Generic “Composite” in Python

PyWeb - Python in Israel
August Meeting

August 07, 2024
18:00-20:30 IDT
Mobileye, Tel Aviv

mobileye

Asher Sterkin

A Software Engineer since 1978

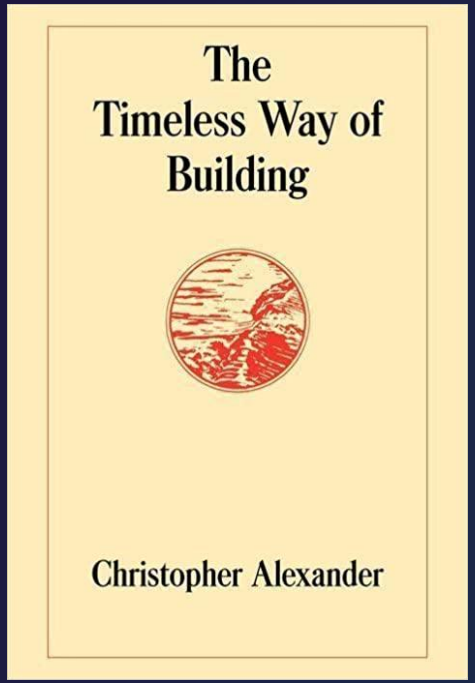
LinkedIn

Medium

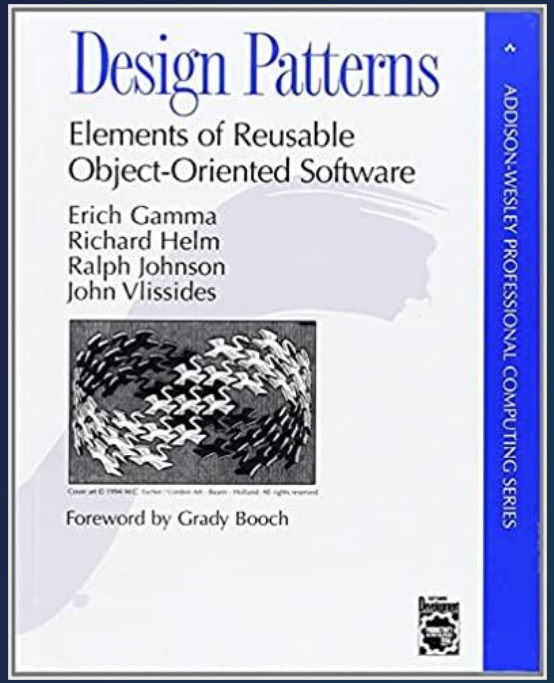
slideshare

Table of Contents

Design Patterns



Origin of the Concept of Design Patterns



Fundamental, though a bit outdated, with examples in C++ and Java

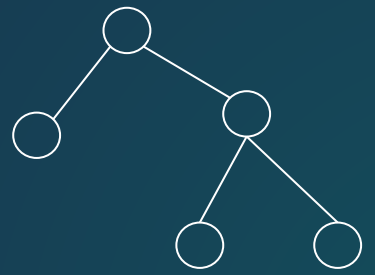
The Wikipedia definition:

“In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.”

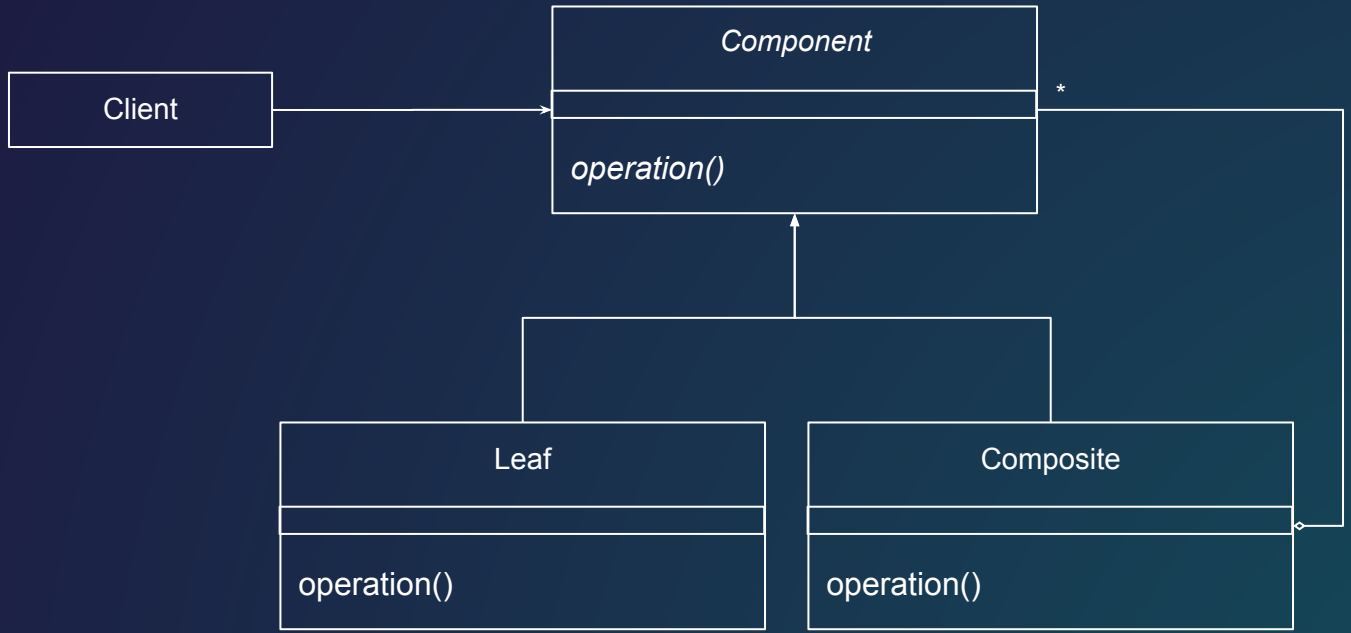
A good collection of Design Patterns in Python: https://github.com/faif/python-patterns

“Composite” Design Pattern Context

Consider the Composite Design Pattern when working with recursive tree-like data structures.



Traditional OO Implementation



Composite Design Pattern (from Wikipedia)

Traditional OO Implementation

- Implement all methods from the abstract interface
- In each Composite method:
  - Iterate over all child components
  - Call the corresponding method on each child
  - Aggregate the results into the Composite's operation result

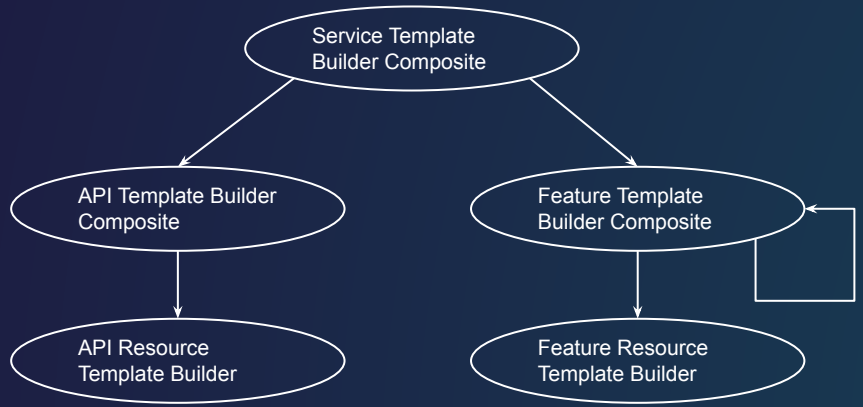
Can we develop a generic solution?

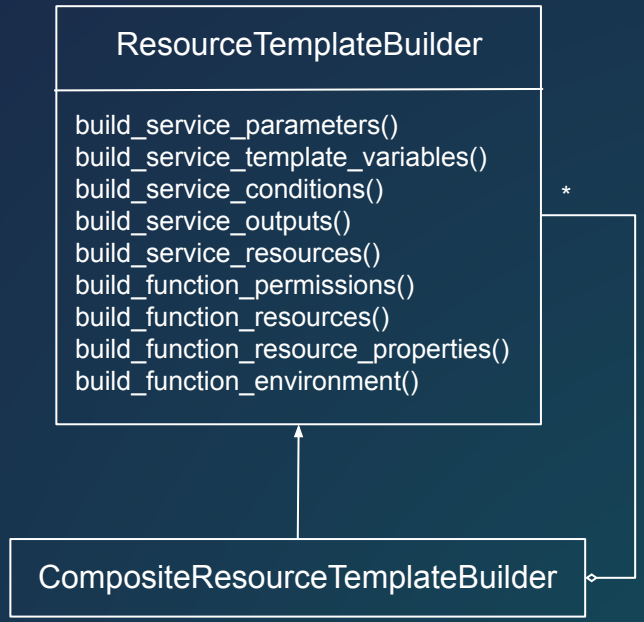
Why Invest in a Generic Solution?

- Addressed a **Real Problem** (next slide)
- **Enhanced Focus** by separating the generic subdomain from the core
- **Deeper Insight** into advanced Python capabilities and limitations

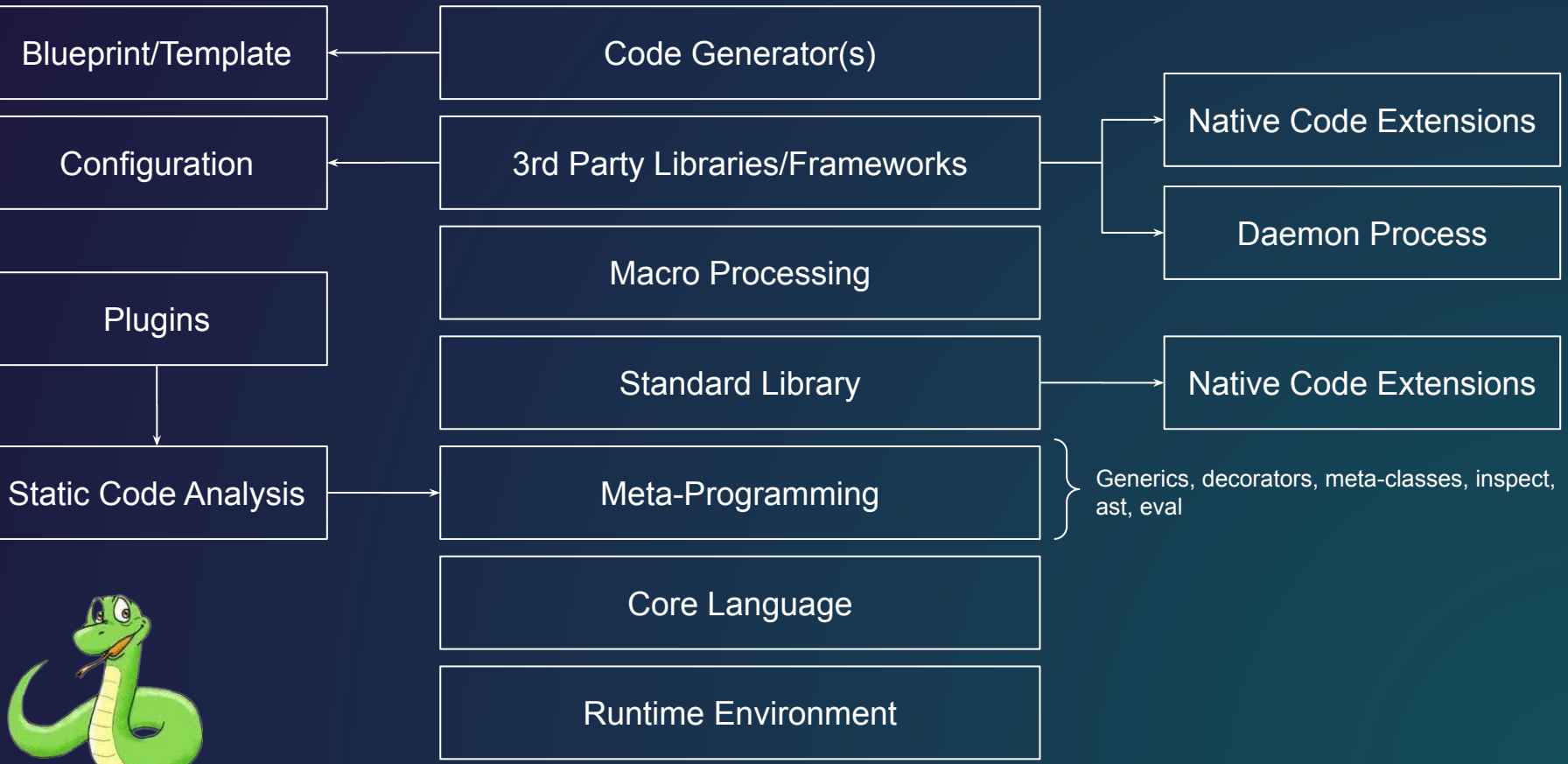
Non-Trivial Case: Service Template Builder Composite

- Every high-level feature (e.g. MySQL Database) has multiple sub-features:
  - Data on rest encryption: yes/no
  - Private network protection: yes/no
  - Interface: (e.g. db_connection vs SQLAlchemy)
  - Allocation: internal/external
  - Access: read-only/read-write
  - User authentication: yes/no
- Each sub-feature might have multiple flavours (e.g. type of encryption)
- Every feature or sub-feature requires one or more cloud resources
- Every service function has to be properly configured to get an access to each feature resources
- Every service function implements a particular API, which might bring its own resource feature(s)





Python Meta-Programming



Sample Code

```python

from unittest import TestCase
from pycomposite import composite

class Builder:
    def awesome(self) -> dict[str, int | list[int]]:
        return {}

class BuilderA(Builder):
    def awesome(self) -> dict[str, int | list[int]]:
        return dict(a=[1, 2], b=3)

class BuilderB(Builder):
    def awesome(self) -> dict[str, int | list[int]]:
        return dict(a=[4, 5], d=6)

@composite
class CompositeBuilder(Builder):
    pass

class BuilderC:
    def awesome(self) -> dict[str, int | list[int]]:
        return {}

class TestCompositeDeepMerge(TestCase):
    def setUp(self) -> None:
        # mypy will detect the error
        # self._builder = CompositeBuilder(BuilderA(), BuilderB(), BuilderC())
        self._builder = CompositeBuilder(BuilderA(), BuilderB())

    def test_deepmerge(self) -> None:
        self.assertEqual({"a": [1, 2, 4, 5], "b": 3, "d": 6}, self._builder.awesome())

```

Class Decorator

```python
_PRIVATE_METHOD_PATTERN: Pattern[str] = re.compile(r'^_{1,2}[^_]')

def _is_private(func_name: str) -> bool:
    return bool(_PRIVATE_METHOD_PATTERN.match(func_name))

_merge: Callable[[Any, Any], Any] = getattr(always_merger, 'merge')

T = TypeVar('T')

def composite(cls: Type[T]) -> Type[T]:
    """
    Generic class decorator to create a Composite from the original class.
    Notes:

    1. The constructor does not create a copy, so do not pass generators
       if you plan to invoke more than one operation.

    2. It will always return flattened results for any operation.

    :param cls: original class
    :return: Composite version of original class
    """
    def _constructor(self: T, *parts: T) -> None: …

    def _iter(self: T) -> Iterable[T]: …

    def _make_method(func_name: str, func: Callable[[Any], Any]) -> Callable[[Any], Any]: …

    def _make_methods() -> None: …

    setattr(cls, '__init__', _constructor)
    setattr(cls, '__iter__', _iter)
    setattr(cls, '__abstractmethods__', {})
    _make_methods()

    return cls
```

Constructor

```
16
17  T = TypeVar('T')
18
19  def composite(cls: Type[T]) -> Type[T]:
20      """
21      Generic class decorator to create a Composite from the original class.
22      Notes:
23
24      1. The constructor does not create a copy, so do not pass generators
25         if you plan to invoke more than one operation.
26
27      2. It will always return flattened results for any operation.
28
29      :param cls: original class
30      :return: Composite version of original class
31      """
32      def _constructor(self: T, *parts: T) -> None:
33          setattr(self, '_parts', parts)
34
```

Iterator

```
34
35      def _iter(self: T) -> Iterable[T]:
36          # Simple depth-first composite Iterator
37          # Recursive version did not work for some mysterious reason
38          # This one proved to be more reliable
39          # Credit: https://stackoverflow.com/questions/26145678/implementing-a-depth-first-tree-iterator-in-python
40          stack = deque(getattr(self, '_parts'))
41          while stack:
42              part = stack.popleft()
43              if cls == type(part):
44                  stack.extendleft(reversed(getattr(part, '_parts')))
45              elif isinstance(part, cls) or not isinstance(part, Iterable):
46                  yield part
47              else:
48                  stack.extendleft(reversed(part))
```

Methods

```
    def _make_method(func_name: str, func: Callable[[Any], Any]) -> Callable[[Any], Any]:
        def _make_reduce(m: str, rt: type) -> Callable[[Any], Any]:
            init_value = rt()
            combine: Callable[[Any, Any], Any] = add if rt in (int, str, tuple) else _merge

            @wraps(func)
            def _reduce_parts(self: Iterable[T], *args: Any, **kwargs: Any) -> Any:
                def _call(acc: Any, obj: Any) -> Any:
                    m_: Callable[[Any, Any], Any] = getattr(obj, m)
                    result: Any = m_(*args, **kwargs)
                    return combine(acc, result)

                return reduce(
                    _call,
                    self,
                    init_value,
                )
            return _reduce_parts

        def _make_foreach(m: str) -> Callable[[Any], Any]:
            @wraps(func)
            def _foreach_parts(self: T, *args: Any, **kwargs: Any) -> None:
                for obj in getattr(self, '__iter__')():
                    getattr(obj, m)(*args, **kwargs)
            return _foreach_parts

        rt_ = signature(func).return_annotation
        rt = get_origin(rt_) or rt_
        method = _make_foreach(func_name) if rt is None else _make_reduce(func_name, rt)
        setattr(method, '__isabstract__', False)
        return method

    def _make_methods() -> None:
        for func_name, func in getmembers(cls, predicate=isfunction):
            if not _is_private(func_name):
                setattr(cls, func_name, _make_method(func_name, func))
```

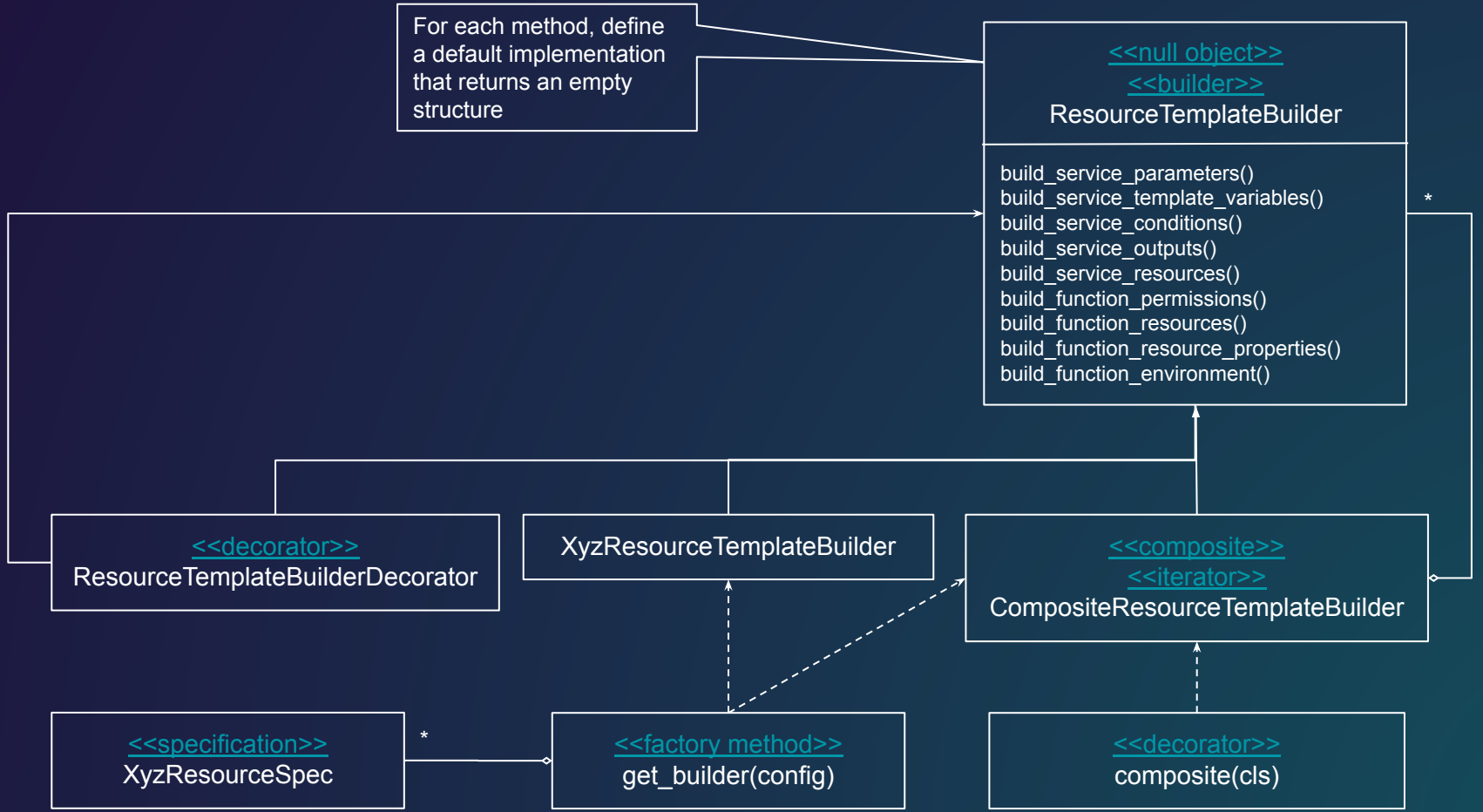
Mypy Plugin

```python

from mypy.plugins.common import add_method
from typing import Type, Callable, Optional
from mypy.types import NoneType, UnionType
from mypy.nodes import ARG_STAR, Argument, Var
from mypy.plugin import Plugin, ClassDefContext

class MyPlugin(Plugin):
    def get_class_decorator_hook(self, fullname: str) -> Optional[Callable[[ClassDefContext], None]]:
        if fullname == 'pycomposite.composite_decorator.composite':
            return class_decorator_hook
        return None

def class_decorator_hook(ctx: ClassDefContext) -> None:
    # Add the __init__ and __iter__ methods dynamically
    # Extract base class type
    if not ctx.cls.info.bases:
        return
    base_type = ctx.cls.info.bases[0]

    # Create type for parts
    # Create Iterable[base_type]
    iterable_type = ctx.api.named_type('typing.Iterable', [base_type])
    # Create Union[base_type, Iterable[base_type]]
    parts_type = UnionType.make_union([base_type, iterable_type])
    parts_var = Var('parts', parts_type)
    parts_arg = Argument(variable=parts_var, type_annotation=parts_type, initializer=None, kind=ARG_STAR)

    add_method(ctx, '__init__', [parts_arg], NoneType())

    iterator_type = ctx.api.named_type('typing.Iterator', [base_type])
    add_method(ctx, '__iter__', [], iterator_type)


def plugin(version: str) -> Type[Plugin]:
    return MyPlugin

```

Design Pattern Density

- The measurement of the amount of design that can be represented as instances of design patterns
- When applied correctly, higher design pattern density implies a higher maturity of the design.
- When applied incorrectly, leads to disastrous over-engineering.

Putting All Patterns Together



Limitations (Reflect my Practical Needs)

- No Visitor Design Pattern implementation
- Limited set of aggregation functions (add and merge)
- No support for Abstract Base Classes by mypy Plugin
- Type “cheating” within decorator

Things to Remember

- Design Patterns are extremely powerful tools
- Design Patterns work best in concert (high density)
- Composite Design Pattern is the first choice for hierarchical data structures
- Python metaprogramming could make miracles
- Generic solutions usually work better
- Python metaprogramming is imperfect
- With more power comes more responsibility
- Chase after advanced techniques for no reason is a sure road to hell

Discussion:

- Design Patterns
- Metaprogramming
- LLMs
- Code Assistants