self, attr\_name: str, func: Optional[Callable]):

if func is not None and not callable(func):

raise TypeError(f"{attr\_name[1:]} must be callable or None")

super().\_\_setattr\_\_(attr\_name, func)

# ---------------------------

# Универсальный поиск Value

# ---------------------------

def \_resolve\_target(self, attr\_name: str) -> Optional[Value]:

# 1. Параметры элемента

if attr\_name in self.\_parameters:

return self.\_parameters[attr\_name]

# 2. value\_name\_portIndex

m\_index = re.match(r"^(.+)\_(\d)\_(\d+)$", attr\_name)

if m\_index:

v\_name, port\_type, port\_index = m\_index[1], int(m\_index[2]), int(m\_index[3])

repo = self.\_in\_ports if port\_type == 0 else self.\_out\_ports

if 0 <= port\_index < len(repo):

port = repo[port\_index]

return port.get\_value(v\_name)

# 3. value\_name\_portName

m\_name = re.match(r"^(.+)\_([A-Za-z0-9]+)$", attr\_name)

if m\_name:

v\_name, port\_name = m\_name[1], m\_name[2]

port = self.\_in\_ports.get\_by\_name(port\_name) or self.\_out\_ports.get\_by\_name(port\_name)

if port:

return port.get\_value(v\_name)

return None

# ---------------------------

# Доступ к данным

# ---------------------------

def \_\_getattr\_\_(self, name: str) -> Any:

if name in self.PROTECTED\_ATTRS:

return super().\_\_getattribute\_\_(name)

value\_obj = self.\_resolve\_target(name)

if value\_obj:

return (value\_obj.value, value\_obj.status)

raise AttributeError(f"{self.\_\_class\_\_.\_\_name\_\_} has no attribute '{name}'")

def \_\_setattr\_\_(self, name: str, value: Any):

if name in self.PROTECTED\_ATTRS:

super().\_\_setattr\_\_(name, value)

return

value\_obj = self.\_resolve\_target(name)

if value\_obj:

self.\_set\_value(value\_obj, value)

return

super().\_\_setattr\_\_(name, value)

def \_set\_value(self, value\_obj: Value, value: Any):

if isinstance(value, tuple) and len(value) == 2:

v, status = value

if isinstance(status, str):

status = ValueStatus.from\_input(status)

value\_obj.update(v, status)

else:

value\_obj.update(value)

# ---------------------------

# API

# ---------------------------

@property

def name(self) -> str:

return self.\_name

@property

def description(self) -> str:

return self.\_description

@property

def in\_ports(self) -> ObjectRepository:

return self.\_in\_ports

@property

def out\_ports(self) -> ObjectRepository:

return self.\_out\_ports

@property

def parameters(self) -> ObjectRepository:

return self.\_parameters

def get\_port(self, identifier: Union[str, uuid.UUID, int]) -> Port:

if isinstance(identifier, uuid.UUID):

return self.\_in\_ports.get\_by\_id(identifier) or self.\_out\_ports.get\_by\_id(identifier)

if isinstance(identifier, str):

return self.\_in\_ports.get\_by\_name(identifier) or self.\_out\_ports.get\_by\_name(identifier)

if isinstance(identifier, int):

if identifier < len(self.\_in\_ports):

return self.\_in\_ports[identifier]

return self.\_out\_ports[identifier - len(self.\_in\_ports)]

raise KeyError(f"Port not found: {identifier}")

def calculate(self):

if self.\_calculate\_func:

self.\_calculate\_func(self)

else:

raise NotImplementedError("Calculate function not implemented")

def update\_internal\_connections(self):

if self.\_update\_int\_conn\_func:

self.\_update\_int\_conn\_func(self)

else:

raise NotImplementedError("Update internal connections function not implemented")

# ---------------------------

# Сериализация

# ---------------------------

def as\_dict(self) -> Dict[str, Any]:

return {

"name": self.name,

"description": self.description,

"in\_ports": [p.as\_dict() for \_, p in self.\_in\_ports.items()],

"out\_ports": [p.as\_dict() for \_, p in self.\_out\_ports.items()],

"parameters": [p.to\_dict() for \_, p in self.\_parameters.items()]

}

@classmethod

def from\_dict(cls, data: Dict[str, Any]) -> "Element":

return cls(

name=data["name"],

description=data["description"],

in\_ports=data["in\_ports"],

out\_ports=data["out\_ports"],

parameters=data["parameters"]

)

def \_\_repr\_\_(self) -> str:

return f"Element({self.name}, in={len(self.\_in\_ports)}, out={len(self.\_out\_ports)}, params={len(self.\_parameters)})"

def \_\_getitem\_\_(self, key: Union[Tuple[int, int], int, str, uuid.UUID]) -> Port:

"""

Варианты использования:

- elem[0, 1] → входной порт (0=in|1=out, index)

- elem[4] → общий индекс по всем портам

- elem['portname'] → поиск по имени

- elem[uuid] → поиск по ID

"""

if isinstance(key, tuple) and len(key) == 2:

port\_type, port\_index = key

if port\_type == 0:

return self.\_in\_ports[port\_index]

elif port\_type == 1:

return self.\_out\_ports[port\_index]

else:

raise IndexError("Неверный тип порта (0=in, 1=out)")

if isinstance(key, int):

if key < len(self.\_in\_ports):

return self.\_in\_ports[key]

key -= len(self.\_in\_ports)

if key < len(self.\_out\_ports):

return self.\_out\_ports[key]

raise IndexError("Порт с таким индексом не найден")

if isinstance(key, str):

port = self.\_in\_ports.get\_by\_name(key)

if port:

return port

port = self.\_out\_ports.get\_by\_name(key)

if port:

return port

raise KeyError(f"Порт '{key}' не найден")

if isinstance(key, uuid.UUID):

port = self.\_in\_ports.get\_by\_id(key)

if port:

return port

port = self.\_out\_ports.get\_by\_id(key)

if port:

return port

raise KeyError(f"Порт с ID {key} не найден")

raise TypeError(f"Неверный тип аргумента для \_\_getitem\_\_: {type(key).\_\_name\_\_}")

# ------------------- Методы доступа через ID -------------------

def get\_port\_by\_id(self, port\_id: uuid.UUID) -> Port:

return self.\_in\_ports.get\_by\_id(port\_id) or self.\_out\_ports.get\_by\_id(port\_id)

def get\_parameter\_by\_id(self, param\_id: uuid.UUID) -> Optional[Value]:

return self.\_parameters.get\_by\_id(param\_id)

def get\_value\_from\_port\_by\_id(self, port\_id: uuid.UUID, value\_name: str) -> Optional[Value]:

port = self.get\_port\_by\_id(port\_id)

if not port:

return None

return port.get\_value(value\_name)

def get\_all\_port\_ids(self) -> list:

return self.\_in\_ports.registered\_ids + self.\_out\_ports.registered\_ids

def get\_all\_parameter\_ids(self) -> list:

return self.\_parameters.registered\_ids

def get\_all\_value\_ids\_in\_ports(self) -> list:

"""Возвращает список (port\_id, value\_name) для всех значений портов"""

result = []

for port\_id, port in self.\_in\_ports.items():

for val\_name in port.\_values.registered\_base\_names:

result.append((port\_id, val\_name))

for port\_id, port in self.\_out\_ports.items():

for val\_name in port.\_values.registered\_base\_names:

result.append((port\_id, val\_name))

return result

Файл ElementFactory.py

import os

import glob

import json

import importlib

from typing import Optional, Callable, Dict, Any, List

from backend.src.Core.Element import Element

from backend.src.Core.Value import ValueClass, Value, ValueStatus

from backend.src.Core.Port import Port

class ElementFactory:

def \_\_init\_\_(self, value\_classes\_path: str, ports\_path: str, elements\_dir: str):

# Счётчики для уникальных имён

self.\_element\_counters: Dict[str, int] = {}

# Загружаем ValueClass

self.value\_classes\_json = self.\_load\_json(value\_classes\_path)

self.value\_classes\_cache = self.\_load\_value\_classes(self.value\_classes\_json)

# Загружаем шаблонные порты

self.ports\_def = self.\_load\_json(ports\_path)

# Загружаем элементы (каждый из отдельного JSON)

self.element\_defs = self.\_load\_all\_elements(elements\_dir)

# Запоминаем метаданные отдельно

self.metadata\_cache = {

name: {k: v for k, v in cfg.items() if k in ["name", "author", "version", "description", "category"]}

for name, cfg in self.element\_defs.items()

}

def \_load\_json(self, path: str):

with open(path, "r", encoding="utf-8") as f:

return json.load(f)

def \_load\_all\_elements(self, directory: str):

elements = {}

for file\_path in glob.glob(os.path.join(directory, "\*\*/\*.json"), recursive=True):

name = os.path.splitext(os.path.basename(file\_path))[0]

with open(file\_path, "r", encoding="utf-8") as f:

elements[name] = json.load(f)

return elements

def \_load\_value\_classes(self, data: dict):

"""Собираем ValueClass с уникальным ключом PhysicsType.ShortName."""

cache = {}

for group\_name, group\_data in data.items():

physics\_type = group\_data.get("physics\_type", group\_name)

for short\_name, vinfo in group\_data["values"].items():

key = f"{physics\_type}.{short\_name}"

cache[key] = ValueClass(

value\_name=vinfo["value\_name"],

physics\_type=physics\_type,

dimension=vinfo["dimension"]

)

return cache

def \_import\_func(self, full\_path: Optional[str]) -> Optional[Callable]:

if not full\_path:

return None

module\_path, func\_name = full\_path.rsplit(".", 1)

mod = importlib.import\_module(module\_path)

return getattr(mod, func\_name)

def \_build\_value(self, spec: Dict[str, Any]) -> Value:

key = spec["value\_class"]

if key not in self.value\_classes\_cache:

raise ValueError(f"ValueClass '{key}' не найден")

vc = self.value\_classes\_cache[key]

return Value(

name=spec["param\_name"],

value\_spec=vc,

value=spec.get("value"),

status=ValueStatus.from\_input(spec.get("status", "unknown")),

min\_value=spec.get("min\_value"),

max\_value=spec.get("max\_value")

)

def \_build\_port(self, port\_spec: Dict[str, Any]) -> Port:

values = []

if "use\_port" in port\_spec:

if port\_spec["use\_port"] not in self.ports\_def:

raise ValueError(f"Шаблон порта '{port\_spec['use\_port']}' не найден")

for v in self.ports\_def[port\_spec["use\_port"]]["values"]:

values.append(self.\_build\_value(v))

elif "values" in port\_spec:

for v in port\_spec["values"]:

values.append(self.\_build\_value(v))

else:

raise ValueError("Порт должен содержать либо 'use\_port', либо 'values'")

return Port(port\_spec["name"], \*values)

def \_build\_ports(self, ports\_def: Any) -> List[Dict[str, Any]]:

# Если число — создаст конструктор наследника

if isinstance(ports\_def, int):

return ports\_def

return [self.\_build\_port(p).as\_dict() for p in ports\_def]

def \_build\_parameters(self, params\_def: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

return [self.\_build\_value(p).to\_dict() for p in params\_def]

def create\_element(self, element\_name: str) -> Element:

"""Создаёт экземпляр элемента по имени из конфигурации"""

if element\_name not in self.element\_defs:

raise ValueError(f"Элемент '{element\_name}' не зарегистрирован")

cfg = self.element\_defs[element\_name]

element\_cls = Element

# Делаем уникальное имя

self.\_element\_counters[element\_name] = self.\_element\_counters.get(element\_name, 0) + 1

unique\_name = f"{cfg['name']} #{self.\_element\_counters[element\_name]}"

# Наследник от Element

if cfg.get("class\_path"):

module\_path, cls\_name = cfg["class\_path"].rsplit(".", 1)

mod = importlib.import\_module(module\_path)

element\_cls = getattr(mod, cls\_name)

in\_ports = cfg.get("in\_ports", 0)

out\_ports = cfg.get("out\_ports", 0)

parameters = self.\_build\_parameters(cfg.get("parameters", []))

else:

in\_ports = self.\_build\_ports(cfg.get("in\_ports", []))

out\_ports = self.\_build\_ports(cfg.get("out\_ports", []))

parameters = self.\_build\_parameters(cfg.get("parameters", []))

calc\_func = self.\_import\_func(cfg.get("functions", {}).get("calculate\_func"))

update\_func = self.\_import\_func(cfg.get("functions", {}).get("update\_int\_conn\_func"))

setup\_func = self.\_import\_func(cfg.get("functions", {}).get("setup\_func"))

return element\_cls(

name=unique\_name,

description=cfg.get("description", ""),

in\_ports=in\_ports,

out\_ports=out\_ports,

parameters=parameters,

calculate\_func=calc\_func,

update\_int\_conn\_func=update\_func,

setup\_func=setup\_func

)

# -------------------

# Информационные методы

# -------------------

def list\_elements(self) -> List[str]:

return list(self.element\_defs.keys())

def list\_ports(self) -> List[str]:

return list(self.ports\_def.keys())

def list\_values(self) -> List[str]:

return list(self.value\_classes\_cache.keys())

def get\_metadata(self, element\_name: str) -> Dict[str, Any]:

return self.metadata\_cache.get(element\_name, {})

def summary(self) -> str:

lines = []

lines.append("=== Value Classes ===")

for vc in self.value\_classes\_cache:

lines.append(f" {vc}")

lines.append("\n=== Ports ===")

for p in self.ports\_def:

lines.append(f" {p}")

lines.append("\n=== Elements ===")

for e in self.element\_defs:

meta = self.metadata\_cache[e]

lines.append(f" {meta['name']} (v{meta['version']} by {meta['author']})")

return "\n".join(lines)

Файл конфигурации физических величин value\_classes.json

{

"Thermodynamics": {

"physics\_type": "Thermodynamics",

"values": {

"G": {"value\_name": "Mass flow", "dimension": "kg/s" },

"p": {"value\_name": "Pressure", "dimension": "Pa" },

"h": {"value\_name": "Enthalpy", "dimension": "J/kg" },

"medium": {"value\_name": "Medium", "dimension": null }

}

},

"General": {

"physics\_type": "General",

"values": {

"eta": {"value\_name": "Efficiency", "dimension": null},

"general": {"value\_name": "General value", "dimension": null}

}

},

"Mechanics":{

"physics\_type": "Mechanics",

"values": {

"p": {"value\_name": "Pressure", "dimension": "Pa"},

"N": {"value\_name": "Power", "dimension": "W"}

}

}

}

Файл конфигурации портов ports.json:

{

"thermodinamyc\_port": {

"description": "Используется для передачи термодинамических параметров, содержит в сете расход, давление, энтальпию и модель определения термодинамических параметров",

"values": [

{ "param\_name": "G", "value\_class": "Thermodynamics.G", "min\_value": 0 },

{ "param\_name": "p", "value\_class": "Thermodynamics.p", "min\_value": 0, "max\_value": 1e7 },

{ "param\_name": "h", "value\_class": "Thermodynamics.h" },

{ "param\_name": "medium", "value\_class": "Thermodynamics.medium" }

]

}

}

пример файла конфигурации элемента (dummy.json):

{

"name": "Dummy",

"author": "Developer 1",

"version": "1.0",

"description": "This is dummy element",

"category": "General",

"class\_path": null,

"in\_ports": [

{"name": "inlet1", "use\_port": "thermodinamyc\_port"},

{"name": "inlet2",

"description": "Используется для передачи части термодинамических параметров",

"values": [

{ "param\_name": "G", "value\_class": "Thermodynamics.G", "min\_value": 0 },

{ "param\_name": "p", "value\_class": "Thermodynamics.p", "min\_value": 0, "max\_value": 1e7 }]

}],

"out\_ports": [

{"name": "outlet1", "use\_port": "thermodinamyc\_port"},

{"name": "outlet2",

"description": "Данные о характеристиках работы",

"values": [

{ "param\_name": "G", "value\_class": "Thermodynamics.G", "min\_value": 0 },

{ "param\_name": "p", "value\_class": "General.general", "min\_value": 0, "max\_value": 10}]

}],

"parameters": [

{

"name": "efficiency",

"value\_class": "General.eta",

"value": 0.85,

"min\_value": 0.5,

"max\_value": 1.0,

"status": "fixed"

},

{

"name": "val2",

"value\_class": "General.general",

"status": "unknown"

}

],

"functions": {

"calculate\_func": null,

"update\_int\_conn\_func": null,

"setup\_func": null

}

}

Пример файла конфигурации элемента, который использует класс наследник от базового Element:

{

"name": "Centrifugal pump",

"author": "Developer 1",

"version": "1.0",

"description": "Centrifugal pump",

"category": "General",

"class\_path": "elements.thermodinamic",

"in\_ports": [1],

"out\_ports": [1],

"parameters": [{"param\_name": "N", "value\_class": "Mechanics.N", "min\_value": 0}],

"functions": {

"calculate\_func": "funcs.pump.calculate",

"update\_int\_conn\_func": "funcs.thermal.update\_ports",

"setup\_func": null

}

}

Структура проекта

