**Правительство Российской Федерации**

**Государственное образовательное бюджетное учреждение**

**высшего профессионального образования**

**«Научно-исследовательский университет –**

**Высшая школа экономики»**

**Факультет: МИЭМ**

**Направление: *Компьютерная безопасность***

# Отчет по лабораторной работе №6 (Реализация структурных паттернов)

**по дисциплине**

**«Методы программирования»**

**Выполнил**

**Студент группы СКБ151**

**Михалицын Пётр**

**Москва, 2017.**

# Содержание

Описание проекта....2
Спецификация модулей....3
Реализация структурного паттерна Bridge и его UML-диаграмма....9
Выводы....11
Коды модулей....12
SetImpl.py....12
Set.py....19
reference.py....26

# Описание проекта

Проект заключался реализации структурного паттерна Bridge. В нашем случае он должен был создать интерфейс над тремя реализациями контейнера «множество». Переключение между которыми осуществлялось в зависимости от размера контейнера.

Во время изменения реализации все изменения и конвертации происходили неявно.

Основой для создания множеств послужили такие встроенные классы как list, set, tuple

Сам проект состоит из двух основных модулей и одного вспомогательного.

Основные модули:

1. SetImpl.py (содержит различные реализации множеств)
2. Set.py (содержит реализацию структурного паттерна)

Вспомогательный модуль reference.py содержит реализацию аналога ссылок из c++ в python. Был необходим для изменения неизменяемых объектов

Посмотреть полностью весь проект можно в репозитории https://github.com/lo1ol/brige-for-set

## Спецификация модулей

Модуль SetImpl.py содержит в себе абстрактный класс Set с 4 абстрактными методами и определением методов для осуществления основных операций над множествами, реализованных с помощью четырех базовых абстрактных методов. Кроме того, модуль содержит три конкретных реализации класса, каждая из которых наследуется от абстрактного класса Set и содержит определение 4ых методов, которые были абстрактными в базовом классе. Названия этих методов SetViaSet, SetViaList, SetViaTuple

Описывать в данном отчете подробно спецификацию на русском языке нету смысла в связи с их изобилием и экономией времени, поэтому снизу приведена их спецификация на английском языке, которая была дана мной же при написании программы.

```
class Set:
    """
    Abstract method Set which contain some virtual methods for other other
implementation of set
    and main methods, which uses overriding virtual methods
    """
    @abstractmethod
    def __init__(self, container=None):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """

    @abstractmethod
    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """

    @abstractmethod
    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """

    @abstractmethod
    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """

    @abstractmethod
    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """

    def empty(self):
        """
        check emptiness of set
        :return: True if set is empty, otherwise -- False
        """
```

```
def update(self, other):
    """
    Add other set to set
    :param other: other set
    :return: None
    """

def union(self, other):
    """
    Copy current set and return new set updated with other
    :param other: other set
    :return: New updated set
    """

def discard_update(self, other):
    """
    Remove elements, which contains other set in current set
    :param other: other set
    :return:
    """

def discard(self, other):
    """
    Copy current set and return new set discarded with other
    :param other: other set
    :return: New discarded set
    """

def intersection(self, other):
    """
    Make intersection with other set in current set
    :param other: other set
    :return: None
    """

def intersection_update(self, other):
    """
    Copy current set and return new set intersected with other
    :param other: other set
    :return: New Intersected set
    """

def difference_update(self, other):
    """
    Remove elements, which not contain other set from current set
    :param other: other set
    :return:None
    """

def difference(self, other):
    """
    Copy current set and return new set differenced with other
    :param other: other set
    :return: New Intersected set
    """

def __copy__(self):
    """
    Make a copy of current set
    :return: copy of current set
    """
```

```
    def __contains__(self, item):
        """
        Overriding operation in
        :param item: item for checking on containing
        :return: True if item in set, otherwise -- False
        """

    def __len__(self):
        """
        Method for getting len of set
        :return: len of set
        """

    def __and__(self, other):
        """
        Overriding of operation & (intersection)
        :param other: other set
        :return: new set
        """

    def __or__(self, other):
        """
        Overriding of operation | (union)
        :param other: other set
        :return: new set
        """

    def __iter__(self):
        """
        Get iterator for set
        :return: iterator for set
        """

    def __str__(self):
        """
        convert set to string
        :return: str
        """
```

Приведу лишь спецификацию самых интересных на мой взгляд, методов

Класс SetViaList

Содержит конструктор, который принимает один необязательный аргумент container, который должен уметь неявно преобразовываться в список. Данный список в дальнейшем будет использоваться в качестве контейнера для элементов множества (естественно перед инициализацией из него удаляются все элементы)

```
def __init__(self, container=()):
    """
    virtual constructor
    :param container: optional parameter for initial container
    """
```

В дальнейшем похожая спецификация конструктора будет иметь место в остальных классах

Самым интересным моментом модуля, на мой взгляд, является атрибут container класса SetViaTuple

Интересен он тем, что поскольку класс tuple в python является неизменяемым, то изменение множества в одном объекте не будет заметно в другом объекте, полученном путем перемещения

В связи с этим и был использован модуль reference с реализацией класса ref. Имеющий конструтор и два метода set_obj и get_obj.

Основная идея класса ref основывается на том, что класс list является изменяемым в python. Таким образом изменение первого элемента списка будет заметно в другом объекте. Поэтому конструктор принимал в качестве параметра некий объект, клал его в список, и позволял изменять первый элемент списка с помощью метода set_obj (изменение был видно всем, кто имеет одну и ту же сущность класса ref.

Чтобы все вышесказанное было наглядным, я привел снизу весь код данного модуля вместе со спецификацией, поскольку код является довольно небольшим

```
class ref():
    """
    Implementation of references in python
    """
    def __init__(self, obj):
        """
        Set object in reference
        :param obj:
        """
        self.obj = [obj]

    def set_obj(self, obj):
        """
        change current object in reference
        :param obj: New object
        :return: None
        """
        self.obj[0] = obj

    def get_obj(self):
        """
        get current object from reference
        :return: current object
        """
        return self.obj[0]
```

В связи с реализацией подомного класса, в класс SetViaTuple, была добавлена возможность сделать атрибут container неявно изменяемым. Были реализованы функции getter, setter и deleter. Которые неявно вызывались при обращении к атрибуту container и скрывали за собой обращение к объекту класса ref, инициализируемого нашим кортежем.

```
def _container_get(self):
    """
    Getter for container attribute
    :return:
    """

def _container_set(self, item):
    """
    setter for container attribute
    :param item:
    :return:
    """

def _container_del(self):
    """
    deleter for container attribute
    :return:
    """

container = property(_container_get, _container_set, _container_del)  #
Property for make tuple changable object in python
```

Модуль Set.py содержит реализацию класса Set, который является интерфейсом над тремя вышеизложенными реализациями множества.

Данный интерфейс действует по следующему правилу, если кол-во элементов в множестве (n):

0 <= n <= 10, то текущей реализацией будет SetViaTuple

10 < n <= 100, то текущей реализацией будет SetViaList

100 < n, то текущей реализацией будет SetViaSet

Данный класс содержит конструктор, свойство set, метод _transform, а также методы, которые поддерживают все реализации множеств.

Спецификация всех методов для реализации операции над множествами будут опущена, про них можно сказать только одно, что если внутри них вызываются методы приводящие к преобразованиям размера множества, то вызывается _transform, который может изменить текущую реализацию множества.

Метод _transform, узнает текущую реализацию и если видит, что текущая реализация является не соответствующей текущему размеру множества, то происходит конвертация атрибута set, к соответствующей реализации

```
def _transform(self):
    """
    This method called after each changes of size in set,
    If changes of type are necessary, they are made
    :return:
    """
```

Конструктор устанавливает соответствующую реализацию множества в зависимости от размера переданного ему размера. Важно заметить, что переданный ему контейнер должен поддерживать итерирование по его элементам.

```
def __init__(self, container=()):
    """
    Constructor of Set bridge
    :param container: initialize set
    """
```

Свойство set стало именно свойством, а не просто атрибутом, для того чтобы изменение данного контейнера стали видны всем другим элементам, имеющим ссылку на сущность класса Set. Например, такие изменения могли бы возникнуть при изменении текущей реализации множества.

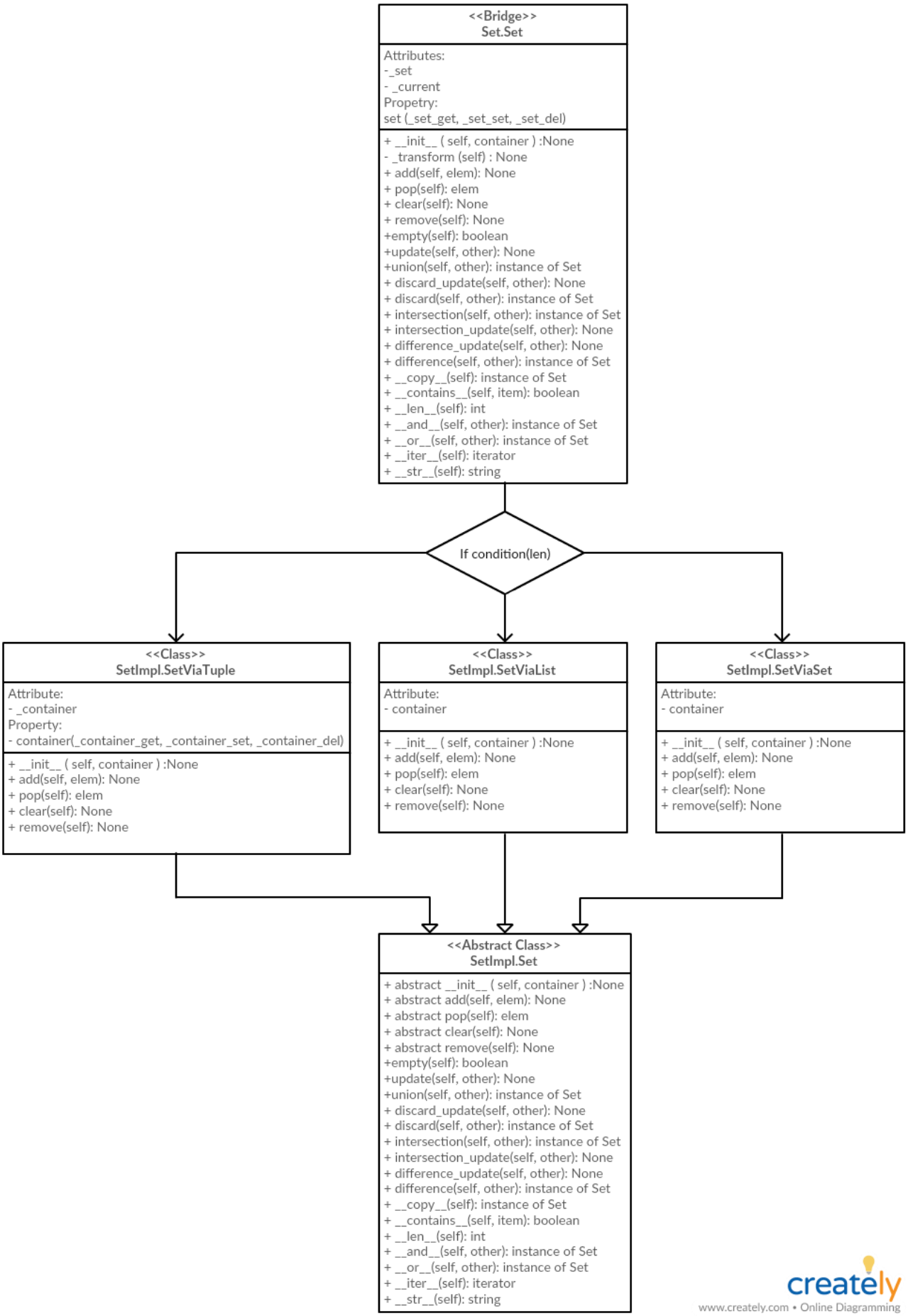
```
def _set_get(self):
    """
    getter for attribute set
    :return:
    """

def _set_set(self, item):
    """
    setter for attribute set
    :return:
    """

def _set_del(self):
    """
    deleter for attribute set
    :return:
    """

set = property(_set_get, _set_set, _set_del)  # Property set uses for make
unchangeable items in python changeable.
# Furthermore it makes changes visible after transform in other set (which
has same reference on object)
```

## Реализация структурного паттерна Bridge и его UML-диаграмма

По сути реализация данного является весьма очевидной и вполне понятной при прочтении спецификации.

Тем не менее сейчас будет описана структура этого паттерна.

Класс Set использует три реализации множества, они изменяются в зависимости от размера множества. Сам класс хранит только ссылку на соответствующую реализацию и отдельный атрибут сообщающий текущее состояние реализации (по сути можно было обойтись без него, но его использование сокращает код программы). Кроме того, Set имеет реализацию всех методов реализации, которые просто вызывают соответствующие методы свойства set. После операций, в которых возможно изменение размера множества вызывался метод _transform, который по необходимости может изменить текущую реализацию множества.


<<Bridge>>
Set.Set
Attributes:
-_set
- _current
Propetry:
set (_set_get, _set_set, _set_del)
+ __init__ ( self, container ) :None
- _transform (self) : None
+ add(self, elem): None
+ pop(self): elem
+ clear(self): None
+ remove(self): None
+empty(self): boolean
+update(self, other): None
+union(self, other): instance of Set
+ discard_update(self, other): None
+ discard(self, other): instance of Set
+ intersection(self, other): instance of Set
+ intersection_update(self, other): None
+ difference_update(self, other): None
+ difference(self, other): instance of Set
+ __copy__(self): instance of Set
+ __contains__(self, item): boolean
+ __len__(self): int
+ __and__(self, other): instance of Set
+ __or__(self, other): instance of Set
+ __iter__(self): iterator
+ __str__(self): string
If condition(len)
<<Class>>
SetImpl.SetViaTuple
Attribute:
- _container
Property:
- container(_container_get, _container_set, _container_del)
+ __init__ ( self, container ) :None
+ add(self, elem): None
+ pop(self): elem
+ clear(self): None
+ remove(self): None
<<Class>>
SetImpl.SetViaList
Attribute:
- container
+ __init__ ( self, container ) :None
+ add(self, elem): None
+ pop(self): elem
+ clear(self): None
+ remove(self): None
<<Class>>
SetImpl.SetViaSet
Attribute:
- container
+ __init__ ( self, container ) :None
+ add(self, elem): None
+ pop(self): elem
+ clear(self): None
+ remove(self): None
<<Abstract Class>>
SetImpl.Set
+ abstract __init__ ( self, container ) :None
+ abstract add(self, elem): None
+ abstract pop(self): elem
+ abstract clear(self): None
+ abstract remove(self): None
+empty(self): boolean
+update(self, other): None
+union(self, other): instance of Set
+ discard_update(self, other): None
+ discard(self, other): instance of Set
+ intersection(self, other): instance of Set
+ intersection_update(self, other): None
+ difference_update(self, other): None
+ difference(self, other): instance of Set
+ __copy__(self): instance of Set
+ __contains__(self, item): boolean
+ __len__(self): int
+ __and__(self, other): instance of Set
+ __or__(self, other): instance of Set
+ __iter__(self): iterator
+ __str__(self): string
creately
www.creately.com • Online Diagramming

# Выводы

Паттерн Bridge является весьма хорошим решением, когда нужно отделить интерфейс класса от его реализации. Более того, можно сделать создать такой шаблон, который будет менять свою реализацию в зависимости от каких-либо условий, что является весьма полезной вещью при разработке сложноорганизованных структур.

## Коды модулей

SetImpl.py

```
from copy import copy
from abc import abstractmethod
from reference import ref


class Set:
    """
    Abstract method Set which contain some virtual methods for other other
implementation of set
    and main methods, which uses overriding virtual methods
    """
    @abstractmethod
    def __init__(self, container=None):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        pass

    @abstractmethod
    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        pass

    @abstractmethod
    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        pass

    @abstractmethod
    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        pass

    @abstractmethod
    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        pass

    def empty(self):
        """
        check emptiness of set
        :return: True if set is empty, otherwise -- False
        """
        if not self:
            return True
        else:
```

```
            return False

    def update(self, other):
        """
        Add other set to set
        :param other: other set
        :return: None
        """
        for elem in other:
            self.add(elem)

    def union(self, other):
        """
        Copy current set and return new set updated with other
        :param other: other set
        :return: New updated set
        """
        new_set = copy(self)
        new_set.update(other)
        return new_set

    def discard_update(self, other):
        """
        Remove elements, which contains other set in current set
        :param other: other set
        :return:
        """
        for elem in other:
            if elem in self:
                self.remove(elem)

    def discard(self, other):
        """
        Copy current set and return new set discarded with other
        :param other: other set
        :return: New discarded set
        """
        new_set = copy(self)
        new_set.discard_update(other)
        return new_set

    def intersection(self, other):
        """
        Make intersection with other set in current set
        :param other: other set
        :return: None
        """
        new_set = copy(self)
        new_set.intersection_update(other)
        return new_set

    def intersection_update(self, other):
        """
        Copy current set and return new set intersected with other
        :param other: other set
        :return: New Intersected set
        """
        cp = copy(self)
        for elem in cp:
            if elem not in other:
                self.remove(elem)

    def difference_update(self, other):
```

```
        """
        Remove elements, which not contain other set from current set
        :param other: other set
        :return:None
        """
        self.__init__(self.discard(other) | other.discard(self))

    def difference(self, other):
        """
        Copy current set and return new set differenced with other
        :param other: other set
        :return: New Intersected set
        """
        new_set = copy(self)
        new_set.difference_update(other)
        return new_set

    def __copy__(self):
        """
        Make a copy of current set
        :return: copy of current set
        """
        return self.__class__(self.container)

    def __contains__(self, item):
        """
        Overriding operation in
        :param item: item for checking on containing
        :return: True if item in set, otherwise -- False
        """
        if item in self.container:
            return True
        else:
            return False

    def __len__(self):
        """
        Method for getting len of set
        :return: len of set
        """
        return len(self.container)

    def __and__(self, other):
        """
        Overriding of operation & (intersection)
        :param other: other set
        :return: new set
        """
        return self.intersection(other)

    def __or__(self, other):
        """
        Overriding of operation | (union)
        :param other: other set
        :return: new set
        """
        return self.union(other)

    def __iter__(self):
        """
        Get iterator for set
        :return: iterator for set
        """
```

```
        return self.container.__iter__()

    def __str__(self):
        """
        convert set to string
        :return: str
        """
        string = str(self.container)
        return "{" + string[1:-1] + "}"


class SetViaList(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        Set.__init__(self, container)
        self.container = list(set(container))

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass
        else:
            self.container.append(elem)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        return self.container.pop()

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = []

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        return self.container.remove(item)


class SetViaSet(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
```

```
        :param container: optional parameter for initial container
        """
        Set.__init__(self, container)
        self.container = set(container)

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass

        else:
            self.container.add(elem)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        return self.container.pop()

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = set()

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        return self.container.remove(item)


class SetViaTuple(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        Set.__init__(self, container)
        self.container = tuple(set(container))

    def _container_get(self):
        """
        Getter for container attribute
        :return:
        """
        return self._container.get_obj()

    def _container_set(self, item):
        """
        setter for container attribute
        :param item:
        :return:
```

```
        """
        if not "_set" in self.__dict__:
            self._container = ref(item)
        else:
            self._container.set_obj(item)

    def _container_del(self):
        """
        deleter for container attribute
        :return:
        """
        if not "_container" in self.__dict__:
            return
        else:
            self.__dict__.pop("_container")

    container = property(_container_get, _container_set, _container_del)  #
Property for make tuple changable object in python

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass
        else:
            self.container = self.container + (elem,)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        elem = self.container[-1]
        self.container = self.container[:-1]
        return elem

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = tuple()

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        index = self.container.index(item)
        self.container = self.container[:index]+self.container[index+1:]


if __name__ == "__main__":
    x = SetViaList()
    x.add(12)
    x.add(13)
    y = SetViaSet(x)
    y.add(14)
    print("x :%s, y: %s" % (x, y))
    y.add(12)
```

```
    y = x
    y.add(1)
    print("x :%s, y: %s" % (x, y))
    print(y.pop())
    print(x)
    x = SetViaTuple({1, 2, 3, 4, 4})
    x.add(10)
    print(x)
    x.pop()
    print(x)
    x.clear()
    print(x)
    x.update(SetViaSet((1, 3, 10, 30)))
    x.remove(3)
    print(set(x))
    print(x)
    print(len(x))
```

Set.py

```
from copy import copy
from abc import abstractmethod
from reference import ref


class Set:
    """
    Abstract method Set which contain some virtual methods for other other
implementation of set
    and main methods, which uses overriding virtual methods
    """
    @abstractmethod
    def __init__(self, container=None):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        pass

    @abstractmethod
    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        pass

    @abstractmethod
    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        pass

    @abstractmethod
    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        pass

    @abstractmethod
    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        pass

    def empty(self):
        """
        check emptiness of set
        :return: True if set is empty, otherwise -- False
        """
        if not self:
            return True
        else:
            return False
```

```
    def update(self, other):
        """
        Add other set to set
        :param other: other set
        :return: None
        """
        for elem in other:
            self.add(elem)

    def union(self, other):
        """
        Copy current set and return new set updated with other
        :param other: other set
        :return: New updated set
        """
        new_set = copy(self)
        new_set.update(other)
        return new_set

    def discard_update(self, other):
        """
        Remove elements, which contains other set in current set
        :param other: other set
        :return:
        """
        for elem in other:
            if elem in self:
                self.remove(elem)

    def discard(self, other):
        """
        Copy current set and return new set discarded with other
        :param other: other set
        :return: New discarded set
        """
        new_set = copy(self)
        new_set.discard_update(other)
        return new_set

    def intersection(self, other):
        """
        Make intersection with other set in current set
        :param other: other set
        :return: None
        """
        new_set = copy(self)
        new_set.intersection_update(other)
        return new_set

    def intersection_update(self, other):
        """
        Copy current set and return new set intersected with other
        :param other: other set
        :return: New Intersected set
        """
        cp = copy(self)
        for elem in cp:
            if elem not in other:
                self.remove(elem)

    def difference_update(self, other):
        """
        Remove elements, which not contain other set from current set
```

```
        :param other: other set
        :return:None
        """
        self.__init__(self.discard(other) | other.discard(self))

    def difference(self, other):
        """
        Copy current set and return new set differenced with other
        :param other: other set
        :return: New Intersected set
        """
        new_set = copy(self)
        new_set.difference_update(other)
        return new_set

    def __copy__(self):
        """
        Make a copy of current set
        :return: copy of current set
        """
        return self.__class__(self.container)

    def __contains__(self, item):
        """
        Overriding operation in
        :param item: item for checking on containing
        :return: True if item in set, otherwise -- False
        """
        if item in self.container:
            return True
        else:
            return False

    def __len__(self):
        """
        Method for getting len of set
        :return: len of set
        """
        return len(self.container)

    def __and__(self, other):
        """
        Overriding of operation & (intersection)
        :param other: other set
        :return: new set
        """
        return self.intersection(other)

    def __or__(self, other):
        """
        Overriding of operation | (union)
        :param other: other set
        :return: new set
        """
        return self.union(other)

    def __iter__(self):
        """
        Get iterator for set
        :return: iterator for set
        """
        return self.container.__iter__()
```

```
    def __str__(self):
        """
        convert set to string
        :return: str
        """
        string = str(self.container)
        return "{" + string[1:-1] + "}"


class SetViaList(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        Set.__init__(self, container)
        self.container = list(set(container))

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass
        else:
            self.container.append(elem)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        return self.container.pop()

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = []

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        return self.container.remove(item)


class SetViaSet(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
```

```
        Set.__init__(self, container)
        self.container = set(container)

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass

        else:
            self.container.add(elem)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        return self.container.pop()

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = set()

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        return self.container.remove(item)


class SetViaTuple(Set):
    """
    Implementation of set via list
    """
    def __init__(self, container=()):
        """
        virtual constructor
        :param container: optional parameter for initial container
        """
        Set.__init__(self, container)
        self.container = tuple(set(container))

    def _container_get(self):
        """
        Getter for container attribute
        :return:
        """
        return self._container.get_obj()

    def _container_set(self, item):
        """
        setter for container attribute
        :param item:
        :return:
        """
        if not "_set" in self.__dict__:
```

```
            self._container = ref(item)
        else:
            self._container.set_obj(item)

    def _container_del(self):
        """
        deleter for container attribute
        :return:
        """
        if not "_container" in self.__dict__:
            return
        else:
            self.__dict__.pop("_container")

    container = property(_container_get, _container_set, _container_del)  #
Property for make tuple changable object in python

    def add(self, elem):
        """
        virtual method implements operation add to set
        :return: None
        """
        if elem in self.container:
            pass
        else:
            self.container = self.container + (elem,)

    def pop(self):
        """
        virtual method implements pop random element from set
        :return: popped element
        """
        elem = self.container[-1]
        self.container = self.container[:-1]
        return elem

    def clear(self):
        """
        virtual method implements clearing set
        :return: None
        """
        self.container = tuple()

    def remove(self, item):
        """
        virtual method for removing item from set, if no item -- exception
        :param item: item for removing
        :return: None
        """
        index = self.container.index(item)
        self.container = self.container[:index]+self.container[index+1:]


if __name__ == "__main__":
    x = SetViaList()
    x.add(12)
    x.add(13)
    y = SetViaSet(x)
    y.add(14)
    print("x :%s, y: %s" % (x, y))
    y.add(12)
    y = x
    y.add(1)
```

```
    print("x :%s, y: %s" % (x, y))
    print(y.pop())
    print(x)
    x = SetViaTuple({1, 2, 3, 4, 4})
    x.add(10)
    print(x)
    x.pop()
    print(x)
    x.clear()
    print(x)
    x.update(SetViaSet((1, 3, 10, 30)))
    x.remove(3)
    print(set(x))
    print(x)
    print(len(x))
```

reference.py

```
class ref():
    """
    Implementation of references in python
    """
    def __init__(self, obj):
        """
        Set object in reference
        :param obj:
        """
        self.obj = [obj]

    def set_obj(self, obj):
        """
        change current object in reference
        :param obj: New object
        :return: None
        """
        self.obj[0] = obj

    def get_obj(self):
        """
        get current object from reference
        :return: current object
        """
        return self.obj[0]
```