Object Oriented Programming

Bas Terwijn

February 16, 2026

1 Introductie

Dit document behandelt principes van Object Oriented Programming (OOP).

1.1 Doelen

Bekend raken met:

- het schrijven van eigen classes in Python
- het gebruik van dunder (double underscore) methoden
- het gebruik van abstractie, encapsulation en information hidding
- het decouplen van code door gebruik van higher-order-functies

2 Opdracht Alarm Clock

Onderstaande 14 Python opgaven helpen om op een object-georiënteerde (OO) manier te leren programmeren met Python. Bij object-oriented programming maken we gebruik van classes om onze eigen types te maken.

2.1 Doctest

In deze opdracht maken we gebruik van doctests om functies te testen. Doctests zijn tests in de vorm van voorbeelden die in docstring staan achter `>>>`. In een doctest wordt eerst de functie aangeroepen en op de eerst

volgende regel zonder `>>>` staat dan het verwachte resultaat. We zien hier drie simpele voorbeelden in de `add_numbers()` functie die de som van twee getallen zou moeten returnen.

assignments/doctest_example.py

```
def add_numbers(a, b):
    """ Return the addition of 'a' and 'b'
    >>> add_numbers(10, 5)
    15
    >>> add_numbers(10, -5)
    5
    >>> add_numbers(-10, 5)
    -5
    """
    return a + b + 1
```

Met dit commando in de `assignments` directory kunnen we de doctests uitvoeren:

```
$ python -m doctest doctest_example.py
**********************************************************************
File "doctest_example.py", line 4, in doctest_example.add_numbers
Failed example:
    add_numbers(10, 5)
Expected:
    15
Got:
    16
**********************************************************************
File "doctest_example.py", line 6, in doctest_example.add_numbers
Failed example:
    add_numbers(10, -5)
Expected:
    5
Got:
    6
**********************************************************************
File "doctest_example.py", line 8, in doctest_example.add_numbers
Failed example:
    add_numbers(-10, 5)
Expected:
    -5
Got:
    -4
**********************************************************************
1 item had failures:
   3 of   3 in doctest_example.add_numbers
***Test Failed*** 3 failures.
```

We zien dan dat alle drie de tests mislukken omdat de implementatie van de functie niet klopt. Pas de implementatie van de functie aan zodat de tests wel slagen. Als alle tests slagen dan geeft dit commando geen output meer. Als je ook de tests wilt zien die wel slagen, dan kun je het commando met de `-v` (verbose) optie uitvoeren:

```
$ python -m doctest assignments/doctest_example.py -v
```

Gebruik steeds doctests om je implementaties in de onderstaande opgaven te testen.

3 Time

In deze opdracht gaan we onze eigen alarmklok programmeren. Hiervoor schrijven we eerst een Time class om daarmee op een eenvoudige manier met tijd-waarden om te kunnen gaan. Schrijf voor onderstaande opgaven je

eigen implementatie, maak **geen** gebruik van al bestaande code voor tijd-waarden. Het maken van onderstaande opgave 1 t/m 5 resulteert in de Time class zoals weergegeven in figuur 1.

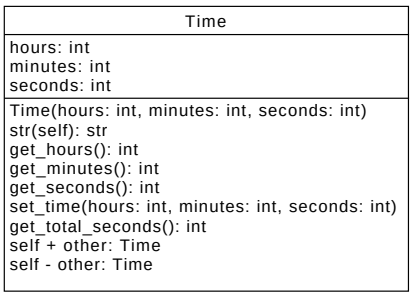

Figure 1: UML class diagram van de Time class.

3.1 opgave1

Implementeer in file `assignments/alarm_clock.py` de methoden van onderstaande class `Time` zodat we objecten van type Time kunnen aanmaken. Een `Time` object heeft 'hours', 'minutes' en 'seconds' en heeft de string representatie van een digitale klok, bijvoorbeeld 21:05:02 voor twee seconden na vijf minuten over negen 's avonds. Voor deze string representatie kun je gebruik maken van f-strings formatting voor "leading zeros" met bijvoorbeeld `f"{123:07}"`.

Zoals we al eerder zagen zijn de methoden met een dubbele underscores "__" speciale methoden en worden dunder methods (soms ook wel magic methods) genoemd. Dunder methoden maken een bepaalde interactie met de class mogelijk:

- __init__() zorgt dat we een object kunnen aanmaken, bijvoorbeeld met:

```
t = Time(8,5,30)
```

- __repr__() zorgt dat een `Time` object naar een string kan worden geconverteerd, bijvoorbeeld met:

```
str(Time(8,5,30))
print(Time(8,5,30))
```

assignments/alarm_clock.py

```
HOURS_IN_DAY = 24
MINUTES_IN_HOUR = 60
SECONDS_IN_MINUTE = 60


class Time:
    """ Represents a time of day."""

    def __init__(self, hours, minutes, seconds):
        """ Initialises a Time object with integers 'hours', 'minutes' and
        'seconds.
        >>> t = Time(18, 30, 0)
        """

    def __repr__(self):
        """ Returns the string representation of a Time object.
        >>> print( Time(8,5,30) )
        08:05:30
        """
```

```
    def get_hours(self):
        """ Returns the hours of the Time object.
        >>> Time(23,0,0).get_hours()
        23
        """

    def get_minutes(self):
        """ Returns the minutes of the Time object.
        >>> Time(0,59,0).get_minutes()
        59
        """

    def get_seconds(self):
        """ Returns the seconds of the Time object.
        >>> Time(0,0,59).get_seconds()
        59
        """
```

Deze en andere code is steeds te vinden in de `assignments` directory. Test steeds je implementatie met gebruik van de doctests en door eventueel een eigen main functie toe te voegen zoals bijvoorbeeld:

```
def main():
    t1 = Time(9, 30, 5)
    print("t1:", t1)
    print("seconds:", t1.get_seconds())

if __name__ == "__main__": # keep this at the bottom of the file
    main()
```

3.2 opgave2

Voeg onderstaande methode `set_time()` toe aan de Time class om de tijd van een al bestaand `Time` object aan te passen. Voor uw gemak kunt u deze code kopiëren uit file "assignments/opgave2.py" zoals steeds met witte lettertjes in de zwarte kop boven een code blok wordt aangegeven.

Zorg daarbij dat de instance variabelen van een `Time` object altijd geldige waarden krijgen, ook als er bijvoorbeeld een negatief aantal seconden wordt geven. Zo zou bijvoorbeeld `Time(0,0,90)` moeten leiden tot een 00:01:30 tijd, en `Time(0,1,-30)` tot een 00:00:30 tijd. Gebruik hiervoor bijvoorbeeld de divmod() functie. Gebruik deze `set_time()` methode vervolgens ook in de `__init__()` methode zodat ook bij het aanmaken van een nieuw Time object ongeldige waarden automatisch worden gecorrigeerd.

assignments/opgave2.py

```
    def set_time(self, hours, minutes, seconds):
        """ Sets the time of the Time object to 'hours', 'minutes',
        and 'seconds' making sure the values are in valid range:
          hours:   [0, HOURS_IN_DAY)
          minutes: [0, MINUTES_IN_HOUR)
          seconds: [0, SECONDS_IN_MINUTE)
        >>> time = Time(0, 0, 0)
        >>> time.set_time(0, 0, 90)
        >>> print(time)
        00:01:30
        >>> time.set_time(0, 0, 3600)
        >>> print(time)
        01:00:00
        >>> time.set_time(0, 0, -1)
        >>> print(time)
        23:59:59
```

```
        >>> time.set_time(10, -121, 0)
        >>> print(time)
        07:59:00
        >>> time.set_time(-50, 0, 0)
        >>> print(time)
        22:00:00
        >>> print(Time(10, -120, -150)) # __init__() test
        07:57:30
        """
```

3.3 opgave3

Voeg ook methode `get_total_seconds()` toe welke het aantal seconden sinds tijdstip 00:00:00 berekent en returned. Schrijf de implementatie.

assignments/opgave3.py

```
    def get_total_seconds(self):
        """ Returns the number of seconds since time 00:00:00.
        >>> Time(0,0,1).get_total_seconds()
        1
        >>> Time(0,1,0).get_total_seconds()
        60
        >>> Time(1,0,0).get_total_seconds()
        3600
        >>> Time(13,30,5).get_total_seconds()
        48605
        """
```

3.4 opgave4

We kunnen het gebruik van de "+" operator op objecten van een class mogelijk maken door de dunder method `__add__()` toe te voegen. Voeg deze methode toe en ook de dunder method `__sub__()` die het gebruik van de "-" operator mogelijk maakt. Schrijf de implementaties.

assignments/opgave4.py

```
    def __add__(self, other):
        """ Returns a valid Time objects which is Time objects
        'other' added to 'self'.
        >>> print(Time(0,0,0) + Time(1,2,3))
        01:02:03
        >>> print(Time(13,30,0) + Time(1,46,-45))
        15:15:15
        """

    def __sub__(self, other):
        """ Returns a valid Time objects which is Time objects
        'other' substracted from 'self'.
        >>> print(Time(10,10,10) - Time(1,2,3))
        09:08:07
        >>> print(Time(10,0,0) - Time(1,50,600))
        08:00:00
        """
```

3.5 opgave5

Voeg ook de functies `get_current_hours_minutes_seconds()` en `now()` toe. Dit zijn net als de main() geen methodes maar losse functies en krijgen dus ook geen indentatie. Functie `get_current_hours_minutes_seconds()` gebruikt zelf weer een functie uit de Python "time" module waar-

door we een “import time” regel boven aan de file moeten toevoegen. De “*” operator in de `now()` functie niet om te vermenigvuldiggen maar, is een unpack operator die de 3 waarden in de tuple uitpakt zodat er 3 losse argumenten aan de `Time()` methode (de `__init__()` dunder method van class Time) worden meegegeven. Test de `now()` functie in je main() functie door bijvoorbeeld uit te rekenen hoeveel seconden de huidige dag nog duurt.

assignments/opgave5.py

```
import time

# <here goes the Time class with its methods>


def get_current_hours_minutes_seconds():
    """ Returns the current (hours, minutes, seconds) as a tuple. """
    t = time.localtime()
    return (t.tm_hour, t.tm_min, t.tm_sec)


def now():
    """ Returns the current time as Time object. """
    return Time(*get_current_hours_minutes_seconds())
```

Test nu de `Time` class met:

```
$ pytest test_time.py
```

4 Event

Onze alarmklok heeft verschillende gebeurtenissen die een tijd hebben waarop een alarm moet afgaan, bijvoorbeeld om 07:59:30 moet een alarm afgaan voor de “opstaan” gebeurtenis. Om een gebeurtenis te representeren schrijven we de `Event` class. De `Event` class is slechts een container class om de tijd en de beschrijving van een gebeurtenis in op te slaan. Ook al heeft deze class weinig logica, het is toch een nuttige class om te maken omdat het een abstractie toevoegt en het makkelijker maakt om over de code na te denken. De `Event` class, die we gaan schrijven in onderstaande opgaven, is weergegeven in figuur 2. Hierin is ook schematish de relatie met de `Time` class weergegeven, een object van de `Event` class heeft altijd 1 object van de `Time` class.

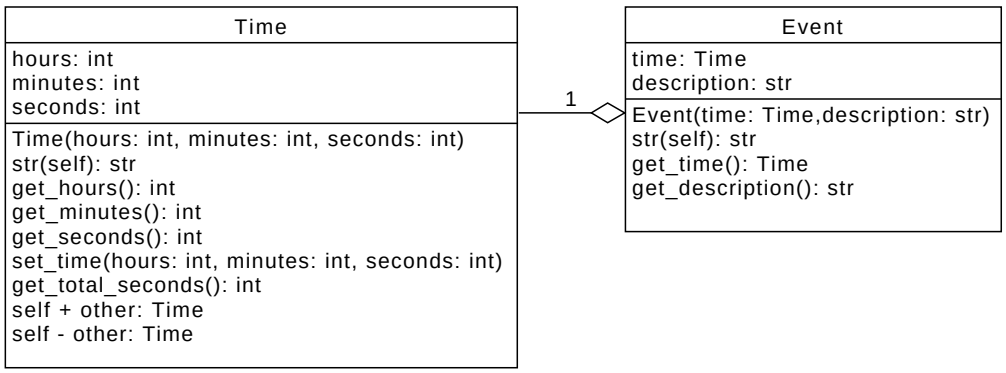

Figure 2: UML class diagram van de Time en Event class.

4.1 opgave6

Voeg onderstaande `Event` class toe en implementeer de methoden van deze class.

assignments/opgave6.py

```
class Event:
    """ Represents an event that happens at a certain time."""

    def __init__(self, time, description):
        """ Initialises an Event object with a 'time' object of type Time and a
        'description' of type str.
        >>> event = Event(Time(18, 30, 0), "dinner")
        """

    def __repr__(self):
        """ Returns the string representation of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner") )
        18:30:00 dinner
        """

    def get_time(self):
        """ Returns the time of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner").get_time() )
        18:30:00
        """

    def get_description(self):
        """ Returns the description of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner").get_description() )
        dinner
        """
```

Test nu de `Event` class met:

```
$ pytest test_event.py
```

5 AlarmClock

Nu zijn we klaar om de `AlarmClock` class te schrijven. Aan een object van class `AlarmClock` class kunnen verschillende events worden toegevoegd. Bij het bereiken van het tijdstip van een event zal de `AlarmClock` het event printen en verwijderen. De `AlarmClock` class is weergegeven in figuur 3 waarbij is weergeven dat een `AlarmClock` object 0 t/m n `Event` objecten kan hebben.

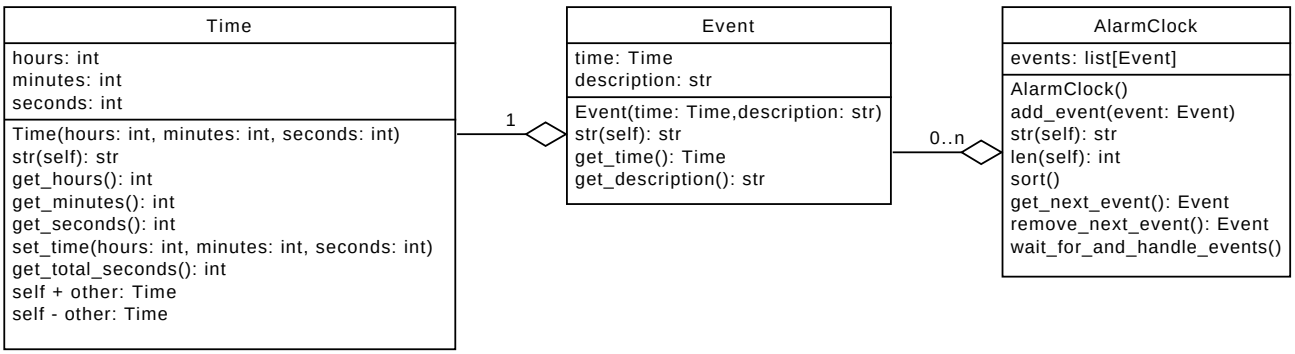

Figure 3: UML class diagram van de Time, Event en AlarmClock class.

5.1 opgave7

Voeg de `AlarmClock` class toe en implementeer onderstaande methoden. Gebruik een list om de verschillende events in op te slaan.

assignments/opgave7.py

```
class AlarmClock:
    """ Represents an alarm clock that can handle events. """

    def __init__(self):
        """ Initialises an AlarmClock object with an empty list of events.
        >>> alarm_clock = AlarmClock()
        """

    def add_event(self, event):
        """ Adds an 'event' to this AlarmClock object, it doesn't return anything.
        >>> alarm_clock = AlarmClock()
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        """

    def __repr__(self):
        """ Returns a string representation of the AlarmClock object.
        >>> alarm_clock = AlarmClock()
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        >>> s = str(alarm_clock)
        >>> "18:30:00" in s
        True
        >>> "dinner" in s
        True
        >>> "breakfast" in s
        False
        """
```

5.2 opgave8

Voeg de `__len__()` dunder method toe aan de `AlarmClock` class en implementeer deze zodat het het aantal events in een `AlarmClock` object teruggeeft.

assignments/opgave8.py

```
    def __len__(self):
        """ Returns the number of events in this AlarmClock object.
        >>> alarm_clock = AlarmClock()
        >>> len(alarm_clock)
        0
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        >>> len(alarm_clock)
        1
        """
```

5.3 opgave9

Voeg de `sort()` methode toe aan de `AlarmClock` class die alle events in een `AlarmClock` object van laag naar hoog sorteert op tijd. Lees voor de implementatie over het sorteren van een lijst. Omdat er geen volgorde voor `Event` objecten gedefinieerd is, is het handig om de key function te gebruiken om aan te geven welke waarde gebruikt moet worden voor het bepalen van de volgorde van events. Gebruik deze `sort()` methode in de `add_event()` methode zodat alle events automatisch op volgorde worden gezet nadat een nieuw event wordt

toegevoegd.

assignments/opgave9.py

```
    def sort(self):
        """ Sorts the events by time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> s = str(alarm_clock)
        >>> s.find("event1") < s.find("event2")
        True
        """
```

5.4 opgave10

Voeg de `get_next_event()` en `remove_next_event()` methoden toe aan de `AlarmClock` class om het eerst volgende event op te vragen dan wel te verwijderen. Implementeer deze methoden.

assignments/opgave10.py

```
    def get_next_event(self):
        """ Returns the next event with the smallest time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.get_next_event().get_description()
        'event2'
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> alarm_clock.get_next_event().get_description()
        'event1'
        """

    def remove_next_event(self):
        """ Removes and returns the next event with the smallest time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> alarm_clock.remove_next_event().get_description()
        'event1'
        >>> alarm_clock.remove_next_event().get_description()
        'event2'
        """
```

5.5 opgave11

Implementeer nu de `AlarmClock` class door als laatste de `wait_for_and_handle_events()` methode aan de class toe te voegen. Deze methode wacht tot de tijd van elke event aanbreekt en print en verdwijdert deze event dan. Gebruik voor de implementatie de sleep(n) functie om n seconden te slapen. De `wait_for_and_handle_events()` methode moet pas returnen wanneer alle events geprint en verwijderd zijn.

assignments/opgave11.py

```
    def wait_for_and_handle_events(self):
        """ Wait for each event to pass and then print the event. """
```

Test deze methode vervolgens in de main() functie, bijvoorbeeld met:

```
def main():
    alarm_clock = AlarmClock()
    alarm_clock.add_event(Event(now() + Time(0, 0,  7), "eat some breakfast"))
    alarm_clock.add_event(Event(now() + Time(0, 0, 12), "off to work"))
    alarm_clock.add_event(Event(now() + Time(0, 0,  2), "good morning, wake up"))
```

```
    # debug here later with print('alarm_clock:', alarm_clock) and mg.s()
    alarm_clock.wait_for_and_handle_events()
```

wat dan elke event op de aangegeven tijd print en dus na 12 seconden ongeveer deze output heeft geven:

```
$ python alarm_clock.py
ALARM: 14:38:11 good morning, wake up
ALARM: 14:38:16 eat some breakfast
ALARM: 14:38:21 off to work
```

Om Python code te debuggen kunnen we tijdelijke print statements toevoegen zoals bijvoorbeeld `print('alarm_clock:', alarm_clock)` in de main() functie, na het toevoegen van de events met `add_event()` calls. Maar, met gebruik van memory_graph kunnen we ook een graaf van alle data genereren voor een beter overzicht wat sneller tot het oplossen van een bug kan leiden, zoals in figuur 4.

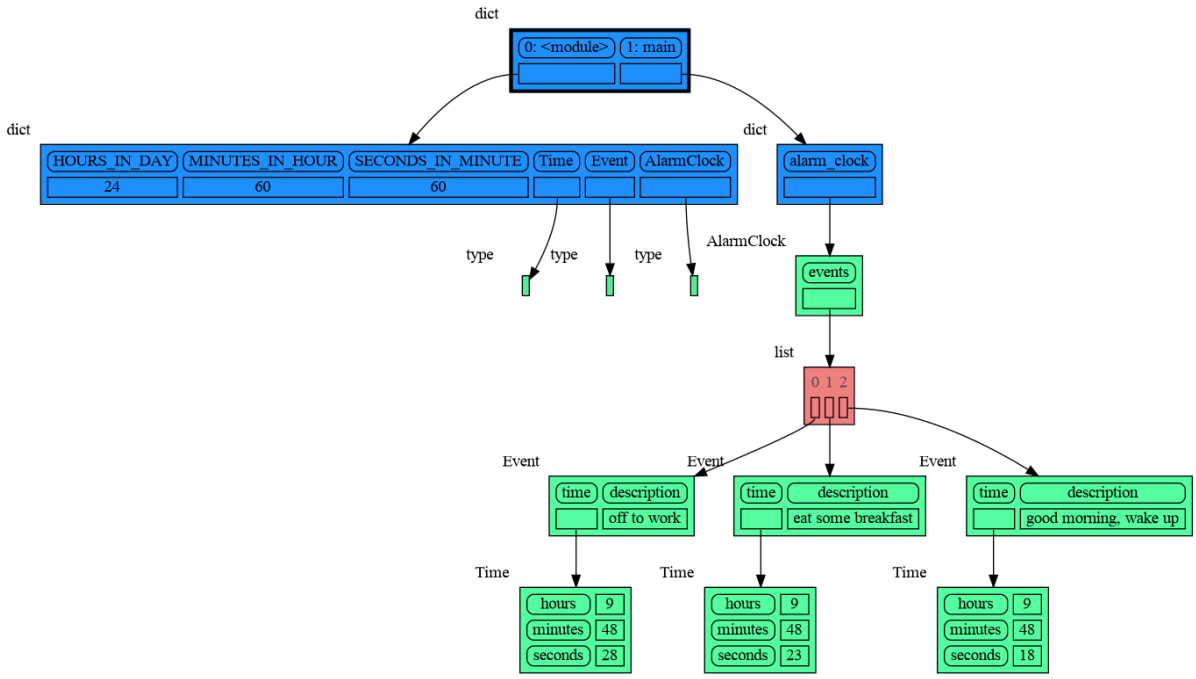

Figure 4: Een memory_graph na toevoegen van events.

6 Speech Synthesis

We hebben nu onze eigen `Time` class geschreven als nuttige oefening van object-georiënteerd programmeren maar, in het algemeen is het beter om eerst te kijken of Python zelf niet al een class heeft die doet wat je wilt zoals bijvoorbeeld de `struct_time` class in de time module. Python biedt heel veel verschillende modules en classes die je gemakkelijk kunt installeren en gebruiken. Stel dat we bijvoorbeeld ook "speech synthesis" aan onze alarmklok willen toevoegen om de event-beschrijving uit te spreken. Na een korte websearch vinden we dat we na installeren van Python module "gtts" en "pygame" met commando:

```
$ pip install gtts pygame
```

we met deze code gemakkelijk speech synthesis kunnen uitvoeren en afspelen:

assignments/speech_synthesis.py

```
from gtts import gTTS
import pygame
```

```python
def text_to_speech(text):
    tts = gTTS(text=text, lang='en')
    filename = "speech.mp3"
    tts.save(filename)
    pygame.mixer.init()
    pygame.mixer.music.load(filename)
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        pass

def main():
    text_to_speech("Hello, this is a speech synthesis test.")


if __name__ == "__main__":
    main()
```

6.1 opgave12

Voeg deze speech synthesis aan de `AlarmClock` class toe zodat wanneer de tijd van een event aanbreekt niet alleen de event wordt geprint maar ook de event-beschrijving wordt uitgesproken.

6.2 Decoupling

Het is **geen** goed idee om de speech synthesis code simpelweg aan de `AlarmClock` class toe te voegen, want dan gebruikt een alarmklok altijd speech synthesis terwijl we dat in andere software, waar we ook de `AlarmClock` class gebruiken, misschien wel helemaal niet willen. Het is veel beter om de `AlarmClock` niet te koppelen met de code die nodig is om een event af te handelen. Een manier om dat te doen is om een functie mee te geven aan de `wait_for_and_handle_events()` methode die verantwoordelijk is voor de afhandeling van events. Ditzelfde principe zagen we al eerder bij de `sort()` methode (of bij de `sorted()` functie) waarbij we een "key" functie konden meegeven die de volgorde van events bepaalde.

Onderstaande voorbeeld laat zien hoe we zelf een `process_numbers()` functie kunnen schrijven waaraan we een `number_handler()` functie kunnen meegeven die dan vervolgens gebruikt wordt om elk nummer af te handelen. De `process_numbers()` functie is hierdoor niet gekoppeld met de code voor de afhandeling en we kunnen daarom dus ook gemakkelijk verschillende afhandelingen gebruiken:

assignments/number_handler.py

```python
def process_numbers(numbers, number_handler):
    print("process_numbers:")
    for i in numbers:
        number_handler(i)


def double_the_number(i):
    print(i * 2)


def repeat_three_times(i):
    print(str(i) * 3)


def main():
    numbers = [1, 2, 3, 4, 5]
    process_numbers(numbers, double_the_number)
    process_numbers(numbers, repeat_three_times)
    process_numbers(numbers, lambda i: print(i * i))
```

```
if __name__ == "__main__":
    main()
```

6.3 opgave13

Voeg een event_handler argument toe aan de `wait_for_and_handle_events()` methode van `AlarmClock` zodat deze class niet gekoppeld is aan de event-afhandeling.

```
def wait_for_and_handle_events(self, event_handler):
```

Roep vervolgens in de `main()` functie deze methode aan met een event_handler die elke event print en uitspreekt wanneer het daar tijd voor is.

7 Information Hidding

De alarmklok software moet ook op digitale horloges gebruikt kunnen worden, maar omdat digitale horloges weinig geheugen hebben moet daarvoor wel eerst de `Time` class worden aangepast zodat het maar 1 integer in plaats van 3 integers gebruikt om de tijd op te slaan. Zo zou de tijd 07:59:30 niet als:

```
self.hours = 7
self.minutes = 59
self.seconds = 30
```

maar bijvoorbeeld als:

```
self.time = 75930
```

in een object van de `Time` class moeten worden opgeslagen. Aangezien de `Event` en `AlarmClock` classes uitsluitend gebruikmaken van de interface van de `Time` class en niet van de interne representatie afhankelijk zijn, blijven deze classes onaangetast door veranderingen in de interne implementatie van de `Time` class. Dit betekent dat aanpassingen beperkt kunnen blijven tot de `Time` class zelf. Dit illustreert een belangrijk voordeel in het werken met interfaces: het mogelijk maken van veranderingen **achter** een interface zonder andere code te hoeven aanpassen.

7.1 opgave14

Herschrijf de implementatie van de `Time` class zodat het maar 1 integer gebruikt om de tijd op te slaan en herschrijf de implementatie van de corresponderende methoden zodat deze aanpassing achter het interface van de `Time` class verborgen blijft en dus de `Event` en `AlarmClock` classes correct blijven werken.

Test nu de `AlarmClock` class met:

```
$ pytest test_alarm_clock.py
```

8 Overzicht

Het uiteindelijke UML class diagram van de alarmklok software ziet er dan uit zoals in figuur 5:

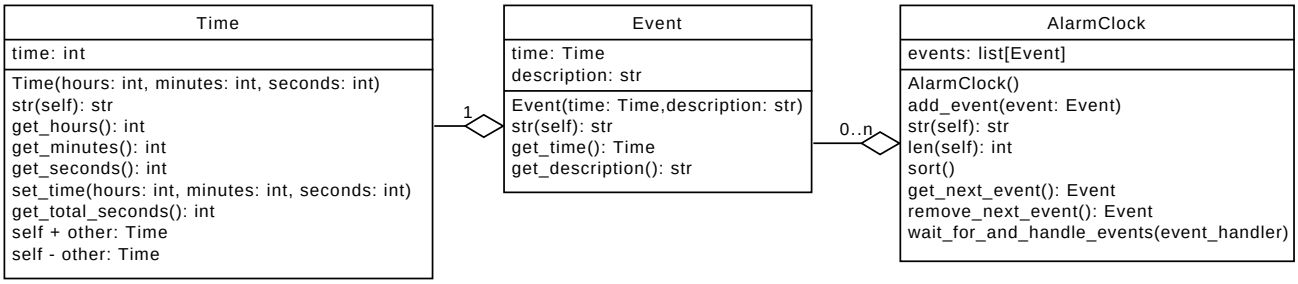

Figure 5: UML class diagram van de alarmklok software.

We hebben in deze opdracht de volgende belangrijke concepten gezien:

- **Abstraction:** we kunnen met gebruik van hun interface de `AlarmClock` en andere classes in andere code hergebruiken zonder de implementatie te hoeven onthouden.
- **Encapsulation:** de interne data van een `Time` object is gecontroleerd en afgeschermd voor direct toegang van buitenaf wat zorgt dat er geen ongeldig `Time` object kan bestaan.
- **Information Hidding:** de interne representatie van een `Time` object is verborgen achter een interface waardoor we deze kunnen aanpassen zonder andere code te hoeven aanpassen.
- **Decoupling:** de `AlarmClock` class is niet gekoppeld aan de event-afhandeling waardoor we zonder code-aanpassingen gemakkelijk verschillende event-afhandelingen kunnen gebruiken.