Software Architecture & Design

ETH-Brownie

จัดทำโดย

นายณัฐพนธ์ สุขถาวร	62010278
นายณัฐภูมิ เพ็ชรชนะ	62010284
นายณัฐวุฒิ ครองอารีธรรม	62010293
นายนนทกร จิตรชิรานันท์	62010452
นายนิธิ น้อมประวัติ	62010497
นายพักตร์ภูมิ ตาแพร่	62010609

เสนอ

อ.ปริญญา เอกปริญญา

คณะวิศวกรรมศาสตร์ สาขาวิศวกรรมคอมพิวเตอร์ ปีการศึกษา 2564

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

คำนำ

รายงานฉบับนี้เป็นส่วนหนึ่งของรายวิชา Software Architecture & Design รหัสวิชา 01076024 คณะวิศวกรรมศาสตร์ สาขาวิชาวิศวกรรมคอมพิวเตอร์ ปีการศึกษา 2564 ซึ่งมีจุดประสงค์เพื่อศึกษาการออกแบบ Software และ Architecture ที่นำมาใช้พัฒนา Open Source โดยทางคณะผู้จัดทำได้เลือกศึกษาโปรแกรม ETH Brownie ซึ่งเป็นหนึ่งใน Open Source ที่ทำหน้าที่ช่วยในการพัฒนา Smart Contract

ทางคณะผู้จัดทำหวังว่ารายงานเล่มนี้จะมีประโยชน์ต่อผู้ที่ต้องการศึกษาเกี่ยวกับการออกแบบ Software Architecture หากมีข้อผิดพลาดประการใดคณะผู้จัดทำต้องขออภัยมา ณ ที่ นี้ด้วย

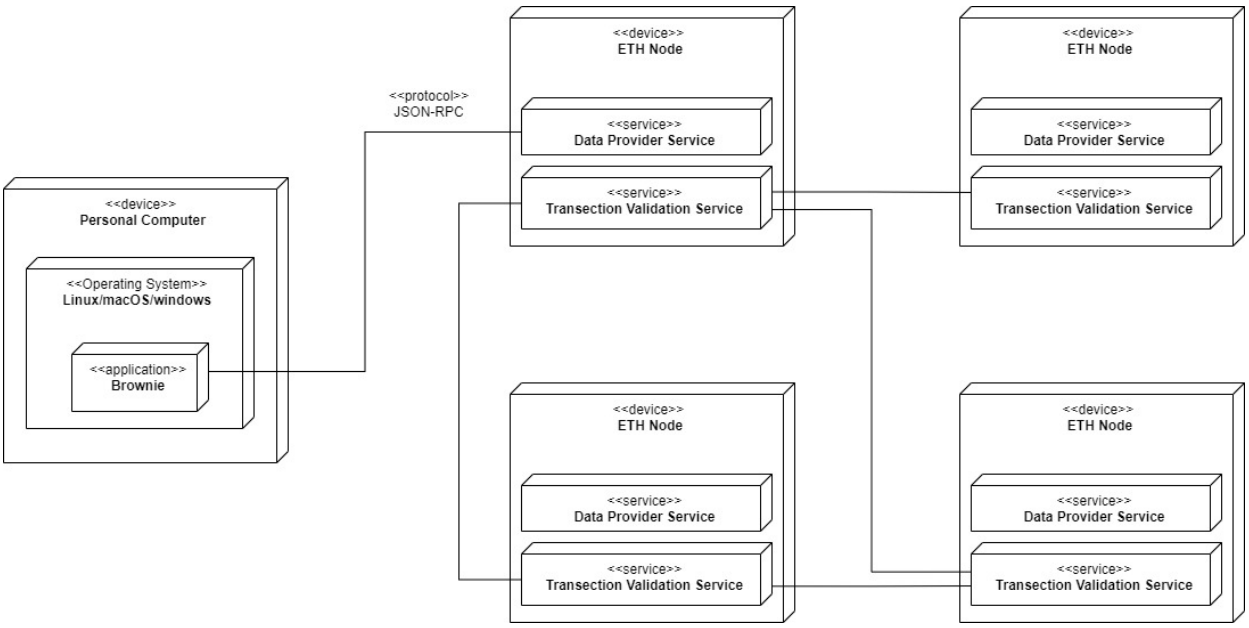
คณะผู้จัดทำ

Architectural Styles

จากการศึกษา ETH Brownie คณะผู้จัดทำได้พบ **Architectural Styles** ดังนี้

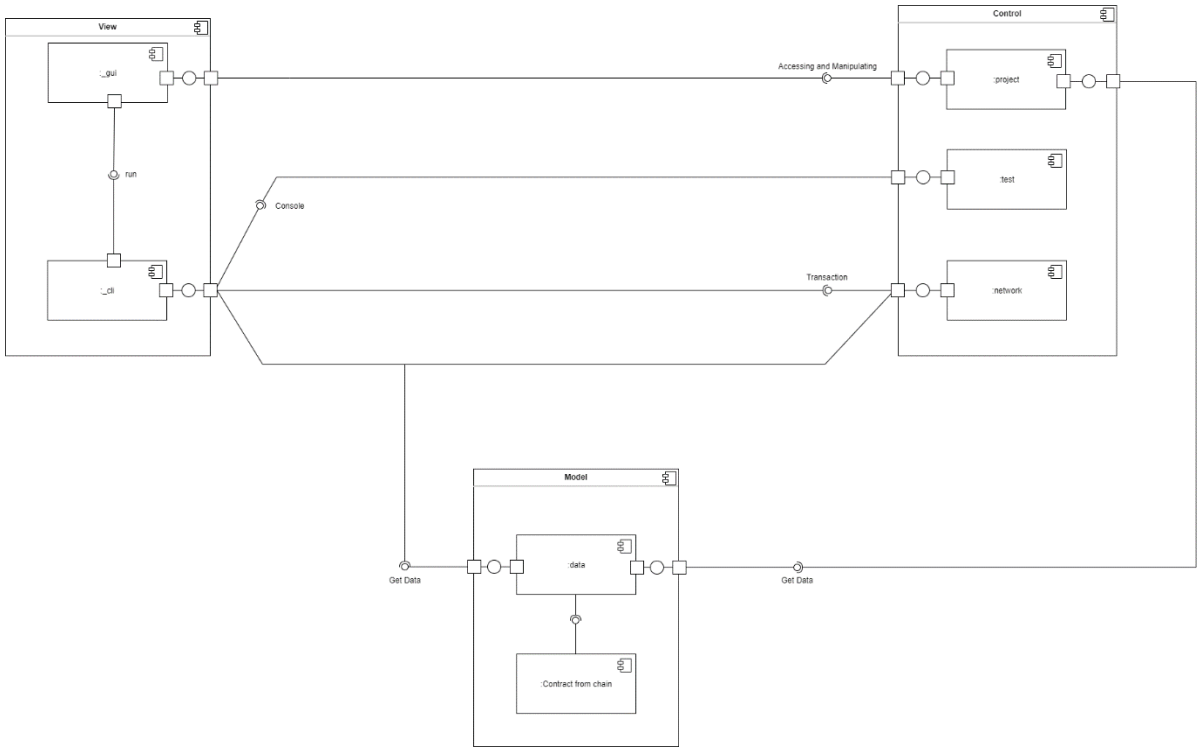
- **Client-Server**

  เนื่องจาก ETH Brownie มีการติดต่อระหว่างเครื่องผู้ใช้งาน กับ Ethereum Blockchain จึงสามารถตีความได้ว่า ETH Brownie มีการใช้งาน Architectural Styles แบบ Client – Server โดยเปรียบ ETH Brownie เป็น Client และเทียบ Node ใด Node หนึ่งที่เชื่อมใน Blockchain ของ Ethereum เป็น Server โดยมีการเชื่อมระหว่าง Client – Server ผ่าน JSON RPC



- **MVC**

ถึงจะไม่ใช่ประเด็นหลักแต่เนื่องจาก ETH Brownie มีทั้ง UI, Command Line ให้ใช้ และเมื่อตรวจสอบ Code ที่ Library มีจะพบว่ามีการแบ่งส่วนออกเป็นส่วน ๆ เราจึงตีความว่า MVC มีความคล้ายคลึงกันกับ MVC โดยส่วน _cli, _gui นับเป็นส่วน View ส่วนของการประมวลผลอื่น ๆ นับเป็น Controller และส่วน Smart Contracts รวมไปถึงข้อมูลอื่น ๆ จาก chain นับเป็น Model



ข้อดี

เนื่องจาก eth-brownie มีการใช้งานทั้ง client-server และ MVC ซึ่งล้วนแล้วแต่เพิ่มความ loose coupling ให้ eth-brownie เป็นเหตุผลให้ brownie มีคุณสมบัติของความ extensible, robustness, maintainability และความ composability สูง

ข้อเสีย

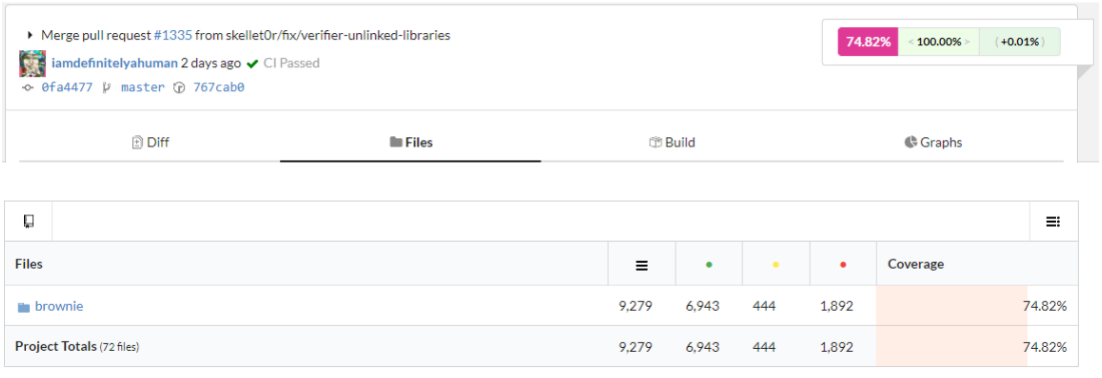
เนื่องจาก eth-brownie มีความ composable สูง เนื่องจากธรรมชาติของความเป็น library ทำให้ module แต่ละ module มีขนาดเล็ก แต่มีข้อเสียตรงที่ถ้าหากจะใช้งาน eth-brownie กับอะไรต้องนำ module เหล่านั้นมาประกอบ (เช่นต้องมีการตั้งค่า network, private key, ...) ซึ่งทำให้ code ที่ใช้งานจะมี boilerplate จำนวนมาก ซึ่งอาจจะแก้ไขโดยการเพิ่ม dependency injection เข้ามาช่วย inject config แทน

Quality Attributes

จากการศึกษา ETH Brownie คณะผู้จัดทำได้พบ Quality Attributes ดังนี้

- **Interoperability**: เนื่องจากธรรมชาติของการเป็น Library ตัว eth-brownie จึงจำเป็นที่จะต้องมี interoperability สูง ซึ่ง eth-brownie ตอบโจทย์โดยมีความสามารถใช้งานผ่านหลายช่องทางเช่น การใช้งานผ่าน ui และการใช้งานผ่าน command line รวมไปถึงยังรองรับการติดตั้งด้วย pip ซึ่งหมายความว่าหากใช้งาน python ก็สามารถที่จะติดตั้งผ่าน pip ได้เลย (มีความ Interoperability ที่รองรับมาตรฐาน pip ด้วย)
- **Availability**: eth-brownie อาจจำเป็นต้องใช้งานในระบบที่เกี่ยวข้องกับการเงิน ดังนั้นจึงจำเป็นต้องมี availability สูง ซึ่ง eth-brownie ทำให้มี availability สูงโดยการที่สามารถเลือก RPC ได้หลายโหนด ไม่จำเป็นต้องติดอยู่กับโหนดเดียว
- **Modifiability**: ด้วยความที่ Blockchain เป็นเทคโนโลยีใหม่ และมีการเปลี่ยนแปลงเรื่อย ๆ โปรเจคที่ทำกับ blockchain จึงต้องสามารถ reflect ความเปลี่ยนแปลงได้ โดยตัวอย่างของความ Modifiability ใน eth-brownie คือมีการใช้ Strategy Design Pattern ซึ่งทำให้ผู้ใช้สามารถปรับเปลี่ยนการใช้งาน ETH Brownie ได้ตามต้องการเช่น การนำมาใช้คำนวณค่าแก๊สบน ETH Chain
- **Testability**: eth-brownie อาจจำเป็นต้องใช้งานในระบบที่เกี่ยวข้องกับการเงิน ดังนั้นจึงจำเป็นต้องมี testability สูง และควรจะนำไปทดสอบได้ง่าย โดย eth-brownie มีการแบ่งสัดส่วน code ให้มีความ Loose Coupling ทำให้สามารถ test ได้ง่าย โดยดูได้จากผลการทดสอบ coverage สูงถึง 74.82%

Brownie have a high coverage score, indicating their ease of testing.



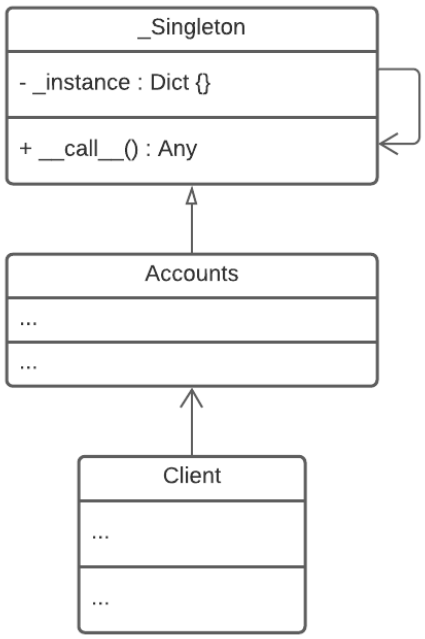
Design pattern

จากการศึกษา ETH Brownie คณะผู้จัดทำได้พบ Design pattern ดังนี้

- **Singleton:** จาก Source Code มีการใช้งาน Singleton เพื่อไม่ให้มี instance ของ object บางอย่างมากกว่าหนึ่ง โดยมีการใช้งานใน Config, TxHistory, Accounts และอื่น ๆ

  วิธีการสร้าง Singleton ที่มีการสร้างคือมีคลาส _Singleton ขึ้นมา และคลาสที่จะใช้ _Singleton ก็ inherit ไปโดยผ่าน metaclass

  Reference: brownie/_singleton.py, brownie/network/account.py



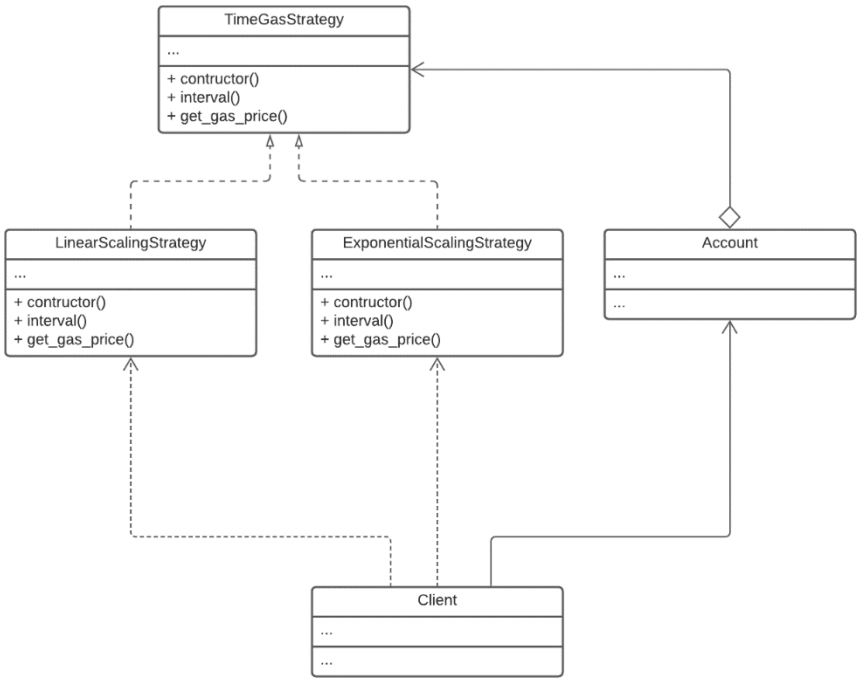

```python
#!/usr/bin/python3
from typing import Any, Dict, Tuple


class _Singleton(type):

    _instances: Dict = {}

    def __call__(cls, *args: Tuple, **kwargs: Dict) -> Any:
        if cls not in cls._instances:
            cls._instances[cls] = super(_Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

```python
class Accounts(metaclass=_Singleton):
    """
    List-like container that holds all available `Account` objects.
    Attributes
    ----------
    default : Account, optional
        Default account to broadcast transactions from.
    """

    def __init__(self) -> None:
        self.default = None
        self._accounts: List = []

        # prevent sensitive info from being stored in readline history
        self.add.__dict__["_private"] = True
        self.from_mnemonic.__dict__["_private"] = True

        _revert_register(self)
        self._reset()
```

- **Strategy:** จะเห็นได้ว่า ETH Brownie มีการใช้ strategy เพื่อให้ง่ายต่อการจัดการอัลกอริธึมหลายชุด โดยไม่ไปกระทบต่อ class อื่นในระบบ

  จาก Source Code เราพบว่า eth-brownie ได้มีการนำ strategy เข้ามาในส่วนของอัลกอริทึมสำหรับการประมาณราคา gas ซึ่งจะถูกใช้ซึ่งมีวิธีการประมาณที่แตกต่างกัน

  Reference: **brownie/network/gas/bases.py, brownie/network/gas/strategies.py**



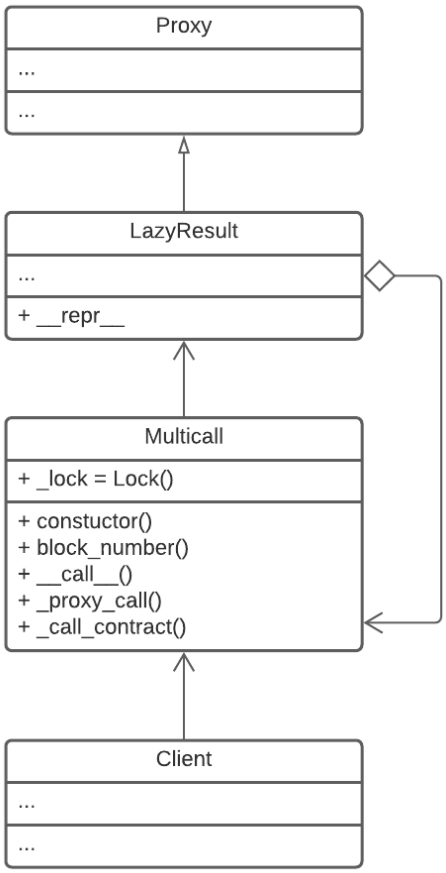

```python
class LinearScalingStrategy(TimeGasStrategy):
    """
    Gas strategy for linear gas price increase.
    Arguments
    ---------
    initial_gas_price : int
        The initial gas price to use in the first transaction
    max_gas_price : int
        The maximum gas price to use
    increment : float
        Multiplier applied to the previous gas price in order to determine
the new gas price
    time_duration : int
        Number of seconds between transactions
    """

    def __init__(
        self,
        initial_gas_price: Wei,
        max_gas_price: Wei,
        increment: float = 1.125,
        time_duration: int = 30,
    ):
        super().__init__(time_duration)
        self.initial_gas_price = Wei(initial_gas_price)
        self.max_gas_price = Wei(max_gas_price)
        self.increment = increment

    def get_gas_price(self) -> Generator[Wei, None, None]:
        last_gas_price = self.initial_gas_price
        yield last_gas_price

        while True:
            last_gas_price = min(Wei(last_gas_price * self.increment),
self.max_gas_price)
            yield last_gas_price
```

```python
class TimeGasStrategy(ScalingGasABC):
    """
    Abstract base class for time gas strategies.
    Time gas strategies are called every `time_duration` seconds and
    can be used to automatically rebroadcast a pending transaction with
    a higher gas price.
    Subclass from this ABC to implement your own time gas strategy.
    """

    duration = 30

    def __init__(self, time_duration: int = 30) -> None:
        self.duration = time_duration

    def interval(self) -> int:
        return int(time.time())

    @abstractmethod
    def get_gas_price(self) -> Generator[int, None, None]:
        """
        Generator function to yield gas prices for a transaction.
        Returns
        -------
        int
            Gas price, given as an integer in wei.
        """
        raise NotImplementedError
```

- **Proxy:** จะเห็นได้ว่า ETH Brownie มีการใช้ proxy เพื่อให้เกิดการทำงานบางอย่างตามต้องการก่อนที่จะทำขั้นตอนอื่นๆ

  จาก Source Code เราพบว่า eth-brownie ใช้ Lazy Load Proxy ในการส่งคำขอไปยังบล็อคเชน ซึ่งเป็นกรณีการใช้งานทั่วไปสำหรับรูปแบบพร็อกซี

  Reference: brownie/network/multicall.py

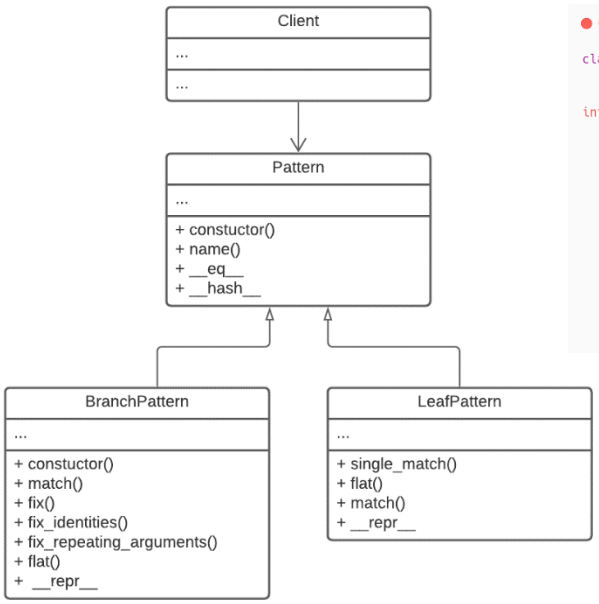



```python
class LazyResult(Proxy):
    """A proxy object to be updated with the result of a multicall."""

    def __repr__(self) -> str:
        return repr(self.__wrapped__)

class Multicall:
    """Context manager for batching multiple calls to constant contract
functions."""

    ...

    def _call_contract(self, call: ContractCall, *args: Tuple, **kwargs:
Dict[str, Any]) -> Proxy:
        """Add a call to the buffer of calls to be made"""
        calldata = (call._address, call.encode_input(*args, **kwargs))  #
type: ignore
        call_obj = Call(calldata, call.decode_output)  # type: ignore
        # future result
        result = Result(call_obj)
        self._pending_calls[get_ident()].append(result)

        return LazyResult(lambda: self._flush(result))
    ...
```

- Interpreter: ETH Brownie มีการใช้ interpreter ในการออกแบบเพื่อให้ผู้ใช้สามารถแยกวิเคราะห์เพื่อใช้ใน CLI

  จาก Source Code เราพบว่า eth-brownie ในการวิเคราะห์แยกแยะคำสั่งที่มีไวยากรณ์ที่หลากหลาย แต่สามารถทำงานผ่านคำสั่ง CLI ได้

  Reference: brownie/utils/docopt.py




```python
class Pattern:
    def __init__(
        self, name: Optional[str], value: Optional[Union[List[str], str,
int]] = None
    ) -> None:
        self._name, self.value = name, value

    @property
    def name(self) -> Optional[str]:
        return self._name

    def __eq__(self, other: Any) -> bool:
        return repr(self) == repr(other)

    def __hash__(self) -> int:
        return hash(repr(self))
```

```python
class LeafPattern(Pattern):

    """Leaf/terminal node of a pattern tree."""

    def __repr__(self) -> str:
        return "%s(%r, %r)" % (self.__class__.__name__, self.name,
self.value)

    def single_match(self, left: List["LeafPattern"]) -> TSingleMatch:
        raise NotImplementedError  # pragma: no cover

    def flat(self, *types) -> List["LeafPattern"]:
        return [self] if not types or type(self) in types else []

    def match(
        self, left: List["LeafPattern"], collected: List["Pattern"] = None
    ) -> Tuple[bool, List["LeafPattern"], List["Pattern"]]:
        collected = [] if collected is None else collected
        increment: Optional[Any] = None
        pos, match = self.single_match(left)
        if match is None or pos is None:
            return False, left, collected
        left_ = left[:pos] + left[(pos + 1) :]
        same_name = [a for a in collected if a.name == self.name]
        if type(self.value) == int and len(same_name) > 0:
            if isinstance(same_name[0].value, int):
                same_name[0].value += 1
            return True, left_, collected
        if type(self.value) == int and not same_name:
            match.value = 1
            return True, left_, collected + [match]
        if same_name and type(self.value) == list:
            if type(match.value) == str:
                increment = [match.value]
            if same_name[0].value is not None and increment is not None:
                if isinstance(same_name[0].value, type(increment)):
                    same_name[0].value += increment
            return True, left_, collected
        elif not same_name and type(self.value) == list:
            if isinstance(match.value, str):
                match.value = [match.value]
            return True, left_, collected + [match]
        return True, left_, collected + [match]
```

```python
class BranchPattern(Pattern):

    """Branch/inner node of a pattern tree."""

    def __init__(self, *children) -> None:
        self.children = list(children)

    def match(self, left: List["Pattern"], collected: List["Pattern"] =
None) -> Any:
        raise NotImplementedError  # pragma: no cover

    def fix(self) -> "BranchPattern":
        self.fix_identities()
        self.fix_repeating_arguments()
        return self

    def fix_identities(self, uniq: Optional[Any] = None) -> None:
        """Make pattern-tree tips point to same object if they are equal."""
        flattened = self.flat()
        uniq = list(set(flattened)) if uniq is None else uniq
        for i, child in enumerate(self.children):
            if not hasattr(child, "children"):
                assert child in uniq
                self.children[i] = uniq[uniq.index(child)]
            else:
                child.fix_identities(uniq)
        return None

    def fix_repeating_arguments(self) -> "BranchPattern":
        """Fix elements that should accumulate/increment values."""
        either = [list(child.children) for child in
transform(self).children]
        for case in either:
            for e in [child for child in case if case.count(child) > 1]:
                if type(e) is Argument or type(e) is Option and e.argcount:
                    if e.value is None:
                        e.value = []
                    elif type(e.value) is not list:
                        e.value = e.value.split()
                if type(e) is Command or type(e) is Option and e.argcount ==
0:
                    e.value = 0
        return self

    def __repr__(self) -> str:
        return "%s(%s)" % (self.__class__.__name__, ", ".join(repr(a) for a
in self.children))

    def flat(self, *types) -> Any:
        if type(self) in types:
            return [self]
        return sum([child.flat(*types) for child in self.children], [])
```