Zeszyty Naukowe WWSI, No 27, Vol. 16, 2022, s. 39-56
DOI: 10.26348/znwwsi.27.39

Code refactoring: a Python example

Ewa Figielska*

Warsaw School of Computer Science

Abstract

In this paper, several refactoring techniques are shown, using an example in which the design of a program for solving a simple problem is gradually improved. Before introducing any change to the program, the drawbacks of its current version are discussed, bad code smells are identified, and some unit tests are provided. The source code is written in Python.

Keywords – Refactoring, Code smells, Object-oriented programming, Unit tests, Python

* E-mail: efigielska@ms.wwsi.edu.pl

Manuscript received October 28, 2022.

1. Introduction

Refactoring is the process of improving the design, structure, and implementation of a software system without changing its external behavior. It is performed on an existing and working code, negating an old engineering adage “if it works, don’t fix it”. The aim of refactoring is to make code cleaner, easier to read and understand and thus to provide software which can be modified and extended with a low risk that introduced changes break something, i.e. to provide easy maintainable software. The term refactoring is usually used in the context of object-oriented systems. It dates back to the 90s of the last century, when the first in-depth study of code refactoring as a software engineering technique was presented in Opdyke PhD thesis “Refactoring object-oriented frameworks” [1, 2].

In this paper, some refactoring techniques are shown, using an example in which a stand-alone procedure is transformed into an object-oriented program whose design is further improved in several steps. In each step, the current design drawbacks are discussed, bad code smells pointed out and appropriate changes introduced. Also, a number of unit tests are provided in response to successive code changes. The source code is written in the Python programming language.

2. Object-oriented programming features

Object-oriented programming offers a number of advantages over procedural approaches [3, 4], which concern modularity, reusability and flexibility of code. Huge problems can be effectively solved by dividing them into smaller modules, i.e. classes, which represent objects from the problem domain. A class bundles together data and operations performed on these data. For example, a class can represent a game hero who fights with a sword. If a hero gets a new weapon or changes the way of fighting, necessary code modifications are restricted to the hero class, whereas in a procedural approach all places referring to the hero data in the procedures external to these data must be found and changed.

In object-oriented programming, code can be reused through inheritance. Shared characteristics from two or more classes can be extracted and combined into a generalized superclass. For example, attributes such as name and strength along with operations of strength checking and fighting which are common to all the heroes can

be moved to a superclass representing a general hero. Specialized heroes, such as a dwarf or an elf, defined by subclasses of the hero class, inherit these attributes and operations and can make some extensions. Namely, they can add attributes of their own, and add new operations or redefine the inherited ones. For example, a dwarf has an axe, an elf has a bow and arrows whose stock should be replenished during a fight. An elf and a dwarf, due to different ways of fighting (one shoots arrows with a bow and the other strikes with an axe) require different definitions of operation (method) `fight`, which, on the other hand, has no specific definition for the general hero. So, the general hero is an abstract concept, and therefore its class can be defined as an abstract one.

An abstract class on the top of an inheritance hierarchy declares all the operations performed by its subclasses (it determines their common interface), thus becoming a representative of the subclasses. Other parts of the program can work with the objects of subclasses knowing only their representative. For example, a battle can start with executing method `fight` for each warrior in an army, and each warrior will do fighting in a specific way defined in its class: an elf will be shooting arrows with a bow, a dwarf will be striking with an axe, a wizard will be casting spells. So, by one command `fight` many different forms of fighting are performed. This ability of a function to take many forms is called polymorphism. Due to polymorphism flexibility of code is achieved. No changes in the existing code of the battle class or its commander class are needed when the army is extended by other types of warriors, so long as they all implement the interface given by an abstract superclass, see Figures 1 and 2.

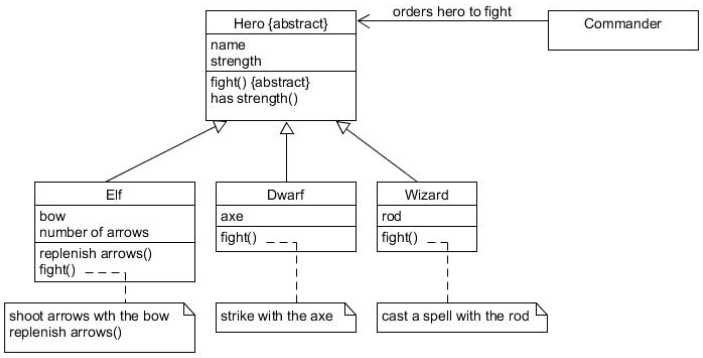

Figure 1. Abstract class Hero defining the interface which is used by the Commander class when working with concrete heroes.

```
list_of_heros = [Elf("GoldLeaf",200), Dwarf("GoodMiner"), Wizard("TheBlack")]
for hero in list_of_heros:
    if hero.has_strength():
        hero.fight()
```

Figure 2. Polymorphic calls of method `fight()` which has different implementations for different heroes.

3. Refactoring and code smells

All computer-based systems change during their lifetime due to the changes in their environment, the necessity of error corrections, user requests for enhancements etc. [5]. But introducing changes to software may ruin abstraction and relationships between classes, increase the number of errors, and thus make further changes more difficult. In practice, even a well-designed software, which initially conforms with the object-oriented concepts and principles, may, after a number of changes, become unclear and difficult to maintain. To avoid such a situation, some improvements, i.e. refactorings, should be introduced before any big change to code (i.e. before adding a new feature) and, if afterwards the working code is not as clear as it could be, further refactoring is needed. Refactoring in itself may be risky, and therefore it cannot be practiced informally or ad hoc but should proceed in a systematic and disciplined way, in small steps, one at a time [6, 7].

A developer can be warned about the code quality problems by so called bad code smells. The word smell means an easily spotted symptom of a possibly severe problem which itself can be difficult to detect. The term code smell was coined by Beck and Fowler and brought to general use by presenting an informal definition of 22 code smells in the 1999 book “Refactoring: Improving the Design of Existing Code” [6]. Bad smells indicate the places in the program which may need refactoring and can also suggest the ways of refactoring [8, 9, 10].

One of the most common signs of code smells are code bloaters, namely elements of code which are hard to work with because of their great size, for example a method with too many lines or which does too many things (smell Long Method), a class with too many fields and methods which imply too many responsibilities (smell Large Class), many parameters on the method parameter list (smell Long Parameter List), related variables appearing together in different parts of code (smell Data Clumps), primitive data types used to represent domain ideas instead of domain-

specific types (smell Primitive Obsession). There are code smells that abuse object orientation of a program, such as complex `switch` or `if…else` statements (smell Switch Statements), fields that get their values only under certain circumstances and outside these circumstances they are empty (smell Temporary Fields), classes with identical functions but with different method names (smell Alternative Classes with Different Interfaces), subclasses using only some of the inherited elements (smell Refused Bequest). Sometimes development of a program is inhibited by situations when one change in code alters many classes, or a single class needs to be edited many times when changes are made outside the class (smells Shotgun Surgery and Divergent Change, respectively) or when creating a subclass for a class requires creating a subclass of another class (smell Parallel Inheritance Hierarches). Some of code elements may be dispensable, for example, comments that can be replaced by good names (smell Comments), the same code in two or more places (smell Duplicate Code), a class containing only fields and methods for accessing them (smell Data Class), not used code element (smell Dead Code), a class that does not doing enough (smell Lazy Class), a class cautiously prepared for many possible future needs (smell Speculative Generality). Classes may be excessively coupled, for example a method using the data of another class object more frequently than its own data (smell Feature Envy), classes tightly linked to one another by using each other private parts (smell Inappropriate Intimacy), a class mainly delegating work to another class (smell Middle Man). High coupling occurs also in a situation when a client asks an object for another object which in turn asks for another object and so on (smell Message Chains).

4. Example

In this section, an example is presented, in which an object-oriented program for a simple problem is developed in several steps. It is inspired by the example given in [10]. In each consecutive step, the current design of the program is discussed, bad code smells and other code flaws are indicated, and improvements are made accordingly. To prevent introducing bugs, the working code should be tested after each change, therefore the exemplary unit tests are also provided.

4.1. Problem

The problem solved is to determine recommended daily calories, an ideal body weight, and a distance from the ideal weight for a patient.

The recommended daily calories (basal metabolic rate) are calculated from the Mifflin St. Jeor equation [11] in the following way:

$$basal\ metabolic\ rate = \begin{cases} 10W + 6.25H - 5A + 5 & \text{for men} \\ 10W + 6.25H - 5A - 161 & \text{for women} \end{cases}$$

where W is body weight in kilograms, H is body height in centimeters, A is age in years, and $basal\ metabolic\ rate$ is reported in kcal/day.

For calculating the ideal body weight, the Devine equation [12] is used:

$$ideal\ body\ weight = \begin{cases} 50 + 0.9(H - 152) & \text{for men} \\ 45.5 + 0.9(H - 152) & \text{for women} \end{cases}$$

where $ideal\ body\ weight$ is reported in kilograms.

The distance d of the patient weight from the ideal body weight is calculated as follows:

$$d = W - ideal\ body\ weight$$

4.2. First program

The first program consists of stand-alone function `make_calculations`, shown in Figure 3, which calculates and returns the values of the recommended calories, the ideal body weight and the distance from the ideal weight for a patient. The patient characteristics: height, weight, age and sex, are input values given by a user. The example of a unit test with mocked inputs for function `make_calculations` is shown in Figure 4. Because the function returns as a result a tuple with three values, three assertions are made in the test.

```
def make_calculations():
    # h=height, w=weight, a=age, s=sex
    h = float(input("patient height (cm): "))
    w = float(input("patient weight (kg): "))
    a = float(input("patient age (years): "))
    s = input("patient sex (male/female): ")

    if s == "male":
        # calculates basal metabolic rate for a male
        mr = 10*w + 6.25*h - 5*a + 5
        # calculates ideal body weight for a male
        bw = 50 + 0.9*(h - 152)
    else:
        # calculates basal metabolic rate for a female
        mr = 10*w + 6.25*h - 5*a - 161
        # calculates ideal body weight for a female
        bw = 45.5 + 0.9*(h - 152)

    # calculates distance from ideal weight
    d = w - bw
    # returns the result
    return mr, bw, d
```

Figure 3. A stand-alone function for calculating recommended calories, an ideal body weight and a distance from ideal weight for a patient.

```
from unittest import TestCase, mock
from calories_calculations import make_calculations


class Test(TestCase):
    @mock.patch('calories_calculations.input', create=True)
    def test_make_calculations(self, mocked_input):
       mocked_input.side_effect = ['170', '60', '40', 'female', 'done']
       solution = make_calculations()
       self.assertAlmostEqual(1301.5, solution[0])
       self.assertAlmostEqual(61.7, solution[1])
       self.assertAlmostEqual(-1.7, solution[2])
```

Figure 4. Example of a unit test for function `make_calculations`.

4.3. Refactoring

Function `make_calculations` shown in Figure 3 does too much being responsible for the interaction with a user to get inputs as well as making required calculations (bad smell Long Method). Therefore, these two responsibilities are separated and placed in two functions `take_patient_parameters` and `make_calculations` as shown in Figure 5. The modified unit test is presented in Figure 6.

```
def take_patient_parameters():
    h = float(input("patient height (cm): "))
    w = float(input("patient weight (kg): "))
    a = float(input("patient age (years): "))
    s = input("patient sex (male/female): ")
    return h, w, a, s


def make_calculations(h, w, a, s):
    # h=height, w=weight, a=age, s=sex

    if s == "male":
        # calculates basal metabolic rate for a male
        mr = 10*w + 6.25*h - 5*a + 5
        # calculates ideal body weight for a male
        bw = 50 + 0.9*(h - 152)
    else:
        # calculates basal metabolic rate for a female
        mr = 10*w + 6.25*h - 5*a - 161
        # calculates ideal body weight for a female
        bw = 45.5 + 0.9*(h - 152)

    # calculates distance from ideal weight
    d = w - bw
    # returns the result
    return mr, bw, d
```

Figure 5. Separation of making calculations from taking data from a user.

```
class Test(TestCase):
    def test_make_calculations(self):
       solution = make_calculations(170, 60, 40, 'female')
       self.assertAlmostEqual(1301.5, solution[0])
       self.assertAlmostEqual(61.7, solution[1])
       self.assertAlmostEqual(-1.7, solution[2])
```

Figure 6. Example of a unit test for function `make_calculations` separated from taking user data.

In Figure 5, we can see that function `make_calculations` takes four parameters: the height, weight, age, and sex of a patient (bad smell Long Parameter List). Also, because the names of variables do not show their meaning, comments are used to describe what is calculated in the successive lines of the code (bad smell Comments). Therefore, to improve the program, the patient characteristics are encapsulated in one class `Patient`, whose instance is passed to the `make_calculations` function

as a parameter, as shown in Figure 7. The variable names are changed, so that comments are no longer needed to understand the code. The unit test shown in Figure 8 creates the `Patient` object in the `setUp` method to be passed to the tested `make_calculations` function.

```
class Patient:
    def __init__(self):
        self.height = None
        self.weight = None
        self.age = None
        self.sex = None

    def take_patient_parameters(self):
        self.height = float(input("patient height (cm): "))
        self.weight = float(input("patient weight (kg): "))
        self.age = float(input("patient age (years): "))
        self.sex = input("patient sex (male/female): ")


def make_calculations(patient):
    if patient.sex == "male":
        metabolic_rate = 10*patient.weight + 6.25*patient.height - 5*patient.age + 5
        ideal_weight = 50 + 0.9*(patient.height - 152)
    else: # for a female
        metabolic_rate = 10*patient.weight + 6.25*patient.height - 5*patient.age - 161
        ideal_weight = 45.5 + 0.9*(patient.height - 152)

    distance_from_ideal_weight = patient.weight - ideal_weight
    return metabolic_rate, ideal_weight, distance_from_ideal_weight
```

Figure 7. Program after encapsulating patient characteristics in a class.

```
class Test(TestCase):
    def setUp(self):
        self.patient = Patient()
        self.patient.height = 170
        self.patient.weight = 60
        self.patient.age = 40
        self.patient.sex = 'female'

    def test_make_calculations(self):
        solution = make_calculations(self.patient)
        self.assertAlmostEqual(1301.5, solution[0])
        self.assertAlmostEqual(61.7, solution[1])
        self.assertAlmostEqual(-1.7, solution[2])
```

Figure 8. Example of a unit test for function `make_calculations` which takes a `Patient` object as a parameter.

The stand-alone `make_calculations` function presented in Figure 7 works mainly with the data attributes of class `Patient` (bad smell Feature Envy). On the other hand, class `Patient` does not process its data, it only gets and stores data (bad smell Data Class). Therefore, function `make_calculations` is moved to class `Patient` as shown in Figure 9. Figure 10 shows the unit test with the `make_calculations` method executed for the `Patient` object.

```
class Patient:
    def __init__(self):
        self.height = None
        self.weight = None
        self.age = None
        self.sex = None

    def take_patient_parameters(self):
        self.height = float(input("patient height (cm): "))
        self.weight = float(input("patient weight (kg): "))
        self.age = float(input("patient age (years): "))
        self.sex = input("patient sex (male/female): ")

    def make_calculations(self):
        if self.sex == "male":
            metabolic_rate = 10 * self.weight + 6.25 * self.height - 5 * self.age + 5
            ideal_weight = 50 + 0.9*(self.height - 152)
        else:  # for a female
            metabolic_rate = 10 * self.weight + 6.25 * self.height - 5 * self.age - 161
            ideal_weight = 45.5 + 0.9*(self.height - 152)

        distance_from_ideal_weight = self.weight - ideal_weight
        return metabolic_rate, ideal_weight, distance_from_ideal_weight
```

Figure 9. Class `Patient` with method `make_calculations`.

```
class TestPatient(TestCase):
    def setUp(self):
        self.patient = Patient()
        self.patient.height = 170
        self.patient.weight = 60
        self.patient.age = 40
        self.patient.sex = 'female'

    def test_make_calculations(self):
        solution = self.patient.make_calculations()
        self.assertAlmostEqual(1301.5, solution[0])
        self.assertAlmostEqual(61.7, solution[1])
        self.assertAlmostEqual(-1.7, solution[2])
```

Figure 10. Example of a unit test for method `make_calculations` called for the `Patient` object.

In method `make_calculations` presented in Figure 9 several arithmetic calculations are performed using different formulas. If a change is needed in one of the formulas, the right line in the method code must be found, and, after introducing the change, the whole method must be tested (bad smell Long Method). To avoid this, it would be better to place each formula in a separate method. Therefore, the following four methods are extracted from `make_calculations`: `daily_calories_recommended_male`, `ideal_body_weight_male`, `daily_calories_recommended_female`, and `ideal_body_weight_female`. The resulting code is shown in Figure 11. Figure 12 presents the examples of unit tests for the extracted methods.

```
class Patient:
    def __init__(self):
        self.height = None
        self.weight = None
        self.age = None
        self.sex = None

    def take_patient_parameters(self):
        self.height = float(input("patient height (cm): "))
        self.weight = float(input("patient weight (kg): "))
        self.age = float(input("patient age (years): "))
        self.sex = input("patient sex (male/female): ")

    def make_calculations(self):
        if self.sex == "male":
            metabolic_rate = self.daily_calories_recommended_male()
            ideal_weight = self.ideal_body_weight_male()
        else:
            metabolic_rate = self.daily_calories_recommended_female()
            ideal_weight = self.ideal_body_weight_female()

        distance_from_ideal_weight = self.weight - ideal_weight
        return metabolic_rate, ideal_weight, distance_from_ideal_weight

    def daily_calories_recommended_male(self):
        return 10 * self.weight + 6.25 * self.height - 5 * self.age + 5

    def ideal_body_weight_male(self):
        return 50 + 0.9 * (self.height - 152)

    def daily_calories_recommended_female(self):
        return 10 * self.weight + 6.25 * self.height - 5 * self.age - 161

    def ideal_body_weight_female(self):
        return 45.5 + 0.9 * (self.height - 152)
```

Figure 11. Class `Patient` after extraction methods for calculating recommended daily calories and an ideal body weight for men and women.

```
class TestPatient(TestCase):
    def setUp(self):
        self.patient = Patient()
        self.patient.height = 170
        self.patient.weight = 60
        self.patient.age = 40
        self.patient.sex = 'female'

    def test_make_calculations(self):
        solution = self.patient.make_calculations()
        self.assertAlmostEqual(1301.5, solution[0])
        self.assertAlmostEqual(61.7, solution[1])
        self.assertAlmostEqual(-1.7, solution[2])

    def test_daily_calories_recommended_female(self):
        solution = self.patient.daily_calories_recommended_female()
        self.assertAlmostEqual(1301.5, solution)

    def test_ideal_body_weight_female(self):
        solution = self.patient.ideal_body_weight_female()
        self.assertAlmostEqual(61.7, solution)
```

Figure 12. Example of unit tests for two of the extracted methods.

```
patient = Patient()
patient.take_patient_parameters()
daily_colories, ideal_weight, distance_from_ideal_weight = patient.make_calculations()
print("calories recommended =", daily_colories)
print("ideal weight =", ideal_weight)
print("distance from ideal weight =", distance_from_ideal_weight)
```

Figure 13. Using values returned by method `make_calculations`.

Method `make_calculations` presented in Figure 11 returns three values, so whenever the method call is placed in the program, we must verify the order in which these values are returned to be able to use them correctly, as shown in Figure 13. Moreover, “make calculations” is not a good name for a function, it does not indicate what exactly the function calculates. But as it calculates not one but several different things, further extraction is performed, resulting in five new methods which replace `make_calculations`, as presented in Figure 14. Due to such extraction, the code using recommended calories, ideal weight, and distance from ideal weight shown in Figure 15, is less prone to bugs than the code in Figure 13. The examples of unit tests for new methods are shown in Figure 16.

```
class Patient:
    def __init__(self):
        self.height = None
        self.weight = None
        self.age = None
        self.sex = None

    def take_patient_parameters(self):
        self.height = float(input("patient height (cm): "))
        self.weight = float(input("patient weight (kg): "))
        self.age = float(input("patient age (years): "))
        self.sex = input("patient sex (male/female): ")

    def daily_calories_recommended(self):
        if self.sex == "male":
            return self.daily_calories_recommended_male()
        else:
            return self.daily_calories_recommended_female()

    def ideal_body_weight(self):
        if self.sex == "male":
            return self.ideal_body_weight_male()
        else:
            return self.ideal_body_weight_female()

    def distance_from_ideal_weight(self):
        if self.sex == "male":
            return self.distance_from_ideal_weight_male()
        else:
            return self.distance_from_ideal_weight_female()

    def distance_from_ideal_weight_male(self):
        return self.weight - self.ideal_body_weight_male()

    def distance_from_ideal_weight_female(self):
        return self.weight - self.ideal_body_weight_female()

    def daily_calories_recommended_male(self):
        return 10 * self.weight + 6.25 * self.height - 5 * self.age + 5

    def ideal_body_weight_male(self):
        return 50 + 0.9 * (self.height - 152)

    def daily_calories_recommended_female(self):
        return 10 * self.weight + 6.25 * self.height - 5 * self.age - 161

    def ideal_body_weight_female(self):
        return 45.5 + 0.9 * (self.height - 152)
```

Figure 14. Class `Patient` after extraction of five methods from `make_calculations`.

```
patient = Patient()
patient.take_patient_parameters()
print("calories recommended =", patient.daily_calories_recommended())
print("ideal weight =", patient.ideal_body_weight())
print("distance from ideal weight =", patient.distance_from_ideal_weight())
```

Figure 15. Getting results for a patient.

```
Class TestPatient(TestCase):
    def setUp(self):
        self.patient = Patient()
        self.patient.height = 170
        self.patient.weight = 60
        self.patient.age = 40
        self.patient.sex = 'female'

   def test_daily_calories_recommended_female(self):
        solution = self.patient.daily_calories_recommended_female()
        self.assertAlmostEqual(1301.5, solution)

   def test_ideal_body_weight_female(self):
        solution = self.patient.ideal_body_weight_female()
        self.assertAlmostEqual(61.7, solution)

    def test_distance_from_ideal_weight_female(self):
        solution = self.patient.distance_from_ideal_weight_female()
        self.assertAlmostEqual(-1.7, solution)

    def test_daily_calories_recommended(self):
        solution = self.patient.daily_calories_recommended()
        self.assertAlmostEqual(1301.5, solution)

    def test_ideal_body_weight(self):
        solution = self.patient.ideal_body_weight()
        self.assertAlmostEqual(61.7, solution)

    def test_distance_from_ideal_weight(self):
        solution = self.patient.distance_from_ideal_weight()
        self.assertAlmostEqual(-1.7, solution
```

Figure 16. Example of unit tests for some of the extracted methods.

We can see in Figure 14 that, unfortunately, similar `if..else` statements are scattered about the program in different places, namely they appear in methods `daily_calories_recommended`, `ideal_body_weight` and `distance_from_ideal_weight` (code smell Switch Statement). The `if..else` statement deals with the calculations for men and for women. In such case,

the inheritance should be applied with separate classes for male and female patients derived from class `Patient`. In Figure 17, class `Patient` is defined as an abstract one with two abstract methods `daily_calories_recommended` and `ideal_body_weight`, and one non-abstract method `distance_from_ideal_weight`. Subclasses `MalePatient` and `FemalePatient` implement in their own ways the abstract methods from `Patient`. Figure 18 presents the example of unit tests for the methods of the `FemalePatient` class.

```
class Patient(ABC):
    def __init__(self):
        self.height = None
        self.weight = None
        self.age = None

    def take_patient_parameters(self):
        self.height = float(input("patient height (cm): "))
        self.weight = float(input("patient weight (kg): "))
        self.age = float(input("patient age (years): "))

    @abstractmethod
    def daily_calories_recommended(self):
        pass

    @abstractmethod
    def ideal_body_weight(self):
        pass

    def distance_from_ideal_weight(self):
        return self.weight - self.ideal_body_weight()


class MalePatient(Patient):
    def daily_calories_recommended(self):
       return 10 * self.weight + 6.25 * self.height - 5 * self.age + 5

    def ideal_body_weight(self):
        return 50 + 0.9 * (self.height - 152)


class FemalePatient(Patient):
    def daily_calories_recommended(self):
       return 10 * self.weight + 6.25 * self.height - 5 * self.age - 161

    def ideal_body_weight(self):
        return 45.5 + 0.9 * (self.height - 152)
```

Figure 17. Inheritance hierarchy for patients.

```
class TestFemalePatient(TestCase):
    def setUp(self):
        self.patient = FemalePatient()
        self.patient.height = 170
        self.patient.weight = 60
        self.patient.age = 40

    def test_daily_calories_recommended(self):
        solution = self.patient.daily_calories_recommended()
        self.assertAlmostEqual(1301.5, solution)

    def test_ideal_body_weight(self):
        solution = self.patient.ideal_body_weight()
        self.assertAlmostEqual(61.7, solution)

    def test_distance_from_ideal_weight(self):
        solution = self.patient.distance_from_ideal_weight()
        self.assertAlmostEqual(-1.7, solution)
```

Figure 18. Example of unit tests for class `FemalePatient`.

4.4. Patient Creator

While working with patients, whether men or women, a client (i.e. an object of a class from outside the patient hierarchy) uses the interface defined in the `Patient` class. In order to get and show the information about a patient ideal weight, distance from the ideal weight and recommended calories, a client needs a patient object, but it does not need to know where this object is coming from (i.e. which subclass of `Patient` is used to generate it). Therefore, a design pattern called Parameterized Factory Method [13] can be used to provide a patient object, as shown in Figure 19. Method `create_a_patient` from the `PatientCeator` class provides an object either of `MalePatient` or of `FemalePatient` depending on the user choice. A client gets a patient from the patient creator, executes patient methods, and prints the obtained information and recommendations. The source codes of class `PatientCreator` and the client are shown in Figures 20 and 21, respectively. In Figure 21, we can see the polymorphic calls of methods `daily_calories_recommended`, `ideal_body_weight` and `distance_from_the ideal_weight`, namely different implementations of these methods will be executed depending on whether the patient is a man or a woman.

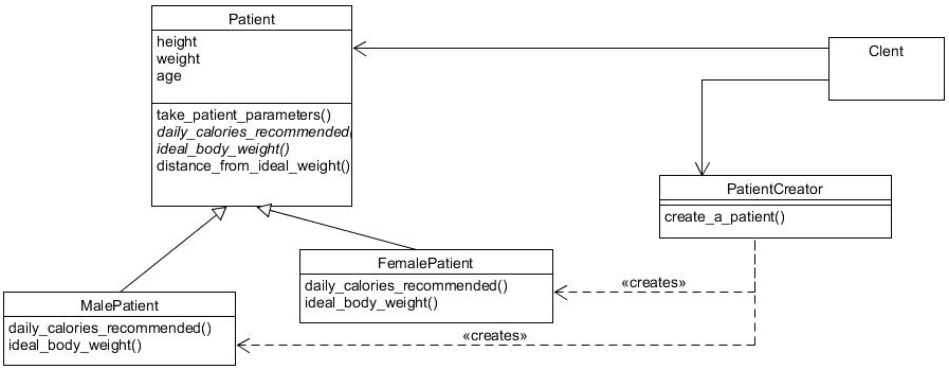

Figure 19. Final design using the Parameterized Factory Method pattern.

```
class PatientCreator:
    def create_a_patient(self):
        sex = input("patient sex (male/female): ")
        if sex == "female":
            return FemalePatient()
        else:
            return MalePatient()
```

Figure 20. Class `PatientCreator`.

```
patient_creator = PatientCreator()
patient = patient_creator.create_a_patient()
patient.take_patient_parameters()
print("calories recommended =", patient.daily_calories_recommended())
print("ideal weight =", patient.ideal_body_weight())
print("distance from ideal weight =", patient.distance_from_ideal_weight())
```

Figure 21. Client code.

5. Summary

The paper considered the problem of creating clean and easy to maintain code. Using a simple example of a Python program for calculating daily recommended calories, an ideal body weight, and a distance from the ideal weight for a patient, a code

refactoring process was shown step by step. In each step, the drawbacks of the code were discussed, bad smells indicated, and improvements proposed. In the final design, a Parameterized Factory Method design pattern was applied.

References

[1] W. Opdyke, *Refactoring object-oriented frameworks*, PhD Thesis 1992. http://www.laputan.org/pub/papers/opdyke-thesis.pdf. [20.11.2022].

[2] *William Opdyke*. https://en.wikipedia.org/wiki/William_Opdyke. [20.11.2022].

[3] G. Booch, *Object Oriented Analysis and Design.* Addison Wesley Longman, Inc. 1994.

[4] R. Half, *4 Advantages of Object-Oriented Programming*. https://www.roberthalf.com/blog/salaries-and-skills/4-advantages-of-object-oriented-programming. [20.11.2022].

[5] D.I.K. Sjøberg, *Managing Change in Information Systems: Technological Challenges*. https://www.duo.uio.no/bitstream/handle/10852/10038/DSjoberg-3.pdf?sequence=1. [20.11.2022].

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, don Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Inc. 1999.

[7] M. Fowler, *Refactoring.com.* https://refactoring.com/. [20.11.2022].

[8] M. Fowler, *CodeSmell*. https://martinfowler.com/bliki/CodeSmell.html. [20.11.2022].

[9] *Code Smells.* https://refactoring.guru/pl/refactoring/smells. [20.11.2022].

[10] D. Arsenovski, *Professional Refactoring in C# & ASP.NET*, Wiley Publishing, Inc., 2009

[11] *Basal metabolic rate*. https://en.wikipedia.org/wiki/Basal_metabolic_rate. [20.11.2022].

[12] *Human body weight*. https://en.wikipedia.org/wiki/Human_body_weight. [20.11.2022].

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.