Assignment 2 Solution

Shazil Arif

February 16, 2020

This report discusses the testing phase for the modules Set, ReactionT, CompoundT and MoleculeT. It also discusses the results of running the same tests on the partner files. The assignment specifications are then critiqued and various discussion questions are answered

1 Testing of the Original Program

Instead of writing one test case, executing it and then writing more and repeating the process, I took a different approach to testing. I wrote all my test cases for every module, covering all execution paths and then running the tests all together. This approach helped identify bugs not only within a certain method but also any issues with the integration of several methods/modules.

2 Results of Testing Partner's Code

My partner's code passed all of my test cases. In the first assignment, my partner's code failed for several test cases. Due to the nature of the first assignment and the design assumptions and choices that had to be made, failed test cases were expected. For this assignment there were no assumptions that had to be made by the developer as everything was made concise in the specification and thus it was expected that majority of test cases should pass. Furthermore, I did not find any problems with mine or my partners code since all tests passed

3 Critique of Given Design Specification

I genuinely really liked the formal specification given for this assignment. The main reason I enjoyed it was the mathematical and formal notation used to communicate clearly any

requirements, an invariant for a module, boolean conditions for exceptions etc. The inputs, outputs and desired behavior was very clear. However, due to the bulk of formal notation and less natural language it took significantly longer to fully interpret and understand the specification and specifically how each module depends and integrates with other modules. For example, it took me time to understand that certain modules were interfaces with generic/abstract methods that would have their unique implementation in a module that would inherit those properties. I did find some aspects of the design that could be changed. I would make the design more essential by removing certain redundant features. For example, the MolecSet and ElmSet modules are redundant since Python is dynamically typed and does not enforce the programmer to have the same types within a list. Simply using an instance of a Set object (Set refers to Set module defined in the context of this assignment and not a generic python set) of ElementT types is no different than using an ElmSet. The requirements are actually not opaque and do not satisfy information hiding. It is actually not stated anywhere whether state variables should be private or public. The state variables should be enforced to be kept as private making the specification opaque. I did not find any examples where the specification was did not meet other quality criteria including high cohesion, low coupling and consistency

4 Answers

a) Comparing the specification of the first assignment with this one, the major difference is the how the specification is communicated. Assignment 1 was described in natural language and Assignment 2 had a formal specification that used mathematical notation and symbols to communicate specifications, invariant and certain properties of methods.

   (a) Advantages of Natural language specification:

      i. General audience, easy to interpret natural language vs technical terminology and notation(formal specification)

      ii. Statements can be vague and not very concise regarding requirements, this allows the developer/programmer to make design choices and assumptions as necessary which encourages creativity and unique solutions

   (b) Disadvantages of Natural language specification:

      i. The vagueness of natural language description may encourage creativity and unique solutions but often opens the door for confusion and may result in a product that does not meet the requirements due to lack of understanding and communication.

ii. Often difficult to communicate technical ideas verbally, it can also be more challenging to interpret by the developer. For example "Function x takes an input y such that y is an integer less than or equal to 5" can be harder to interpret than "Function x takes a input y such that $y : \mathbb{Z} \leq x$"

(c) Advantages of a Formal specification

i. The requirements are clear and concise. Mathematical notation and symbols indicate exactly what is required. For example, a method's output can be described as the output of a mathematical function. Let's say a method computes lengths of sides of a triangle using the Pythagorean theorem and so the method's specification can be be described as "given a and b as parameters return $c = \sqrt{a^2 + b^2}$". This formal notation can be extended to explain certain properties or an invariant, for example "Method x(y) where y is a parameter containing a list of integers, should raise an error $\forall a \in y.a \geq 20$". The formal notation makes the requirements clear and avoids confusions among programmers leading to a higher quality end product.

(d) Disadvantages of a formal specification

i. Due to the technical nature of the notation, it can be difficult for certain developers to read and interpret. It requires extra effort and a basic background knowledge and familiarity with the notation and language.

b) The process of converting string to logical and meaningful syntactical components is called *Lexical Analysis*. To convert a string to a reaction in the context of this assignment and the ReactionT module, we need to supply ReactionT a left and right hand side of a chemical equation. Suppose the user enters a equation such as "2H2 + O2 -¿ 2H2O" as a string. To convert this to a reaction, we can split the string first by the "-¿" character and then inspect individual characters such as O and map them to ElementT types. We also need to take the numbers associated with the elements (the coefficients and subscripts) and construct MoleculeT objects, then a MolecSet, CompoundT and eventually the two sets of CompoundT objects to create a ReactionT object. The main secrets are substring search and/or regex splitting which can be implemented in a new Module that could be called StringReactionT.

c) I would create a new module that would contain a mapping between elements and a their atomic mass. Additionally, the element would be of type ElementT and mapped to a floating point value. This may be achieved using a dictionary that would look something like :

```
weights = { ElementT.H  : 1.007,
```

```
ElementT.He : 4.002
   }
```

This mapping can be easily used to retrieve the weight of ElementT type. Additionally, the constit_elements() method defined in both CompoundT and MoleculeT retrieves a list of elements contained in the compound/molecule which can be passed to a new method that will iteratively get the weights all of the elements, add them up and get the total weight of a molecule or compound. This is elegant because it does not require us to change any existing implementation and only add new code.

d) The usual convention in Chemistry is to have a chemical equation balanced with integer coefficients. The domain of Integer Programming addresses this problem. An algorithm that can compute coefficients as integers can be found *here*

e) In a dynamically typed language such as Python the types of variables are checked during execution, as the program runs. This means a programmer could write a statement such as

```
print("hello" + 5)
```

and Python would not indicate any errors and would execute. But, as the program executes Python will throw an error along the lines of not being able to convert a string to an integer since the types are clearly mismatched. Furthermore, if the above print statement was placed inside a if statement whose condition is never true, then Python will never throw an error.

In a statically typed language like Java the types of variables are checked before runtime. If a programmer writes a statement such as

```
System.out.println("hello" + 5)
```

Java will not even compile due to this type mismatch. If the print statement is placed inside a if statement that may never even execute, Java will still not compile due to the type mismatch.

Advantages of static typing are enforcing a coding standard. The strict type checking makes it harder to get past the compiler and as a result when the program executes, gives the programmer high confidence that they have done things correctly as opposed to a dynamically typed language where this confidence may be lower since some type

errors may never even be caught. Static typing also results in better performance at run time since the type of every variable does not need to be checked.

Disadvantages of static typing include frustration, constraints on a program and making simple programs unnecessarily complicated due to type checking. Some programmers may argue that constant errors showing up can be frustrating during the development process and leads to a decrease in productivity. Static typing places constraints on a program since it can only worth with a given type at a time. For example, a programmer may want to write a method to convert numbers to string and support this for both integers and floating point values. Unfortunately, due to static type checking the method can only take one type of value and do the conversion and a additional method must be defined to do the same process but for a different type of the input. This leads to the last point; making simple programs complicated. Consider the Quicksort sorting algorithm. We want to implement a generic quick sort that can be used on any comparable type such as strings and integers. In a dynamically typed language this is not a problem, we can have a function sort() that can take a list of any type. However, it is more complicated in a statically typed language such as Java. This is exactly why Java has *Generics*. In conclusion, the strict type checking leads to more and usually complicated code as opposed to dynamic typing.

f)
```
[(x,y) for x in range(1,11) for y in range(x,11) if x % 2 and y % 2 and x < y]
```

g)
```
def length(x):
        return sum(map(lambda n:1,x))
```

h) The interface can be thought of as a "contract" between a client and a programmer that may state the public methods in a class, their parameters and return values. The interface communicates to a client the necessary knowledge to use the methods in a module without having to worry about how the module works internally. The implementation is the underlying code, algorithms and secrets used to achieve the output stated in the interface of the module. For example, the interface of a math module may define:

```
def sin(x)
```

as a method that returns the sine of a value x in radians. A client can easily call this method and get the correct output values, they do not have to worry about how the value is being computed. In contrast, the implementation is the underlying algorithm/secret that is used to compute the sine of a value which is hidden from the client. The method may use a Taylor series approximation to compute sin(x) but a client does not to worry about this.

i) The following Software Engineering principles should guide the design of a module's interface as follows:

   (a) Abstraction

      i. Abstraction is the idea of "abstracting" unnecessary details of a software module interface from a client. The programmer should only provide the details necessary for a client to achieve their desired output. Abstraction should encourage the idea of focusing on the *interface* rather than *implementation*. This means state information should be encapsulated, the interface should not expose any details about internal algorithms/secrets (potentially from a security perspective for a sensitive application such as a password hashing module, but more importantly to hide these details from the client to avoid confusion).

   (b) Anticipation of Change

      i. Anticipation of change (AoC) addresses software design from a *long term and maintainable* perspective. Suppose we design a module that satisfied our clients needs but six months later, we need to implement new functionality/features. By AoC principle to implement these features we should be able to do so smoothly and elegantly. This means not a lot of existing code should be modified but instead new code should be able to be easily added and integrated. This encourages the design of software in *modules* and each module should address a different concern/sub problem and the module as a whole should have high cohesion. This also means that software should have low coupling since high coupling would imply changing one component of the code would not work without changing another component.

   (c) Generality

      i.

   (d) Modularity

      i. Modularity ties in with Anticipation of Change (AoC). It can be viewed as a consequence of AoC. As we discussed above AoC encourages software to be designed in module/smaller units to increase cohesion and achieve long term maintainability. This means we should have seperate classes that group together various methods used to achieve a common task. For example, in a Chemical equations software we should have a module that addresses balancing of equations and a seperate module that addresses the representation of an equation. These two modules would then have their own methods that achieve the sub problem that they are addressing (a balancing method in the balancing module)

(e) Separation of concerns

i. Seperation of conerns (SoC) specifies the need for humans to work with little information at any given time. Humans have a difficult time understanding all the complex and technical details to design a software system. To avoid confusion and enable a person to be able to work on a software, we should work with small bits of information/work with smaller sub problems at a time. This encourges the design of software to be done in modules (enabling Modularity)

E Code for ChemTypes.py

```
## @file ChemTypes.py
#  @author Shazil Arif
#  @details ElementT contains an enumeration for elements on the periodic table
#  @Date Feb 8th, 2020

## @brief ElementT contains an enumeration for elements on the periodic table


class ElementT():
    [H,
     He,
     Li,
     Be,
     B,
     C,
     N,
     O,
     F,
     Ne,
     Na,
     Mg,
     Al,
     Si,
     P,
     S,
     Cl,
     Ar,
     K,
     Ca,
     Sc,
     Ti,
     V,
     Cr,
     Mn,
     Fe,
     Co,
     Ni,
     Cu,
     Zn,
     Ga,
     Ge,
     As,
     Se,
     Br,
     Kr,
     Rb,
     Sr,
     Y,
     Zr,
     Nb,
     Mo,
     Tc,
     Ru,
     Rh,
     Pd,
     Ag,
     Cd,
     In,
     Sn,
     Sb,
     Te,
     I,
     Xe,
     Cs,
     Ba,
     La,
     Ce,
     Pr,
     Nd,
     Pm,
     Sm,
     Eu,
     Gd,
     Tb,
     Dy,
     Ho,
```

```
Er,
Tm,
Yb,
Lu,
Hf,
Ta,
W,
Re,
Os,
Ir,
Pt,
Au,
Hg,
Tl,
Pb,
Bi,
Po,
At,
Rn,
Fr,
Ra,
Ac,
Th,
Pa,
U,
Np,
Pu,
Am,
Cm,
Bk,
Cf,
Es,
Fm,
Md,
No,
Lr,
Rf,
Db,
Sg,
Bh,
Hs,
Mt,
Ds,
Rg,
Cn,
Nh,
Fl,
Mc,
Lv,
Ts,
Og] = range(1,
                119)
```

F Code for ChemEntity.py

```
## @file ChemEntity.py
#  @author Shazil Arif
#  @brief ChemEntity is a sub module used to build equation balancer
#  @date Feb 8th 2020


from abc import ABC, abstractmethod

## @brief ChemEntity is a class with abstract methods
## @details these methods are meant to be overriden by other modules


class ChemEntity(ABC):

    @abstractmethod
    ## @brief a generic method for counting the number of atoms
    #  @param elm the element to count the number of atoms for
    #  @return an integer indicating the number of atoms
    def num_atoms(self, elm):
        pass

    @abstractmethod
    ## @brief a generic method for getting the elements
    #  @details can be inherited and overriden to specific modules
    #  @return an ElmSet of elements
    def constit_elems(self):
        pass
```

G Code for Equality.py

```
## @file Equality.py
#  @author Shazil Arif
#  @date Feb 8th, 2020

from abc import ABC, abstractmethod


## @brief Equality contains a generic equals methods

class Equality(ABC):

    @abstractmethod
    ## @brief a generic method to compare two values/objects
    ## @param t any type
    ## @return boolean indicating results of comparison
    def equals(self, t):
        pass
```

H Code for Set.py

```
##  @file Set.py
#  @author Shazil Arif
#  @brief Set.py contains a class that implements a set data type
#  @date Feb 8th, 2020

from Equality import *

## @brief Set is a class that implements a Date object containing a year, month and a day


class Set(Equality):

    ## @brief constructor method for class Set, intializes a Set from a given sequence t
    #  @param t a sequence of values that will be converted a set
    def __init__(self, t):
        # source:
        #
            http://www.martinbroadhurst.com/removing-duplicates-from-a-list-while-preserving-order-in-python.html
        seen = set()
        self._S = [x for x in t if not (x in seen or seen.add(x))]

    ## @brief add a new element to the set
    #  @param e The element to add to the set
    def add(self, e):
        if(e not in self._S):
            self._S.append(e)

    ## @brief remove a element from the set
    #  @param e The element to remove from the set
    #  @throws ValueError if parameter e is not a member of the set
    def rm(self, e):
        if (e in self._S):
            self._S.remove(e)
        else:
            raise ValueError("{elm} is not a member".format(elm=e))

    ## @brief check if an element is in the set
    #  @param e The element to add to the set
    #  @return boolean value indicating whether parameter e was found in the set
    def member(self, e):
        return e in self._S

    ## @brief return size of the set
    #  @return integer representing the size of the set
    def size(self):
        return len(self._S)

    ## @brief check if two sets are equal
    #  @details two sets are considered equal if they have same size and elements
    #  @param r the set to compare against
    #  @return boolean indicating if the sets are equal
    def equals(self, r):
        if(r.size() != self.size()):
            return False
        temp_set = r.to_seq()
        for element in self._S:
            if (element not in temp_set):
                return False
        return True

    ## @brief convert the set to a sequence
    #  @return a sequence containing all elements of the set
    def to_seq(self):
        return self._S
```

I Code for ElmSet.py

```
## @file ElmSet.py
#  @author Shazil Arif
#  @brief Represents a Set of type ElementT found in ChemEntity.py
#  @date Feb 8th, 2020

from Set import *

## @brief ElmSet is a set of Elements from ElementT
#  @extends from Set.py


class ElmSet(Set):
    pass
```

J Code for MolecSet.py

```
## @file MolecSet.py
#  @author Shazil Arif
#  @brief MolecSet is a sub module used to build a Equation balancer
#  @date Feb 8th, 2020

from Set import *

## @brief MolecSet is a set of MoleculeT objects
#  @details extends from Set.py


class MolecSet(Set):

    ## @brief compare two MolecSets
    #  @details Overrides equals defined in Set.py
    #  @param other The other MolecSet to compare against
    #  @return Boolean indicating if two sets are equal
    def equals(self, other):
        if(self.size() != other.size()):
            return False

        self_obj = self.to_seq()
        other_obj = other.to_seq()

        for i in range(len(other_obj)):
            if(not other_obj[i].__dict__ == self_obj[i].__dict__):
                return False
        return True
```

K Code for CompoundT.py

```
## @file CompoundT.py
#  @author Shazil
#  @brief CompoundT is a sub module used to build a chemical equation balancer
#  @date February 8th, 2020


from MoleculeT import *
from MolecSet import *

## @brief CompoundT is used to represent a chemical compound


class CompoundT(MoleculeT):
    ## @brief constructor for class CompoundT
    #  @param molec_set A MolecSet Object
    def __init__(self, molec_set):
        self._C = MolecSet(molec_set.to_seq())

    ## @brief return the molecules in the compound
    #  @return a Set containing the molecules in the compound
    def get_molec_set(self):
        return self._C

    ## @brief count the number of atoms of a element in the compound
    #  @param e the element to check for in the compound
    #  @return integer indicating the number of atoms of element e in the compound
    def num_atoms(self, e):
        temp_seq = self._C.to_seq()
        count = 0
        for molecule in temp_seq:
            count += molecule.num_atoms(e)
        return count

    ## @brief return an ElmSet of the elements in the molecules that are in the compound
    #  @return ElmSet containing the elements
    def constit_elems(self):
        molecs = self._C.to_seq()
        elems = []
        for molecule in molecs:
            elems.append(molecule.get_elm())
        return ElmSet(elems)

    ## @brief check if two compounds are equals
    #  @param d the object to compare against
    #  @return boolean indicating if they are equal
    def equals(self, d):
        return self.get_molec_set().equals(d.get_molec_set())
```

L Code for ReactionT.py

```
## @file ReactionT.py
#  @author Shazil Arif
#  @brief ReactionT is responsible for balancing equations
#  @date Feb 8th, 2020

from CompoundT import *
from ElmSet import ElmSet
import numpy as np


## @brief ReactionT is responsible for balancing equations
#  @details extends from CompoundT

class ReactionT(CompoundT):

    ## @brief constructor method for ReactionT
    #  @details the chemical equation is balanced in the constructor
    #  @param l a sequence of compounds on the left side of the equation
    #  @param r a sequence of compounds on the right side of the equation
    #  @throws ValueError if the equation cannot be balanced
    def __init__(self, l, r):
        self._lhs = l
        self._rhs = r
        self._coeff_l = []
        self._coeff_r = []
        self.__balance()

    ## @brief getter method for the Compounds on the left side of reaction
    #  @return a sequence of CompoundT
    def get_lhs(self):
        return self._lhs

    ## @brief getter method for the Compounds on the right side reaction
    #  @return a sequence of CompoundT
    def get_rhs(self):
        return self._rhs

    ## @brief getter method for the coefficients of compounds on the left side
    #  @details indicates the coefficient of the compounds retrieved from get_lhs()
    #  @return a sequence of real numbers indicating the coefficents
    def get_lhs_coeff(self):
        return self._coeff_l

    ## @brief getter method for the coefficients of compounds on the right side
    #  @details indicates the coefficient of the compounds retrieved from get_rhs()
    #  @return a sequence of real numbers indicating the coefficents
    def get_rhs_coeff(self):
        return self._coeff_r

    ## @brief setter method for the coefficients on the left side
    #  @details private method, coefficients are not to be modified by a client
    #  @param coeff The list of coefficient values to assign
    def __set_lhs_coeff(self, coeff):
        self._coeff_l = coeff

    ## @brief setter method for the coefficients on the right side
    #  @details private method, coefficients are not to be modified by a client
    #  @param coeff The list of coefficient values to assign
    def __set_rhs_coeff(self, coeff):
        self._coeff_r = coeff

    ## @brief balanc method for the coefficients on the right side
    def __balance(self):
        # get all elements in reaction
        elm_set = self.__elm_in_chem_eq(self.get_lhs())
        matrix = []
        index = 0

        # source:
        #
            https://stackoverflow.com/questions/45220032/how-to-balance-a-chemical-equation-in-python-2-7-using-matrices
        # iterate over all elements
        for elm in elm_set.to_seq():
            # append new row
            matrix.append([])
```

```
            # iterate over left and right, totalling the count for each element
            for compound in self.get_lhs():
                matrix[index].append(compound.num_atoms(elm))

            # add the negative of the count on right
            # can be though of as "subtracting from both sides"
            for compound in self.get_rhs():
                matrix[index].append(-compound.num_atoms(elm))

            index += 1

        # Since we are solving the system Ax=B
        # here we construct the B vector by taking the first row of the matrix
        b_vector = []
        for i in range(len(matrix)):
            b_vector.append(-matrix[i][0])

            # remove from matrix after putting in b vector
            matrix[i].pop(0)

        # solve the system using numpy
        coeffs = np.linalg.lstsq(matrix, b_vector, rcond=-1)[0]
        # append 1 to start since first row was set to 1
        coeffs = np.append([1], coeffs)

        # check if coefficients were all positive
        if(self.__pos(coeffs)):
            # take subarrays corresponding to each side of the equation
            self.__set_lhs_coeff(list(coeffs[0:len(self.get_lhs())]))
            self.__set_rhs_coeff(list(coeffs[len(self.get_lhs()):len(coeffs)]))
        else:
            raise ValueError("Unable to balance")

    ## @brief check if all values in a sequence are positive
    #  @param seq the Sequence to check
    #  @return boolean indicating the result
    def __pos(self, seq):
        for i in seq:
            if (i <= 0):
                return False
        return True

    ## @brief Get all elements in a sequence of compounds
    #  @param Sequence of CompoundT objects
    #  @return Elmset containing the elements
    def __elm_in_chem_eq(self, seq_compounds):
        elms = []
        for i in seq_compounds:
            temp = i.constit_elems().to_seq()
            elms = elms + temp
        return ElmSet(elms)
```

M Code for test_All.py

```
## @file test_All.py
#  @author Shazil
#  @brief test_All.py is used to test several modules used to balance chemical equations
#  @date Feb 8th 2020

import pytest
from MoleculeT import MoleculeT
from ChemTypes import ElementT
from Set import Set
from ElmSet import ElmSet
from CompoundT import CompoundT
from MolecSet import MolecSet
from ReactionT import ReactionT

## @brief Test methods from Set.py


class TestSetADT:

    # initialize an instance of Set for each test
    def setup_method(self, method):
        self.test_list = [1, 2, 3, 4, 5, 6, 7]
        self.test_set = Set(self.test_list)

    # reset state variables
    def teardown_method(self, method):
        self.test_set = None
        self.test_list = None

    # test to_seq() method
    def test_to_seq(self):
        assert self.test_set.to_seq() == self.test_list

    # test add method
    def test_add_with_new_element(self):
        self.test_set.add(8)
        assert 8 in self.test_set.to_seq()

    # test add method with existing element
    def test_add_with_existing_element(self):
        self.test_set.add(2)

        # check if 2 in set and it should occur only once
        assert 2 in self.test_set.to_seq() and self.test_set.to_seq().count(2) == 1

    # test member method, it should return True
    def test_member_with_existing_element(self):
        assert self.test_set.member(1)

    def test_member_with_non_existing_element(self):
        assert not self.test_set.member(0)

    def test_remove_method_exception(self):
        with pytest.raises(ValueError):
            self.test_set.rm(max(self.test_list) + 1)

    def test_remove_method(self):
        # assume we are blackbox testing, remove arbitrary values
        removed_element = max(self.test_list)
        self.test_set.rm(removed_element)
        assert removed_element not in self.test_set.to_seq()

    def test_size(self):
        assert self.test_set.size() == len(self.test_list)

    # test size method with a zero size
    def test_size_zero(self):
        test = Set([])
        assert test.size() == 0

    def test_equals_with_different_size_sets(self):
        test = Set([1, 2])
        assert not self.test_set.equals(test)

    def test_equals_with_same_size_sets(self):
        test = self.test_set
```

```
        assert self.test_set.equals(test)

    def test_equals_with_nonequal_sets(self):
        test = Set([])
        length = len(self.test_set.to_seq())
        for i in range(0, length):
            # some arbitrary values not equal to those in test_set
            test.add(i * -1)
        assert not self.test_set.equals(test)

# @brief test MoleculeT.py


class TestMoleculeT:
    def setup_method(self, method):
        self.elm_num = 2
        self.elm = ElementT.H
        self.molecule = MoleculeT(self.elm_num, self.elm)

    # reset state variables
    def teardown_method(self, method):
        self.elm = None
        self.elm_num = None
        self.molecule = None

    def test_get_num(self):
        assert self.molecule.get_num() == self.elm_num

    def test_get_elm(self):
        assert self.molecule.get_elm() == self.elm

    def test_num_atoms(self):
        assert self.molecule.num_atoms(self.elm) == self.elm_num

    def test_num_atom_with_wrong_element(self):
        # add one to self.elm to test with arbitrary element
        # main idea is to use blackbox approach and minimize hardcoding
        assert self.molecule.num_atoms(self.elm + 1) == 0

    def test_constit_elems(self):
        assert self.molecule.constit_elems().equals(ElmSet([self.elm]))

    def test_equals_with_same_molecule(self):
        test_molec = MoleculeT(self.elm_num, self.elm)
        assert self.molecule.equals(test_molec)

    def test_equals_with_different_molecule(self):
        test_molec = MoleculeT(self.elm_num + 1, self.elm)
        assert not self.molecule.equals(test_molec)

# @brief Test CompoundT


class TestCompoundT:
    # state variables to be used for all tests
    def setup_method(self, method):
        self.elm_num = 2
        self.elm = ElementT.H
        self.molecule = MoleculeT(self.elm_num, self.elm)
        self.molecule_two = MoleculeT(self.elm_num + 1, self.elm + 1)
        self.molec_set = MolecSet([self.molecule, self.molecule_two])
        self.compound = CompoundT(self.molec_set)

    # reset state variables
    def teardown_method(self, method):
        self.elm_num = None
        self.elm = None
        self.molecule = None
        self.molecule_two = None
        self.molec_set = None
        self.compound = None

    def test_get_molec_set(self):
        assert self.compound.get_molec_set().equals(self.molec_set)

    def test_num_atoms(self):
        assert self.compound.num_atoms(self.molecule.get_elm()) == self.molecule.get_num()

    def test_constit_elems(self):
        assert self.compound.constit_elems().equals(
```

```
            ElmSet([self.molecule.get_elm(), self.molecule_two.get_elm()]))

    def test_equals(self):
        test_compound = CompoundT(self.molec_set)
        assert self.compound.equals(test_compound)

# @brief Test ReactionT


class TestReactionT:
    def setup_method(self):
        '''
        Test the equation that looks like

        (a)NaOH + (b)H2SO4 -> (x)Na2SO4 + (y)H2O
        Balanced equation is :
        NaOH + (0.5)H2SO4 -> (0.5)Na2SO4 + H2O

        '''

        # create sodium hydroxide (NaOH - left side)
        left_h = MoleculeT(1, ElementT.H)
        left_na = MoleculeT(1, ElementT.Na)
        left_o = MoleculeT(1, ElementT.O)
        sodium_hydroxide = CompoundT(MolecSet([left_na, left_o, left_h]))

        # create Sulfurice Acid (H2SO4 - left side)
        left_h2 = MoleculeT(2, ElementT.H)
        left_sulfur = MoleculeT(1, ElementT.S)
        left_o4 = MoleculeT(4, ElementT.O)
        sulfuric_acid = CompoundT(MolecSet([left_h2, left_sulfur, left_o4]))

        # create Sodium sulfate (Na2SO4 - right side)
        right_na2 = MoleculeT(2, ElementT.Na)
        right_sulfur = MoleculeT(1, ElementT.S)
        right_o4 = MoleculeT(4, ElementT.O)
        sodium_sulfate = CompoundT(MolecSet([right_na2, right_sulfur, right_o4]))

        # create water (H2O - right side)
        right_h2 = MoleculeT(2, ElementT.H)
        right_o = MoleculeT(1, ElementT.O)
        water = CompoundT(MolecSet([right_h2, right_o]))

        # create Reaction
        self.left = [sodium_hydroxide, sulfuric_acid]
        self.right = [sodium_sulfate, water]
        self.left_coeffs = [1, 0.5]
        self.right_coeffs = [0.5, 1]
        self.reaction = ReactionT(self.left, self.right)

    def teardown_method(self):
        self.left = None
        self.right = None
        self.left_coeffs = None
        self.right_coeffs = None
        self.reaction = None

    def tet_get_lhs(self):
        assert self.is_equal_array(self.reaction.get_lhs(), self.left)

    def test_get_rhs(self):
        assert self.is_equal_array(self.reaction.get_rhs(), self.right)

    def test_get_lhs_coeff(self):
        assert self.is_equal_numbers_array(self.reaction.get_lhs_coeff(), self.left_coeffs)

    def test_get_rhs_coeff(self):
        assert self.is_equal_numbers_array(self.reaction.get_rhs_coeff(), self.right_coeffs)

    # utility function
    def is_equal_array(self, one, two):
        for i in range(len(one)):
            if(not one[i].equals(two[i])):
                return False
        return True

    # utility function
    # admit a tolerance of 0.1
    def is_equal_numbers_array(self, one, two):
        for i in range(len(one)):
```

```
    if(abs(one[i] - two[i]) > 0.1):
        return False
return True
```

N Code for Partner's Set.py

```
## @file Set.py
#  @author Amir Afzali
#  @title Set
#  @date February 8, 2020
from Equality import *

## @brief An ADT for holding sets of generic elements
#  @details This class implements the equality abstract class.


class Set(Equality):

    ## @brief Constructor for Set
    #  @details Constructor accepts one parameter, a list of any data type
    #  If the passed objects data type is not a list, an empty array is initialized
    #  @param s: Intial list of values
    def __init__(self, s: list):
        if type(s) == list:
            self.__S = list(set(s))
        else:
            self.__S = []

    def __eq__(self, R: 'Set'):
        return self.equals(R)

    ## @brief Adds some passed element to the set
    #  @details If the element is already in the set, do nothing
    #  @param e: Any passed item
    def add(self, e) -> None:
        if not self.member(e):
            self.__S.append(e)

    ## @brief Removes some passed element to the set
    #  @details If the element is not in the set, a ValueError exception is raised
    #  @param e: Any passed item
    def rm(self, e) -> None:
        self.__S.remove(e)

    ## @brief Returns if some passed element is in the set
    #  @param e: Any passed item
    #  @return bool True if element is in set
    def member(self, e) -> bool:
        return e in self.__S

    ## @brief Returns the size of the set
    #  @return int length of set
    def size(self) -> int:
        return len(self.__S)

    ## @brief Returns if some passed Set is 'equal' to the current set
    #  @details Two sets are considered equal if they are of equal length
    #  and all elements in the current set exist in the passed set
    #  @param R: Set object
    #  @return bool True if the two sets are equal
    def equals(self, R: 'Set') -> bool:
        if(R.size() != self.size()):
            return False
        return all([x in R.to_seq() for x in self.__S])

    ## @brief Returns the set as a list sequence
    #  @return list representation of object's set
    def to_seq(self) -> list:
        return self.__S
```

O Code for Partner's MoleculeT.py

```
## @file MoleculeT.py
#  @author Amir Afzali
#  @title MoleculeT
#  @date February 8, 2020
from ElmSet import ElmSet
from ChemTypes import ElementT
from ChemEntity import ChemEntity
from Equality import Equality

## @brief An ADT for representing chemical molecules
#  @details This class is allows for creating and accessing properties on
#  chemical molecules. This class implements the ChemEntity and Equality abstract classes.


class MoleculeT(ChemEntity, Equality):

    ## @brief Constructor for MoleculeT
    #  @details Constructor accepts two parameters: int n and ElementT e
    #  @param n: int representing amount of the element
    #  @param e: ElementT representing the element of which the molecule consists of
    def __init__(self, n: int, e: ElementT):
        self.__num = n
        self.__elm = e

    ## @brief Getter method for the molecule num property
    #  @return int n for the object's num property
    def get_num(self) -> int:
        return self.__num

    ## @brief Getter method for the molecule elm property
    #  @return ElementT m for the object's elm property
    def get_elm(self) -> ElementT:
        return self.__elm

    ## @brief Returns the number of atoms of some ElementT contained in the molecule
    #  @details This methods takes some ElementT and determines the number of atoms
    #  of that element within the MoleculeT. In other words, if e is the object's elm,
    #  return the num property. Otherwise return 0.
    #  @return int number of atoms
    def num_atoms(self, e) -> int:
        return self.get_num() if e == self.get_elm() else 0

    ## @brief Returns an ElmSet of consisting of the object's elm.
    #  @return ElmSet of current Molecule's element
    def constit_elems(self) -> ElmSet:
        return ElmSet([self.get_elm()])

    ## @brief Returns if the current MoleculeT object is equal to the passed MoleculeT
    #  @details Equality is determined if the two object's have the same elm property
    #  and the same num property. That is to say, they are the same number of the same elm.
    #  @param m: MoleculeT to be compared
    #  @return bool with True if both molecules are equal
    def equals(self, m) -> bool:
        return m.get_elm() == self.get_elm() and m.get_num() == self.get_num()
```

P Code for Partner's CompoundT.py

```
## @file CompoundT.py
#  @author Amir Afzali
#  @title CompoundT
#  @date February 8, 2020
from Equality import Equality
from ChemEntity import ChemEntity
from ElmSet import ElmSet
from MolecSet import MolecSet

## @brief An ADT for representing chemical compounds
#  @details This class is allows for creating and accessing properties on
#  chemical compounds. This class implements the ChemEntity and Equality abstract classes.


class CompoundT(ChemEntity, Equality):

    ## @brief Constructor for CompoundT
    #  @details Constructor accepts one parameter, a MolecSet
    #  @param M: MolecSet representing the compound molecules
    def __init__(self, M: MolecSet):
        self.__C = M

    ## @brief Getter method for the compound MolecSet
    #  @return MolecSet for the object's MolecSet property
    def get_molec_set(self) -> MolecSet:
        return self.__C

    ## @brief Returns the number of atoms of some ElementT contained in the compound
    #  @details This methods takes some ElementT and determines the number of atoms
    #  of that element within the CompoundT MolecSet.
    #  @return int number of atoms
    def num_atoms(self, e) -> int:
        sum = 0
        for molec in self.get_molec_set().to_seq():
            if molec.get_elm() == e:
                sum += molec.get_num()
        return sum

    ## @brief Returns an ElmSet of all ElementT's contained in the CompoundT MolecSet
    #  @return ElmSet of unique ElementT in MolecSet
    def constit_elems(self) -> ElmSet:
        return ElmSet([x.get_elm() for x in self.get_molec_set().to_seq()])

    ## @brief Returns if the current CompoundT object is equal to the passed CompoundT
    #  @details Equality is determined if the two object's MolecSets are equal
    #  @param D: CompoundT to be compared
    #  @return bool with True if both compounds are equal
    def equals(self, D: 'CompoundT') -> bool:
        c_set = self.get_molec_set()
        d_set = D.get_molec_set()
        return c_set.equals(d_set)
```

Q Code for Partner's ReactionT.py

```
## @file ReactionT.py
#  @author Amir Afzali
#  @title ReactionT
#  @date February 8, 2020
from typing import List
from CompoundT import CompoundT
from ChemTypes import ElementT
from ElmSet import ElmSet
from numpy import linalg

## @brief An ADT for representing chemical reactions
#  @details This class is allows for creating and balancing chemical reactions


class ReactionT:

    ## @brief Constructor for ReactionT
    #  @details Constructor accepts two parameters: Sequence of CompoundT for left
    #  and right side of chemical reaction. Constructor will balance the reaction if
    #  possible, otherwise a ValueError is thrown
    #  @param L: CompoundT sequence representing compounds in left side of reaction
    #  @param R: CompoundT sequence representing compounds in right side of reaction
    def __init__(self, L: List[CompoundT], R: List[CompoundT]):
        self.__lhs = L
        self.__rhs = R
        self.__coeffL = []
        self.__coeffR = []

        self.__balance__()

        if not (self.__is_balanced__(L, R, self.get_lhs_coeff(), self.get_rhs_coeff())
                and self.__pos__(self.get_lhs_coeff())
                and self.__pos__(self.get_rhs_coeff())):
            raise ValueError('Cannot balance equation')

    ## @brief Getter method for the reaction left side compounds
    #  @return List of CompoundT
    def get_lhs(self) -> List[CompoundT]:
        return self.__lhs

    ## @brief Getter method for the reaction right side compounds
    #  @return List of CompoundT
    def get_rhs(self) -> List[CompoundT]:
        return self.__rhs

    ## @brief Getter method for the reaction left side coefficients
    #  @return List of float coefficient
    def get_lhs_coeff(self) -> list:
        return self.__coeffL

    ## @brief Getter method for the reaction right side coefficients
    #  @return List of float coefficient
    def get_rhs_coeff(self) -> list:
        return self.__coeffR

    def __balance__(self):
        L, R = self.get_lhs(), self.get_rhs()

        ls = self.__populate__(L[0:1])
        mid = self.__populate__(L[1:], True)
        rs = mid + self.__populate__(R)

        full = self.__elm_in_chem_eq__(L).to_seq()

        coeff1 = self.__get_coeffs__(full, ls)
        coeff2 = self.__get_coeffs__(full, rs)
        coeff1 = [j for sub in coeff1 for j in sub]
        coeffs = linalg.lstsq(coeff2, coeff1, -1)

        final_coeffs = list(map(lambda x: round(x, 4), coeffs[0].tolist()))

        self.__coeffL = [1] + final_coeffs[:len(L) - 1]
        self.__coeffR = final_coeffs[len(L) - 1:]

    def __populate__(self, L: List[CompoundT], neg=False) -> List[dict]:
        ls = []
```

```
    for i in range(len(L)):
        ls.append({})
        for molec in L[i].get_molec_set().to_seq():
            ls[i][molec.get_elm()] = molec.get_num() if not neg else molec.get_num() * -1
    return ls

def __get_coeffs__(self, scope, side) -> list:
    coefficients = []
    for i, elm in enumerate(scope):
        coefficients.append([])
        for compound in side:
            toPush = compound[elm] if elm in compound else 0
            coefficients[i].append(toPush)
    return coefficients

def __pos__(self, s: list) -> bool:
    print(s)
    return all(x >= 0 for x in s)

def __n_atoms__(self, C: List[CompoundT], c: list, e: ElementT) -> int:
    sum = 0
    for i in range(len(C)):
        sum = sum + c[i] * C[i].num_atoms(e)
    return sum

def __elm_in_chem_eq__(self, C: List[CompoundT]):
    elemArr = []
    for c in C:
        elemArr = elemArr + c.constit_elems().to_seq()
    return ElmSet(elemArr)

def __is_bal_elm__(self, L: List[CompoundT], R: List[CompoundT], cL: list, cR: list, e):
    return self.__n_atoms__(L, cL, e) == self.__n_atoms__(R, cR, e)

def __is_balanced__(self, L: List[CompoundT], R: List[CompoundT], cL: list, cR: list):
    eInChem = self.__elm_in_chem_eq__
    isBal = self.__is_bal_elm__

    firstCond = eInChem(L).equals(eInChem(R))
    secondCond = all([isBal(L, R, cL, cR, e) for e in eInChem(L).to_seq()])
    return firstCond and secondCond
```