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 порождающих паттернов)

по дисциплине

«Методы программирования»

Выполнил

Студент группы СКБ151

Михалицын Пётр

Москва, 2017.

Содержание

Описание проекта

Проект заключался реализации порождающего паттерна Singleton на классе Logger, осуществляющего логирование событий, происходящих в системе. Система должна была иметь несколько видов событий с поддержкой комментариев к ним, сохранять последние 10 событий и при необходимости выводить их.

Исключительно из «спортивного» интереса в проект была добавлена поддержка хронологии событий, синхронизация Loggera с файловой системой и непосредственный пример использования данного объекта в многопоточной программе.

Проект содержит в себе два модуля: реализация Loggera SingletonImpl.py и модуль содержащий проект с примером использования данного Loggera example_of_use.py

Из соображений экономии времени и пространства данного отсчета детали модуля, содержавшего пример использования Loggera, рассматриваться не будут. В конце будет приведен пример работы программы и текст самой программы

Посмотреть полностью весь проект можно в репозитории
https://github.com/lo1ol/SingletonLog

Спецификация модулей

Модуль SingletonImpl.py содержит реализацию классов Event, Logger, функции singleton, использовавшаяся в качестве декоратора к классу Logger, для придания свойства Singleton любому классу; и трех констант для определения уровня сообщения (обычное, заметка и ошибка)

Класс Event содержит конструктор со специальным методом преобразования в строку.

Конструктор принимает один обязательный параметр type, определяющий тип события, и один опциональный параметр message, прикрепляющий сообщение к записи в логфайле (по умолчанию пустая строка)

```
def __init__(self, type, message=''):
    """
    Make instance of event
    :param type: identify type of event
    :param message: optional param which contain message attached to event
    """
```

Специальный метод преобразования в строку преобразуем объект в строку формата, указанного в спецификации

```
def __str__(self):
    """
    special method for converting object to str
    :return: string in format
    "
    time type
    Comment:
    ...
    "
    """
```

Класс Logger содержит один конструктор, метод event и get_log

Конструктор имеет один необязательный аргумент – путь к файлу в файловой системе, с которым будет происходить синхронизация при поступлении событий (по умолчанию синхронизация отключена).

```
def __init__(self, file=None):
    """
    Initialize Logger instance
    :param file: optional file for synchronisation log file with file system
(if param absent -- doesn't make sync)
    """
```

Метод event принимает тип событие и опционально сообщение к нему. Данное событие создается и заносится в массив последних 10 событий. В случае включенной синхронизации, наступление данного события отображается в файловой системе

```
def event(self, type, message=''):
    """
    Make record in log file with type, time of recording and attached message
    If sync is True -- make events visible in file system
    :param type: type of event
    :param message: attached message to event (optional)
    :return:
    """
    event
```

Метод get_log возвращает 10 последних событий, произошедших в системе в виде отформатированных строк, если параметр format имел значение, которое неявно преобразовывалось к False, то возвращался массив, содержащий объекты типа Event

```
def get_log(self, format=True):
    """
    Method for get list of 10 last records in log
    :param format: if format is True (default) return formatted log, else list
of events
    :return: list of last 10 events in log
    """
```

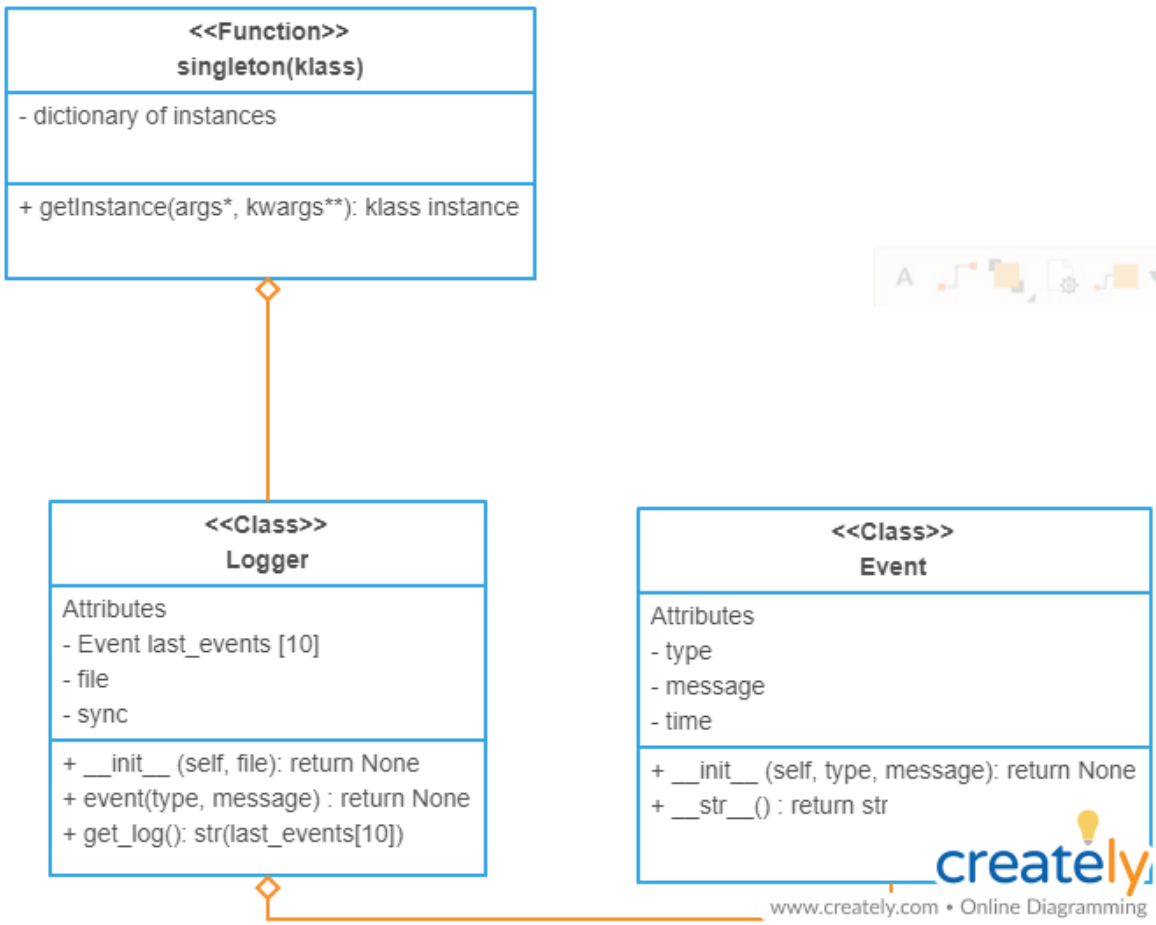
Функция singleton используется в качестве декоратора, и позволяет классу получить свойство Singleton. Реализация данного класса будет рассмотрена ниже

```
def singleton(klass):
    """
    Function which uses for creating singleton object
    Contain nested function, which returned
    dictionary instances which contain conformity between classes and their
single instances
    this object is kept because it is in the enclosing function scope
    :param klass:
    :return:
    """
```

Реализация порождающего паттерна Singleton и его UML-диаграмма

Паттерн Singleton реализован в питоне с использованием декораторов, Сама по себе функция singleton содержит в себе вложенную функцию и словарь instances, находящемся в области видимости вложенных функции, за счет чего ом может сохранять свои значения во вложенной функции от вызова к вызову.

Сам по себе словарь осуществляет отображение между классом и его сущностью. В случае, если при вызове конструктора класса (неявном вызове декоратора), оказывается, что данный класс уже есть в словаре, то возвращается объект данного класса соответствующий ему, иначе создается сущность данного класса, делается пометка в словаре и возвращается только что сконструированная функция с переданными ей параметрами (которые на самом деле принял декоратор).



Выводы

Как оказалось, паттерн Singleton является весьма приятной конструкцией для создания объектов в программе, которые должны быть одни на всю программу.

Таким образом при реализации многопоточное программы, не пришлось заморачиваться по поводу передачи объекта Logger каждому потоку. Ведь каждый поток мог просто создать себе по одному объекту Logger (на самом деле один и тот же у каждого потока) и записывать туда информацию, задумываясь только о синхронизации доступа к нему, и не боясь потерять ссылку или указатель на объект, ведь его всегда можно получить в любое время.

Более того, при применении других паттернов возможно будет и абстрагироваться от синхронизации доступа к дог файлу (например паттерн proxy)

Пример лог файла

Tue Jan 30 03:51:18 2018 normal

Comment:

Oh my god

It's so awesome program

Tue Jan 30 03:51:20 2018 error

Comment:

name 'unknown_var' is not defined

Tue Jan 30 03:51:20 2018 remark

Comment:

If anyone wanted ter find out some stuff,

all they’d have ter do would be ter follow the spiders.

That’d lead ’em right!

That’s all I’m sayin

Tue Jan 30 03:51:22 2018 error

Comment:

unsupported operand type(s) for /: 'str' and 'int'

Tue Jan 30 03:51:25 2018 error

Comment:

list assignment index out of range

Tue Jan 30 03:51:26 2018 remark

Comment:

If anyone wanted ter find out some stuff,

all they’d have ter do would be ter follow the spiders.

That’d lead ’em right!

That’s all I’m sayin

Tue Jan 30 03:51:26 2018 error

Comment:

unsupported operand type(s) for /: 'str' and 'int'

Tue Jan 30 03:51:28 2018 error

Comment:

unsupported operand type(s) for /: 'str' and 'int'

Tue Jan 30 03:51:33 2018 normal

Comment:

Oh my god

It's so awesome program

Коды модулей

SingletonImpl.py

```
from time import ctime
from copy import copy


"""
Module contain:
3 constants or identification type of events (normal 0, remark 1, error 2)
class Event for containing type of events and their properties (time, type,
message) with method of converting to str
class Logger, which implement Logger in program
and function singleton. It is uses like decorator for implementation of
singleton object
"""

normal = 0
remark = 1
error = 2


class Event:
    """
    Class contain constructor
    and special method for converting object to str
    attributes of time, type and message of event
    """
    def __init__(self, type, message=''):
        """
        Make instance of event
        :param type: identify type of event
        :param message: optional param which contain message attached to event
        """
        self.time = ctime()
        self.type = type
        self.message = message

    def __str__(self):
        """
        special method for converting object to str
        :return: string in format
        "
        time type
        Comment:
        ...
        "
        """
        format = """{0} {1}\nComment:\n{2}\n"""
        if self.type == 0:
            type = "normal"
        elif self.type == 1:
            type = "remark"
        elif self.type == 2:
            type = "error"
        else:
            type = "unknown event %d" % self.type

        return format.format(self.time, type, self.message)


def singleton(klass):
    """
```

```
    Function which uses for creating singleton object
    Contain nested function, which returned
    dictionary instances which contain conformity between classes and their
single instances
    this object is kept because it is in the enclosing function scope
    :param klass:
    :return:
    """
    instances = {}

    def getInstance(*args, **kwargs):
        nonlocal instances
        if klass in instances:
            return instances[klass]
        else:
            inst = klass(*args, **kwargs)
            instances[klass] = inst
            return inst

    return getInstance


@singleton
class Logger:
    """
    Class decorated by singleton function
    Contain:
    1 construction
    method event for make note in log
    method get_log for getting last 10 records in log
    """
    def __init__(self, file=None):
        """
        Initialize Logger instance
        :param file: optional file for synchronisation log file with file
system (if param absent -- doesn't make sync)
        """
        if file:
            self.sync = True
            self.file = file
        else:
            self.sync = False

        self.last_events = [None for _ in range(10)]
        self.last = 0

    def event(self, type, message=''):
        """
        Make record in log file with type, time of recording and attached
message
        If sync is True -- make events visible in file system
        :param type: type of event
        :param message: attached message to event (optional)
        :return:
        """
        event = Event(type, message)
        self.last_events[self.last] = event
        self.last = (self.last + 1) % 10
        if self.sync:
            file = open(self.file, 'w')
            file.write('\n'.join(self.get_log()))
            file.close()
```

```
    def get_log(self, format=True):
        """
        Method for get list of 10 last records in log
        :param format: if format is True (default) return formatted log, else
list of events
        :return: list of last 10 events in log
        """
        if format:
            return map(str, filter((lambda x: x), self.last_events[self.last:] +
self.last_events[:self.last]))
        else:
            return copy(self.last_events)


if __name__ == "__main__":
    inst = Logger("log.txt")
    inst.event(1)
    inst.event(2, "com1")
    inst.event(0, "com2")
    inst2 = Logger()
    inst2.event(0, "com3")
    print('\n'.join(inst.get_log()), "\n")
    print('\n'.join(inst2.get_log()))
```

example_of_use.py

```
from SingletonImpl import Logger, error, normal, remark
from random import randint
from threading import Thread, Semaphore
from time import sleep, clock
import os
import signal

stop = False  # To stop threads
sem = Semaphore(1)  # Semaphore for sync access to Logger
semstop = Semaphore(1)  # Semaphore for sync access to stop


def show_log():
    """
    Function showing log in console (last 10 records in log)
    :return: None
    """
    if os.name == 'nt':
        os.system('cls')
    elif os.name == 'posix':
        os.system('clear')
    sem.acquire()
    loglist = Logger().get_log()
    sem.release()
    print('\n'.join(loglist))


def stopper(signum, frame):
    """
    Handler for keyboard interruption to stop threads
    :param signum:
    :param frame:
    :return:
    """
    global stop
    semstop.acquire()
    stop = True
    semstop.release()


def error_maker():
    """
    Function which making error events
    :return:
    """
    global stop
    log = Logger('log.txt')
    x = [i for i in range(10)]  # list with maximum index 9!
    semstop.acquire()
    while not stop:
        semstop.release()
        sleep(0.5)
        try:
            error_type = randint(0, 2)  # choice of error
            if error_type == 0:
                x[randint(0, 10)] = randint(0, 10)  # index out of range if 10

            elif error type ==1:
                if randint(0, 10) == 0:
                    "123"/3  # unsupported operation for this types
```

```
            elif error_type==2:
                if randint(0, 10) == 0:
                    print(unknown_var)  # unknown var

        except IndexError as err:  # three exceptions for make records in lof
            sem.acquire()
            log.event(error, err)
            sem.release()

        except TypeError as err:
            sem.acquire()
            log.event(error, err)
            sem.release()

        except NameError as err:
            sem.acquire()
            log.event(error, err)
            sem.release()

        semstop.acquire()
    semstop.release()


def normal_maker():
    """
    Function which making normal events
    :return:
    """
    global stop
    log = Logger("log.txt")
    semstop.acquire()
    while not stop:
        semstop.release()
        sleep(0.5)
        if randint(0, 10) == 0:
            sem.acquire()
            log.event(normal, "Oh my god\nIt's so awesome program")
            sem.release()

        semstop.acquire()
    semstop.release()


def remark_maker():
    """
    Function which making remark events
    :return:
    """
    global stop
    log = Logger("log.txt")
    semstop.acquire()
    while not stop:
        semstop.release()
        sleep(0.5)
        if randint(0, 10) == 0:
            sem.acquire()
            log.event(remark, "If anyone wanted ter find out some stuff,\nall
they'd have ter do would be ter follow the spiders.\nThat'd lead 'em
right!\nThat's all I'm sayin")
            sem.release()

        semstop.acquire()
```

```
    semstop.release()


if __name__ == "__main__":
    signal.signal(signal.SIGINT, stopper)  # Set handler for keyboard
interrupt

    err_thread = Thread(target=error_maker)
    norm_thread = Thread(target=normal_maker)
    rem_thread = Thread(target=remark_maker)
    err_thread.start()
    norm_thread.start()
    rem_thread.start()

    show_log()
    last_log_show = clock()
    semstop.acquire()
    while not stop:  # Refresh screen with last logs, until stop == True
        semstop.release()
        sleep(0.1)
        if clock() - last_log_show > 1:
            show_log()
            last_log_show = clock()
        semstop.acquire()
    semstop.release()

    err_thread.join()
    norm_thread.join()
    rem_thread.join()
    Logger().event(3, "Ho-ho-ho")  # Make last unknown event

    show_log()  # show last log
```