Criação:

Padrão de Criação de Classe - Factory Method

```python
from abc import ABC, abstractmethod

# Produto
class Product(ABC):
    @abstractmethod
    def display(self) -> None:
        pass

# Produto Concreto
class ConcreteProduct(Product):
    def display(self) -> None:
        print("Concrete Product")

# Criador (Classe base com Factory Method)
class Creator(ABC):
    @abstractmethod
    def factory_method(self) -> Product:
        pass

    def operation(self) -> None:
        product = self.factory_method()
        product.display()

# Criador Concreto
class ConcreteCreator(Creator):
    def factory_method(self) -> Product:
        return ConcreteProduct()

# Testando o Factory Method
def test_factory_method() -> None:
    creator = ConcreteCreator()
    creator.operation()
```

Creator é a classe base que contém o factory_method.

ConcreteCreator é uma subclasse de Creator que implementa o factory_method, que cria uma instância de ConcreteProduct.

O método operation é usado para realizar operações no produto criado.

Padrão de Criação de Objeto - Abstract Factory

```
from abc import ABC, abstractmethod


# Abstract Factory
class AbstractFactory(ABC):
    @abstractmethod
    def create_product_a(self):
        pass


    @abstractmethod
    def create_product_b(self):
        pass


# Concrete Factory
class ConcreteFactory1(AbstractFactory):
    def create_product_a(self):
        return ProductA1()


    def create_product_b(self):
        return ProductB1()


# Produto A
class AbstractProductA(ABC):
    @abstractmethod
    def display(self) -> None:
        pass
```

```
# Produto Concreto A1
class ProductA1(AbstractProductA):
    def display(self) -> None:
        print("Product A1")

# Produto B
class AbstractProductB(ABC):
    @abstractmethod
    def display(self) -> None:
        pass

# Produto Concreto B1
class ProductB1(AbstractProductB):
    def display(self) -> None:
        print("Product B1")

# Testando Abstract Factory
def test_abstract_factory() -> None:
    factory = ConcreteFactory1()
    product_a = factory.create_product_a()
    product_b = factory.create_product_b()
    product_a.display()
    product_b.display()

# Executando o teste
test_abstract_factory()
```

AbstractFactory é a interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

ConcreteFactory1 é uma implementação concreta de AbstractFactory que cria produtos concretos ProductA1 e ProductB1.

Os produtos concretos implementam interfaces específicas (AbstractProductA e AbstractProductB).

Padrão de Criação de Classe - Builder:

from abc import ABC, abstractmethod

```
# Produto
class Product:
    def __init__(self):
        self.parts = []

    def add_part(self, part):
        self.parts.append(part)

    def display(self):
        print("Product Parts:", ", ".join(self.parts))

# Builder
class Builder(ABC):
    @abstractmethod
```

```
    def build_part_a(self):
        pass

    @abstractmethod
    def build_part_b(self):
        pass

    @abstractmethod
    def get_result(self):
        pass

# Concrete Builder
class ConcreteBuilder(Builder):
    def __init__(self):
        self.product = Product()

    def build_part_a(self):
        self.product.add_part("Part A")

    def build_part_b(self):
        self.product.add_part("Part B")

    def get_result(self):
        return self.product

# Director
class Director:
    def construct(self, builder):
```

```
        builder.build_part_a()
        builder.build_part_b()


# Testando o Builder
def test_builder() -> None:
    builder = ConcreteBuilder()
    director = Director()
    director.construct(builder)
    product = builder.get_result()
    product.display()


# Executando o teste
test_builder()
```

Builder é a interface para criar partes de um produto.

ConcreteBuilder é uma implementação concreta de Builder que constrói um produto específico (Product) com suas partes.

Director é responsável por construir um produto usando um builder específico.

Padrão de Criação de Objeto - Prototype:

```
from copy import deepcopy


# Prototype
class Prototype:
    def clone(self):
        return deepcopy(self)
```

```
# Concrete Prototype
class ConcretePrototype(Prototype):
    def __init__(self, attribute):
        self.attribute = attribute

    def display(self):
        print(f"ConcretePrototype - Attribute: {self.attribute}")

# Testando o Prototype
def test_prototype() -> None:
    prototype = ConcretePrototype("Original Attribute")
    clone = prototype.clone()

    prototype.display()
    clone.display()

# Executando o teste
test_prototype()
```

Prototype é a interface que declara o método de clonagem.

ConcretePrototype é uma implementação concreta de Prototype que possui um atributo.

O método clone faz uma cópia profunda do objeto, permitindo criar novos objetos clonados sem afetar o original.

ESTRUTURA:

Padrão de Estrutura de Classe - Class Adapter:

```
from abc import ABC, abstractmethod

# Adaptee
class Adaptee:
    def specific_request(self):
        return "Request from Adaptee"

# Target
class Target(ABC):
    @abstractmethod
    def request(self):
        pass

# Adapter
class ClassAdapter(Adaptee, Target):
    def request(self):
        return f"Class Adapter: {self.specific_request()}"

# Testando o Class Adapter
def test_class_adapter() -> None:
    adapter = ClassAdapter()
    result = adapter.request()
    print(result)

# Executando o teste
```

```
test_class_adapter()
```

Adaptee é a classe existente com uma interface incompatível.

Target é a interface desejada para interagir com Adaptee.

ClassAdapter é uma classe adaptadora que herda de Adaptee e implementa Target, adaptando a interface.

Padrão de Estrutura de Objeto - Object Adapter:

```
from abc import ABC, abstractmethod


# Adaptee
class Adaptee:
    def specific_request(self):
        return "Request from Adaptee"


# Target
class Target(ABC):
    @abstractmethod
    def request(self):
        pass


# Adapter
class ObjectAdapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee
```

```
    def request(self):
        return f"Object Adapter: {self.adaptee.specific_request()}"

# Testando o Object Adapter
def test_object_adapter() -> None:
    adaptee = Adaptee()
    adapter = ObjectAdapter(adaptee)
    result = adapter.request()
    print(result)

# Executando o teste
test_object_adapter()
```

ObjectAdapter é uma classe adaptadora que contém uma instância de Adaptee e implementa Target, adaptando a interface.

Ambos os padrões permitem que um cliente utilize a interface desejada (Target) para interagir com a classe existente (Adaptee) de uma maneira transparente.

Comportamental:

Padrão Comportamental de Classe - Template Method:

```
from abc import ABC, abstractmethod

class Funcionario(ABC):
    def __init__(self, salario):
```

```
        self._salario = salario

    @abstractmethod
    def calcDescontosPrevidencia(self) -> float:
        pass

    @abstractmethod
    def calcDescontosPlanoSaude(self) -> float:
        pass

    @abstractmethod
    def calcOutrosDescontos(self) -> float:
        pass

    def calcSalarioLiquido(self) -> float:
        prev: float = self.calcDescontosPrevidencia()
        saude: float = self.calcDescontosPlanoSaude()
        outros: float = self.calcOutrosDescontos()

        return self._salario - prev - saude - outros

class FuncionarioCLT(Funcionario):
    def __init__(self, salario):
        super().__init__(salario)

    def calcDescontosPrevidencia(self) -> float:
        return self._salario * 0.1
```

```
    def calcDescontosPlanoSaude(self) -> float:
        return 100.0

    def calcOutrosDescontos(self) -> float:
        return 20.0

# Testando o Template Method
def test_template_method() -> None:
    func: FuncionarioCLT = FuncionarioCLT(1000.0)
    salario: float = func.calcSalarioLiquido()
    print(f"Salário Líquido: {salario}")

# Executando o teste
test_template_method()
```

Funcionario é a classe abstrata que define o Template Method calcSalarioLiquido.

FuncionarioCLT é uma classe concreta que herda de Funcionario e implementa os métodos abstratos.

Padrão Comportamental de Objeto - Iterator:

```
from __future__ import annotations
from collections.abc import Iterable, Iterator
from typing import Any, List

class IteradorListaPalavras(Iterator):
    _posicao: int = 0
    _reverso: bool = False
```

```
    def __init__(self, palavras: ListaPalavras, reverso: bool = False) -> None:
        self._palavras = palavras
        self._reverso = reverso
        self._posicao = -1 if reverso else 0

    def __next__(self):
        try:
            item: str = self._palavras[self._posicao]
            self._posicao += -1 if self._reverso else 1
        except IndexError:
            raise StopIteration()

        return item

class ListaPalavras(Iterable):
    def __init__(self, palavras: List[str] = []) -> None:
        self._palavras = palavras

    def __getitem__(self, index: int) -> str:
        return self._palavras[index]

    def __iter__(self) -> IteradorListaPalavras:
        return IteradorListaPalavras(self)

    def get_reverse_iterator(self) -> IteradorListaPalavras:
        return IteradorListaPalavras(self, True)
```

```
    def add_item(self, item: str):
        self._palavras.append(item)
```

No exemplo apresentado, temos as classes IteradorListaPalavras e ListaPalavras. A classe IteradorListaPalavras implementa a interface Iterator, fornecendo a capacidade de iterar sobre os elementos da ListaPalavras.