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 (a )
        :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