University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Informatics I – HS18

Exercise 9

Submission Details

- **Submission Format:** ZIP file containing your .py solution files
- **Submission Deadline:** 12:00, Tuesday 27th November
- The name of the ZIP file must have the following format: firstname_lastname_studentidentificationnumber_info1_exercise_9.zip, e.g. *hans_muster_12345678_info1_exercise_9.zip*.
- Your ZIP file should contain the following files: `task_1.py` and `task_2.py`. **Do not rename these files.**
- **Important:** Please follow the naming conventions very strictly, otherwise we will not be able to grade your submission.
- Please submit the assignment using OLAT. You may upload multiple solutions, but note that **only the last submission** will be evaluated.
- Your submissions will be tested on systems running **Python 3.7.X**. Make sure that your solutions work on this version.

Introduction

This week's exercise is again a little bit different. This time around, in task 1 we will give you fully functioning but badly designed code. We will then guide you through the refactoring steps needed in order to make it more cohesive and modular following the **SOLID** principles. Task 2 will be in the usual form and allow you to practice inheritance and class design.

1 Task: Car wash (5 Points)

An intern at *Car Wash Inc.* was tasked with writing a car wash service system with the following requirements:

- A car can enter the shop. The system registers, the name, mobile phone number of the customer and car license plate and assigns it a job service id, with which the customer can later get his car back.
- Once finished, the system notifies the customer that the car is ready for pickup. Notification should be sent via SMS.

The intern after a while came up with this solution:

```
1  class CarWash(object):
2      job_counter = 0
3
4      def __init__(self):
5          self.persistence = {}
6          self.sms_sender = SmsSender()
7
8      def register_car_for_wash(self, car, customer):
9          job_id = CarWash.new_job_id()
10         self.persistence[job_id] = (car, customer)
11         return job_id
12
13     def complete_wash(self, job_id):
14         car, customer = self.persistence[job_id]
15         phone_number = customer.mobile_phone
16         msg = f'Job {job_id}, Car {car.plate} washed.'
17         self.sms_sender.send(phone_number, msg)
18
19     @staticmethod
20     def new_job_id():
21         job_id = CarWash.job_counter
22         CarWash.job_counter += 1
23         return f'#{job_id}'
24
25
26 class Customer(object):
27
28     def __init__(self, name, mobile_phone):
29         self.name = name
30         self.mobile_phone = mobile_phone
31
32
33 class Car(object):
34
35     def __init__(self, plate):
36         self.plate = plate
37
```

```
38
39 class SmsSender(object):
40
41     def send(self, phone_number, msg):
42         print(f'Sending text message to {phone_number}: {msg}')
43         # ... do some weird SMS magic
44
45
46 if __name__ == '__main__':
47     car_wash = CarWash()
48     car1 = Car('ZH 123456')
49     car2 = Car('AG 654321')
50     customer1 = Customer('Foo', '079 xxx xxxx')
51     customer2 = Customer('Bar', '078 xxx xxxx')
52
53     job_id1 = car_wash.register_car_for_wash(car1, customer1)
54     job_id2 = car_wash.register_car_for_wash(car2, customer2)
55     assert job_id1 != job_id2
56
57     car_wash.complete_wash(job_id1)
```

Listing 1: Initial code. File `task_1.py`.

While this solution does indeed work, it has some issues and violates some design principles. For example, the class `CarWash` does not respect the *Single Responsibility Principle (SRP)*, by handling the persistence and creating new job ids, nor does it respect *Dependency Inversion* by implicitly depending on `SmsSender`.

After a while two additional feature requests came in:

- The system saves everything in-memory, eventually a better persistence solution will be needed. Add the possibility to write data to file, so that it can persist between application start-ups. The existing in-memory functionality, should not be deleted to simplify testing.
- At the moment only SMS notifications are supported, but some customers have expressed the wish to be notified by phone call or email.

The intern unfortunately does not work at the company anymore, so your manager has tasked you with implementing the new features.

Increasing cohesion

Let's kick-off our refactoring session by pulling all car wash information together into a single class. First define a class `CarWashJob`, the advantage of using this class instead of working with tuples is to offload some responsibilty from `CarWash` into a separate class, putting all data which is related together. This class should wrap the `customer` and `car` objects and the `job_id`. Add also two properties:

- `contact_details` which returns the customers mobile phone number
- `notification_message` which creates a string exactly like the one we passed on previoulsy to `sms_sender`.
  I.E. `Job {job_id}, Car {plate} washed.`

Finally, pull the `new_job_id` functionality inside the `CarWashJob` and modify the constructor such that if no id is given, it should create a new one using `new_job_id`.

Task: implement the `CarWashJob` class such that the `CarWash` as implemented below works.

```
1  class CarWash(object):
2
3      def __init__(self):
4          self.persistence = {}
5          self.sms_sender = SmsSender()
6
7      def register_car_for_wash(self, car, customer):
8          job = CarWashJob(car, customer)
9          self.persistence[job.job_id] = job
10         return job.job_id
11
12     def complete_wash(self, job_id):
13         job = self.persistence[job_id]
14         self.sms_sender.send(job.contact_details, job.notification_message
       )
15
16 # ========================================
17 #
18 #   Class CarWashJob goes here...
19 #
20 # ========================================
21
22 if __name__ == '__main__':
23     car_wash = CarWash()
24
25     car1 = Car('ZH 123456')
26     car2 = Car('AG 654321')
27     customer1 = Customer('Foo', '079 xxx xxxx')
28     customer2 = Customer('Bar', '078 xxx xxxx')
29
30     job_id1 = car_wash.register_car_for_wash(car1, customer1)
31     job_id2 = car_wash.register_car_for_wash(car2, customer2)
32
33     car_wash.complete_wash(job_id2)
34     car_wash.complete_wash(job_id1)
```

Listing 2: Initial code. After adding `CarWashJob`.

Refactoring the persistence layer

We'll start now with the refactoring of the persistence functionality.

One bad solution might be to set a flag in `CarWash` which tells us how we want to save items. Then every time we want to save or retrieve an object, we can check if we want to fetch from disk or from the dictionary:

```
class CarWash(object):

    def __init__(self, persistence_mode, filename):
        self.persistence_mode = persistence_mode
        self.inmemory_persistence = {}
        self.filename = filename
        # [...]

    def register_car_for_wash(self, car, customer):
        if self.persistence_mode == 'inmemory':
            # save to dictionary
            # [...]
            pass
        else:
            # save to file
            # [...]
            pass

        return job_id

    def complete_wash(self, job_id):
        if self.persistence_mode == 'inmemory':
            # fetch from dictionary
            # [...]
            pass
        else:
            # fetch from file
            # [...]
            pass

        # rest of method...
```

Listing 3: Bad way to extend the functionality. Do not do this!

The problem with this variant is that the `CarWash` is very tightly coupled to both persistence strategies. Any time we add a new persistence strategy we will have to refactor the `if/else` statements with new cases which hurts the readability of the code making it harder to reason about. Code which is hard to maintain will inevitably be more error-prone.

The better solution is to leverage *dependency inversion*:

1. We define an abstract `CarJobRepository` class with two abstract methods: `save` and `find_by_id`.
2. We then implement the `FileCarJobRepository` and `InMemoryCarJobRepository`
   - `InMemoryCarJobRepository` persists the jobs in and retrieves from a dictionary as before
   - `FileCarJobRepository` writes the entries to file upon saving and retrieves them from file when calling `find_by_id`. It should also have a method `drop_db()` which deletes the contents of the file. The same method is also used by the constructor in case the optional `drop_on_startup` parameter is set to `True`. The constructor also takes the name of the file to use as database.

3. All implementations of `find_by_id` should raise a `ValueError` in case no job with the given id is found.
4. Finally, modify the constructor of `CarWash`, so that the repository is injected into `CarWash`.

Task: Implement the steps described above.

```
class CarWash(object):

    def __init__(self, persistence):
        self.persistence = persistence
        self.sms_sender = SmsSender()

    def register_car_for_wash(self, car, customer):
        job = self.persistence.save(CarWashJob(car, customer))
        return job.job_id

    def complete_wash(self, job_id):
        job = self.persistence.find_by_id(job_id)
        self.sms_sender.send(job.contact_details, job.notification_message
    )


# ================================
# [...]
# ================================

if __name__ == '__main__':
    in_memory_db = InMemoryCarJobRepository()
    file_db = FileCarJobRepository('car-wash-db.tsv', drop_on_startup=True
    )
    in_memory_car_wash = CarWash(in_memory_db)
    file_db_car_wash = CarWash(file_db)
```

Listing 4: Clean persistence design using dependency inversion. This design allows us to easily swap any implementation of `CarJobRepository` without needing to change the `CarWash` class itself.

Refactoring the notification layer

Try to refactor the `SmsSender` dependency similarly as you did with the `Repository` following the dependency inversion principle. Call the abstract class `Notifier`. At the end you should be able to freely swap any concrete implementation such as e.g. `EmailSender`, `PhoneCallSender`, etc. To test this add a `CallSender` class (you don't have to implement anything fancy, it can behave as the `SmsSender` does, just slightly change the print message).

Task: add the `Notifier` class and its subclasses and modify `CarWash` such that it adheres to the dependency inversion principle.

```
class CarWash(object):

    def __init__(self, persistence, notifier):
```

```
4           self.persistence = persistence
5           self.notifier = notifier
6
7       def register_car_for_wash(self, car, customer):
8           job = self.persistence.save(CarWashJob(car, customer))
9           return job.job_id
10
11      def complete_wash(self, job_id):
12          job = self.persistence.find_by_id(job_id)
13          self.notifier.send(job)
14
15
16  # ================================
17  # [...]
18  # ================================
19
20  if __name__ == '__main__':
21      in_memory_db = InMemoryCarJobRepository()
22      file_db = FileCarJobRepository('car-wash-db.tsv', drop_on_startup=True
        )
23
24      sms_sender = SmsSender()
25      call_sender = CallSender()
26
27      car_wash = CarWash(in_memory_db, sms_sender)
28      car_wash = CarWash(file_db, call_sender)
```

Listing 5: Refactored notifier layer using dependency inversion.

Conclusion

At the end of these steps we should now have a much cleaner design following the **SOLID** principles:

- **Single Responsibility**: The class `CarWash` now does one thing only; it handles the logic of the car wash, but all details concerning saving data and notifying users is delegated to other parts of the code. For example: if we want to change the dictionary in `InMemoryCarJobRepository` to some other data structure, we do not need to change `CarWash`.
- **Open/Closed**: it is possible to extend the functionality of this system without actually changing `CarWash`. Say we want to add a new way of notifying the user. We can simply create a new implementation which abides to the necessary contract and pass this to `CarWash` on construction.
- **Liskov substitution**: we can swap out any implementation of `CarJobRepository` or `Notifier` for another. We can even change these dynamically at run-time. For example if the SMS sending service is offline for some reason, we could simply swap it out for another implementation, without even restarting the application.
- **Interface Segregation**: does not really apply here :-). However consider what would happen if the `drop_db()` method was added to the contract of the abstract `CarJobRepository`

class: any subclass would have to implement it as well. While it would make sense for many implementations, for the `InMemoryCarJobRepository` it would not. `InMemoryCarJobRepository` would then probably implement the method with an empty body or by raising an exception. This would violate both **Interface Segregation** as well as **Liskov** (the class would not abide the contract of the super class)

- **Dependency inversion**: `CarWash` does not depend on any concrete implementation. It only needs to know the contract defined by the abstraction to be able to use its dependencies.

2 Task: Santa's Elves (5 Points)

As Christmas is approaching, Santa is very busy preparing all the presents for the world's children. Noticing that kids' Christmas wishes get bigger every year, Santa has decided that he needs to automate some of his tasks, otherwise he will not be able to finish every present in time. You will implement an assembly line where toys are assembled, painted and wrapped by Santa's helper elves.

Instructions:

- Create a class `Toy`:
  - It must be initialized with a name argument and have three instance variables `is_assembled`, `is_painted` and `is_wrapped`. Toys are **not** assembled, painted and wrapped initially.
  - Define a method `is_complete()` which returns a boolean indicating whether the toy is assembled, painted and wrapped.
- Create a class `AssemblyLine`:
  - Initialize it with a list of toys and store that list in a private instance variable `__toys`. You can assume that your class will always be initialized with a list of `Toy` objects, so you do not need to check for the type of the argument.
  - Create the method `get_toys` which allows you to get the list of toys in the assembly line. Also implement a `get_number_of_toys` method that returns the number of toys in the assembly line.
  - Create a method `take_toy` that removes the first toy from the assembly line (i.e., the toy at position 0 in the list) and returns it. If there are no toys in the assembly line, `None` should be returned.
  - Create the corresponding method `put_toy_back(toy)` that adds the toy back to the assembly line (at the last position). Make sure it is not possible to put anything other than a toy back into the assembly line.
- Create an abstract class `Elf`:
  - It will be an abstract superclass for the different elf classes you will create. It should be initialized without arguments and create an internal instance variable `_toy_working_on` that is initially `None`.
  - It should have an abstract `do_job` method. This method will be overwritten in the subclasses, so you can just `pass` it.
  - Write a method `take_from(assembly_line)` which takes a toy from an assembly line that is passed as parameter and stores it to `_toy_working_on`. If the elf is already working on a toy, this method should do nothing.
  - Write the corresponding method `put_back(assembly_line)` that puts the toy the elf was working on back into the given assembly line. After putting back the toy, the elf should have no reference to the toy it has worked on anymore.
- Now create three subclasses of `Elf`, namely `AssemblerElf`, `PainterElf` and `WrapperElf`. The only thing you have to do in these subclasses is override the `do_job` method. Depending on the type of elf, it should take care of its part of the job (assembling/painting/wrapping) on the toy currently working on, which in this case means setting the respective value to True. If the elf is not currently working on an item, the `do_job` method will not do anything of course.
  Note: While it does not matter whether a toy is assembled or painted first, the wrapping of course can only happen after it has already been painted and assembled. You need to consider this fact in your `do_job` implementation. In case of an invalid wrapping attempt, do not throw an exception, but just leave the toy unwrapped.

- Finally, create a class `Santa`. The only method you need to implement in this class is `verify(assembly_line)`, which takes an assembly line and returns a boolean value indicating whether all toys in the assembly line are completed or not.

Note: Check the example code below to understand how the methods and classes are supposed to work.

Task: Implement the classes, following the instructions above.

```
if __name__ == '__main__':
    # Create three toys
    toy1 = Toy("Toy1")
    toy2 = Toy("Toy2")
    toy3 = Toy("Toy3")

    # Create an assembly line with three toys
    line = AssemblyLine([toy1, toy2, toy3])

    # Create three elves, one of each subclass
    assembler = AssemblerElf()
    painter = PainterElf()
    wrapper = WrapperElf()

    # Create a Santa :-)
    santa = Santa()

    # Let the elves work through the assembly line
    for elf in [assembler, wrapper, painter]:  # Wrong order: wrapping can't happen before painting!
        for i in range(line.get_number_of_toys()):
            elf.take_from(line)
            elf.do_job()
            elf.put_back(line)

    # The line cannot be verified because the toys are not wrapped
    assert not santa.verify(line)

    # Create three new toys...
    toy4 = Toy("Toy4")
    toy5 = Toy("Toy5")
    toy6 = Toy("Toy6")

    # ... and a new assembly line
    line2 = AssemblyLine([toy4, toy5, toy6])

    # This time, let the elves work through the assembly line in the right order
    for elf in [painter, assembler, wrapper]:  # Right order: wrap at the end!
```

```
        for i in range(line2.get_number_of_toys()):
            elf.take_from(line2)
            elf.do_job()
            elf.put_back(line2)

    # Now the line can be verified
    assert santa.verify(line2)
```

Listing 6: Example usage.