```
##############################################################################
# Singleton
##############################################################################
"""
In software engineering, the singleton pattern is a
software design pattern that restricts the instantiation of a class to one
"single" instance. This is useful when exactly one object is needed to
 coordinate actions across the system.
The term comes from the mathematical concept of a singleton.
"""

def Singleton(cls):
  _instances = {}
  def getinstance():
    if cls not in _instances:
      _instances[cls] = cls()
    return _instances[cls]
  return getinstance

@Singleton
class Counter(object):
  def __init(self):
    if not hasattr(self ,'val'): self.val = 0
  def get(self): return self.val
  def incr(self): self.val += 1

##############################################################################
# Factory
##############################################################################
"""
Create objects from a set of classes.
Implementation depends on class definitions.
Introduction of new classes and other changes need recompilation.
Class constructors exposed.


Factory: Use interface functions/objects to
encapsulate class names and constructors.
Use a interface functions, methods to provide you instances.
Rest is handled by polymorphism.
"""
@Singleton
class PlayerFactory(object):
  def new(self, name, type):
    if type == 'peasant':
      return Peasant(name)
    elif type == 'warior':
      return Warior(name)
    return None

##############################################################################
# Proxy
```

```
################################################################################
"""
You need to access a hard to duplicate, limited, probably old class definition.
You donot have a chance to improve it or change the interface.
Or, you want to have restricted access to methods (authorization).
Or, you want smarter access like caching.

Proxy: Write a class interface implementing
functionalities missing in the original interface.

"""

class CounterProxy(object):
    def __init__(self, type):
        self.type = type
    def incr(self):
        if self.type == 'W':
            Counter().incr()
    def get(self):
        Counter().get()

################################################################################
# Facade
################################################################################
"""
A complex library, tool or system; consisting of many functions and/or classes;
 probably poorly designed is tried to be accessed. It is hard to read and
 understand. There are many dependencies distributed in the source,
 needs many housekeeping tasks and stages to access.
 Any change in the system require changes in the whole source code.


Facade:Define a class implementing all details of the library/system
and providing a simple uniform interface. Access the library through
this interface.
"""
class AuthFacade(object):
    def __init__(self ,method):
        if method == 'passwd':
            #
        elif method =='oauth':
            #
        elif method =='otp':
            #
        else:
                throw ...
    def auth(self,identity,data):
        #check authentication based on setting



################################################################################
```

```python
# Observer
##############################################################################
"""
Objects depending on
eachothers states need to be informed when the other encountered a change.
 Event handling systems.


Observer: Maintain a registry of observing objects in Subject object.
 When an event occurs, notify the observers.
"""


class Subject(object):
    _observers = []
    def register(self, obs):
        self._observers.append(obs)
    def unregister(self, obs):
        self._observers.remove(obs)
    def notify(self):
        for o in self._observers:
            o.update(self)
    def state(self): pass
class Observer(object):
    def update(self, subj): pass
class Clock(Subject):
    def __init__(self):
        self.value = 0
    def state(self):
        return self.value
    def tick(self):
        self.value += 1
        self.notify()
class Person(Observer):
    def update(self, obj):
        print "heyo", obj.state()
a = Person()
b = Clock()
b.register(a)
b.tick()


##############################################################################
# Producer
##############################################################################
"""
The producer / consumer design pattern is a pre-designed solution to separate
the two main components by placing a queue in the middle, letting the producers
and the consumers execute in different threads.

"""
```

```python
class Producer(Thread):

  def __init__(self, queue, cvs, maxsize=5):
    Thread.__init__(self)
    self.queue = queue
    self.maxsize = maxsize
    self.notfull = cvs[0]
    self.notempty = cvs[1]
    self.terminate = False
    self.counter = 0

  def run(self):
    while not self.terminate:
      sleep(0.1 * randint(0, 10)) # sleep randomly

      self.notfull.acquire()
      while len(self.queue) >= self.maxsize: # full
        print "full queue, waiting"
        self.notfull.wait()
        if self.terminate:
          break
      self.notfull.release()
      self.counter += 1
      self.queue.append(self.counter)
      self.notempty.acquire()
      self.notempty.notify() # notify consumer, a new item
      self.notempty.release()

  def quit(self):
    self.terminate = True
    self.notfull.acquire()
    self.notfull.notify()
    self.notfull.release()


################################################################################
# Chain of responsibilty
################################################################################
"""
This pattern gives us a way to treat a request using different methods,
 each one addressing a specific part of the request.
You know, one of the best principles for good code is the
Single Responsibility principle.

Every piece of code must do one, and only one, thing.

For example, if we want to filter some content we can implement different
filters, each one doing one precise and clearly defined type of filtering.
These filters could be used to filter offensive words, ads,
unsuitable video content, and so on.

"""
```

```python
class ContentFilter(object):
    def __init__(self, filters=None):
        self._filters = list()
        if filters is not None:
            self._filters += filters

    def filter(self, content):
        for filter in self._filters:
            content = filter(content)
        return content

filter = ContentFilter([
                offensive_filter,
                ads_filter,
                porno_video_filter])
filtered_content = filter.filter(content)



###############################################################################
# Command
###############################################################################
"""
The command pattern is handy in situations when, for some reason,
we need to start by preparing what will be executed and then to execute it when needed.

The advantage is that encapsulating actions in such a way enables Python
developers to add additional functionalities related to the executed actions,
such as undo/redo, or keeping a history of actions and the like.
"""

class RenameFileCommand(object):
    def __init__(self, from_name, to_name):
        self._from = from_name
        self._to = to_name

    def execute(self):
        os.rename(self._from, self._to)

    def undo(self):
        os.rename(self._to, self._from)

class History(object):
    def __init__(self):
        self._commands = list()

    def execute(self, command):
        self._commands.append(command)
        command.execute()

    def undo(self):
        self._commands.pop().undo()
```

```python
history = History()
history.execute(RenameFileCommand('docs/cv.doc', 'docs/cv-en.doc'))
history.execute(RenameFileCommand('docs/cv1.doc', 'docs/cv-bg.doc'))
history.undo()
history.undo()

##############################################################################
# Depndency Injection
##############################################################################
"""
very good mechanism of implementing loose couplings,
and it helps make our application maintainable and extendable.

The advantage is that encapsulating actions in such a way enables Python
developers to add additional functionalities related to the executed actions,
such as undo/redo, or keeping a history of actions and the like.
"""
# Inject dependencies through the constructor
class Command:
    def __init__(self, authenticate=None, authorize=None):
        self.authenticate = authenticate or self._not_authenticated
        self.authorize = authorize or self._not_autorized

    def execute(self, user, action):
        self.authenticate(user)
        self.authorize(user, action)
        return action()

if in_sudo_mode:
    command = Command(always_authenticated, always_authorized)
else:
    command = Command(config.authenticate, config.authorize)
command.execute(current_user, delete_user_action)

# Inject dependencies by setting directly the object properties

command = Command()

if in_sudo_mode:
    command.authenticate = always_authenticated
    command.authorize = always_authorized
else:
    command.authenticate = config.authenticate
    command.authorize = config.authorize
command.execute(current_user, delete_user_action)




##############################################################################
# Adapter
##############################################################################
```

```
"""
Adapters are all about altering the interface.
Like using a cow when the system is expecting a duck.

"""
import socket

class SocketWriter(object):

    def __init__(self, ip, port):
        self._socket = socket.socket(socket.AF_INET,
                                     socket.SOCK_DGRAM)
        self._ip = ip
        self._port = port

    def write(self, message):
        self._socket.send(message, (self._ip, self._port))

def log(message, destination):
    destination.write('[{}] - {}'.format(datetime.now(), message))

upd_logger = SocketWriter('1.2.3.4', '9999')
log('Something happened', udp_destination)

###############################################################################
# DECORATOR
###############################################################################
"""
The decorator pattern is about introducing additional
functionality and in particular, doing it without using inheritance.

"""
def autheticated_only(method):
    def decorated(*args, **kwargs):
        if check_authenticated(kwargs['user']):
            return method(*args, **kwargs )
        else:
            raise UnauthenticatedError
    return decorated


def authorized_only(method):
    def decorated(*args, **kwargs):
        if check_authorized(kwargs['user'], kwargs['action']):
            return method(*args, **kwargs)
        else:
            raise UnauthorizedError
    return decorated


@authorized_only
@authenticated_only
```

```python
def execute(action, *args, **kwargs):
    return action()




###############################################################################
# Iterator
###############################################################################
"""
Lets you traverse elements of a collection without exposing its
underlying representation (list, stack, tree, etc.).


Usage examples: The pattern is very common in Python code.
Many frameworks and libraries use it to provide a
standard way for traversing their collections.
"""

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


"""
To create an iterator in Python, there are two abstract classes from the built-
in `collections` module - Iterable,Iterator. We need to implement the
`__iter__()` method in the iterated object (collection), and the `__next__ ()`
method in theiterator.
"""


class AlphabeticalOrderIterator(Iterator):
    """
    Concrete Iterators implement various traversal algorithms. These classes
    store the current traversal position at all times.
    """

    """
    `_position` attribute stores the current traversal position. An iterator may
    have a lot of other fields for storing iteration state, especially when it
    is supposed to work with a particular kind of collection.
    """
    _position: int = None

    """
    This attribute indicates the traversal direction.
    """
    _reverse: bool = False

    def __init__(self, collection: WordsCollection, reverse: bool = False) -> None:
        self._collection = collection
        self._reverse = reverse
        self._position = -1 if reverse else 0

    def __next__(self):
        """
        The __next__() method must return the next item in the sequence. On
        reaching the end, and in subsequent calls, it must raise StopIteration.
        """
        try:
            value = self._collection[self._position]
            self._position += -1 if self._reverse else 1
        except IndexError:
            raise StopIteration()

        return value


class WordsCollection(Iterable):
    """
    Concrete Collections provide one or several methods for retrieving fresh
    iterator instances, compatible with the collection class.
    """

    def __init__(self, collection: List[Any] = []) -> None:
        self._collection = collection

    def __iter__(self) -> AlphabeticalOrderIterator:
        """
        The __iter__() method returns the iterator object itself, by default we
        return the iterator in ascending order.
        """
        return AlphabeticalOrderIterator(self._collection)

    def get_reverse_iterator(self) -> AlphabeticalOrderIterator:
        return AlphabeticalOrderIterator(self._collection, True)

    def add_item(self, item: Any):
        self._collection.append(item)


if __name__ == "__main__":
    # The client code may or may not know about the Concrete Iterator or
    # Collection classes, depending on the level of indirection you want to keep
    # in your program.
    collection = WordsCollection()
    collection.add_item("First")
    collection.add_item("Second")
    collection.add_item("Third")

    print("Straight traversal:")
    print("\n".join(collection))
    print("")
```

```python
    print("Reverse traversal:")
    print("\n".join(collection.get_reverse_iterator()), end="")


# Output.txt: Execution result
# Straight traversal:
# First
# Second
# Third
#
# Reverse traversal:
# Third
# Second
# First



################################################################################
# Bridge
################################################################################
"""
Lets you split a large class or a set of closely related classes into two
separate hierarchies—abstraction and implementation—which can be
developed independently of each other.

Usage examples: The Bridge pattern is especially useful when dealing with
cross-platform apps, supporting multiple types of database servers or working
 with several API providers of a certain kind
 (for example, cloud platforms, social networks, etc.)


Identification: Bridge can be recognized by a clear distinction between some
controlling entity and several different platforms that it relies on.
"""
from __future__ import annotations
from abc import ABC, abstractmethod


class Abstraction:
    """
    The Abstraction defines the interface for the "control" part of the two
    class hierarchies. It maintains a reference to an object of the
    Implementation hierarchy and delegates all of the real work to this object.
    """

    def __init__(self, implementation: Implementation) -> None:
        self.implementation = implementation

    def operation(self) -> str:
        return (f"Abstraction: Base operation with:\n"
                f"{self.implementation.operation_implementation()}")
```

```python
class ExtendedAbstraction(Abstraction):
    """
    You can extend the Abstraction without changing the Implementation classes.
    """

    def operation(self) -> str:
        return (f"ExtendedAbstraction: Extended operation with:\n"
                f"{self.implementation.operation_implementation()}")


class Implementation(ABC):
    """
    The Implementation defines the interface for all implementation classes. It
    doesn't have to match the Abstraction's interface. In fact, the two
    interfaces can be entirely different. Typically the Implementation interface
    provides only primitive operations, while the Abstraction defines higher-
    level operations based on those primitives.
    """

    @abstractmethod
    def operation_implementation(self) -> str:
        pass


"""
Each Concrete Implementation corresponds to a specific platform and implements
the Implementation interface using that platform's API.
"""


class ConcreteImplementationA(Implementation):
    def operation_implementation(self) -> str:
        return "ConcreteImplementationA: Here's the result on the platform A."


class ConcreteImplementationB(Implementation):
    def operation_implementation(self) -> str:
        return "ConcreteImplementationB: Here's the result on the platform B."


def client_code(abstraction: Abstraction) -> None:
    """
    Except for the initialization phase, where an Abstraction object gets linked
    with a specific Implementation object, the client code should only depend on
    the Abstraction class. This way the client code can support any abstraction-
    implementation combination.
    """

    # ...
    print(abstraction.operation(), end="")

    # ...


if __name__ == "__main__":
    """
    The client code should be able to work with any pre-configured abstraction-
    implementation combination.
    """

    implementation = ConcreteImplementationA()
    abstraction = Abstraction(implementation)
    client_code(abstraction)

    print("\n")

    implementation = ConcreteImplementationB()
    abstraction = ExtendedAbstraction(implementation)
    client_code(abstraction)

# Output.txt: Execution result
# Abstraction: Base operation with:
# ConcreteImplementationA: Here's the result on the platform A.
#
# ExtendedAbstraction: Extended operation with:
# ConcreteImplementationB: Here's the result on the platform B.




################################################################################
# Builder
################################################################################
"""
Lets you construct complex objects step by step.
The pattern allows you to produce different types and representations of an
object using the same construction code.

Usage examples: The Builder pattern is a well-known pattern in Python world.
 It’s especially useful when you need to create an object with lots of
  possible configuration options.

Identification: The Builder pattern can be recognized in a class,
 which has a single creation method and several methods to configure the
 resulting object. Builder methods often support chaining
  (for example, someBuilder.setValueA(1).setValueB(2).create()).
"""

from __future__ import annotations
from abc import ABC, abstractmethod, abstractproperty
from typing import Any
```

```python
class Builder(ABC):
    """
    The Builder interface specifies methods for creating the different parts of
    the Product objects.
    """

    @abstractproperty
    def product(self) -> None:
        pass

    @abstractmethod
    def produce_part_a(self) -> None:
        pass

    @abstractmethod
    def produce_part_b(self) -> None:
        pass

    @abstractmethod
    def produce_part_c(self) -> None:
        pass


class ConcreteBuilder1(Builder):
    """
    The Concrete Builder classes follow the Builder interface and provide
    specific implementations of the building steps. Your program may have
    several variations of Builders, implemented differently.
    """

    def __init__(self) -> None:
        """
        A fresh builder instance should contain a blank product object, which is
        used in further assembly.
        """
        self.reset()

    def reset(self) -> None:
        self._product = Product1()

    @property
    def product(self) -> Product1:
        """
        Concrete Builders are supposed to provide their own methods for
        retrieving results. That's because various types of builders may create
        entirely different products that don't follow the same interface.
        Therefore, such methods cannot be declared in the base Builder interface
        (at least in a statically typed programming language).

        Usually, after returning the end result to the client, a builder
        instance is expected to be ready to start producing another product.
        That's why it's a usual practice to call the reset method at the end of
        the `getProduct` method body. However, this behavior is not mandatory,
        and you can make your builders wait for an explicit reset call from the
        client code before disposing of the previous result.
        """
        product = self._product
        self.reset()
        return product

    def produce_part_a(self) -> None:
        self._product.add("PartA1")

    def produce_part_b(self) -> None:
        self._product.add("PartB1")

    def produce_part_c(self) -> None:
        self._product.add("PartC1")


class Product1():
    """
    It makes sense to use the Builder pattern only when your products are quite
    complex and require extensive configuration.

    Unlike in other creational patterns, different concrete builders can produce
    unrelated products. In other words, results of various builders may not
    always follow the same interface.
    """

    def __init__(self) -> None:
        self.parts = []

    def add(self, part: Any) -> None:
        self.parts.append(part)

    def list_parts(self) -> None:
        print(f"Product parts: {', '.join(self.parts)}", end="")


class Director:
    """
    The Director is only responsible for executing the building steps in a
    particular sequence. It is helpful when producing products according to a
    specific order or configuration. Strictly speaking, the Director class is
    optional, since the client can control builders directly.
    """

    def __init__(self) -> None:
        self._builder = None

    @property
    def builder(self) -> Builder:
        return self._builder

    @builder.setter
    def builder(self, builder: Builder) -> None:
        """
        The Director works with any builder instance that the client code passes
        to it. This way, the client code may alter the final type of the newly
        assembled product.
        """
        self._builder = builder

    """
    The Director can construct several product variations using the same
    building steps.
    """

    def build_minimal_viable_product(self) -> None:
        self.builder.produce_part_a()

    def build_full_featured_product(self) -> None:
        self.builder.produce_part_a()
        self.builder.produce_part_b()
        self.builder.produce_part_c()


if __name__ == "__main__":
    """
    The client code creates a builder object, passes it to the director and then
    initiates the construction process. The end result is retrieved from the
    builder object.
    """

    director = Director()
    builder = ConcreteBuilder1()
    director.builder = builder

    print("Standard basic product: ")
    director.build_minimal_viable_product()
    builder.product.list_parts()

    print("\n")

    print("Standard full featured product: ")
    director.build_full_featured_product()
    builder.product.list_parts()

    print("\n")

    # Remember, the Builder pattern can be used without a Director class.
    print("Custom product: ")
    builder.produce_part_a()
    builder.produce_part_b()
    builder.product.list_parts()

#  Output.txt: Execution result
# Standard basic product:
# Product parts: PartA1
#
# Standard full featured product:
# Product parts: PartA1, PartB1, PartC1
#
# Custom product:
# Product parts: PartA1, PartB1


###############################################################################
# Visitor
###############################################################################
"""
Lets you separate algorithms from the objects on which they operate.

Usage examples: Visitor isn’t a very common pattern because
of its complexity and narrow applicability.
"""
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import List


class Component(ABC):
    """
    The Component interface declares an `accept` method that should take the
    base visitor interface as an argument.
    """

    @abstractmethod
    def accept(self, visitor: Visitor) -> None:
        pass


class ConcreteComponentA(Component):
    """
    Each Concrete Component must implement the `accept` method in such a way
    that it calls the visitor's method corresponding to the component's class.
    """

    def accept(self, visitor: Visitor) -> None:
        """
        Note that we're calling `visitConcreteComponentA`, which matches the
        current class name. This way we let the visitor know the class of the
        component it works with.
        """

        visitor.visit_concrete_component_a(self)
    def exclusive_method_of_concrete_component_a(self) -> str:
        """
        Concrete Components may have special methods that don't exist in their
        base class or interface. The Visitor is still able to use these methods
        since it's aware of the component's concrete class.
        """

        return "A"


class ConcreteComponentB(Component):
    """
    Same here: visitConcreteComponentB => ConcreteComponentB
    """

    def accept(self, visitor: Visitor):
        visitor.visit_concrete_component_b(self)

    def special_method_of_concrete_component_b(self) -> str:
        return "B"


class Visitor(ABC):
    """
    The Visitor Interface declares a set of visiting methods that correspond to
    component classes. The signature of a visiting method allows the visitor to
    identify the exact class of the component that it's dealing with.
    """

    @abstractmethod
    def visit_concrete_component_a(self, element: ConcreteComponentA) -> None:
        pass

    @abstractmethod
    def visit_concrete_component_b(self, element: ConcreteComponentB) -> None:
        pass


"""
Concrete Visitors implement several versions of the same algorithm, which can
work with all concrete component classes.

You can experience the biggest benefit of the Visitor pattern when using it with
a complex object structure, such as a Composite tree. In this case, it might be
helpful to store some intermediate state of the algorithm while executing
visitor's methods over various objects of the structure.
"""


class ConcreteVisitor1(Visitor):
    def visit_concrete_component_a(self, element) -> None:
        print(f"{element.exclusive_method_of_concrete_component_a()} + ConcreteVisitor1")

    def visit_concrete_component_b(self, element) -> None:
        print(f"{element.special_method_of_concrete_component_b()} + ConcreteVisitor1")


class ConcreteVisitor2(Visitor):
    def visit_concrete_component_a(self, element) -> None:
        print(f"{element.exclusive_method_of_concrete_component_a()} + ConcreteVisitor2")

    def visit_concrete_component_b(self, element) -> None:
        print(f"{element.special_method_of_concrete_component_b()} + ConcreteVisitor2")


def client_code(components: List[Component], visitor: Visitor) -> None:
    """
    The client code can run visitor operations over any set of elements without
    figuring out their concrete classes. The accept operation directs a call to
    the appropriate operation in the visitor object.
    """

    # ...
    for component in components:
        component.accept(visitor)
    # ...


if __name__ == "__main__":
    components = [ConcreteComponentA(), ConcreteComponentB()]

    print("The client code works with all visitors via the base Visitor interface:")
    visitor1 = ConcreteVisitor1()
    client_code(components, visitor1)

    print("It allows the same client code to work with different types of visitors:")
    visitor2 = ConcreteVisitor2()
    client_code(components, visitor2)

# Output.txt: Execution result
# The client code works with all visitors via the base Visitor interface:
# A + ConcreteVisitor1
# B + ConcreteVisitor1
# It allows the same client code to work with different types of visitors:
# A + ConcreteVisitor2
# B + ConcreteVisito
```