שאלה 1.1:

הפתרון הפשוט והאלגנטי – הוא להשתמש בפתרון של 1.2, אבל גם פתרון של שכפול קוד - התקבל

```
public boolean isSymmetric2() {
    return allNoneSymmetric().isEmpty();
}
```

```
public boolean isSymmetric() {
    boolean ans = true;
    Iterator<node_data> v_iter = this.getV().iterator();
    while(v_iter.hasNext()) {
        int id = v_iter.next().getKey();
        Iterator<edge_data> e_iter = this.getE(id).iterator();
        while(ans & e_iter.hasNext()) {
            edge_data e = e_iter.next();
            edge_data e1 = this.getEdge(e.getDest(),e.getSrc());
            if(e1==null || e.getWeight()!=e1.getWeight()) {ans = false;}
        }
    }
    return ans;
}
```

שאלה 1.2:

```
public ArrayList<edge_data> allNoneSymmetric() {
    ArrayList<edge_data> ans = new ArrayList<edge_data>();
    Iterator<node_data> v_iter = this.getV().iterator();
    while(v_iter.hasNext()) {
        int id = v_iter.next().getKey();
        Iterator<edge_data> e_iter = this.getE(id).iterator();
        while(e_iter.hasNext()) {
            edge_data e = e_iter.next();
            edge_data e1 = this.getEdge(e.getDest(),e.getSrc());
            if(e1==null || e.getWeight()!=e1.getWeight()) {ans.add(e);}
        }
    }
    return ans;
}
```

שאלה 2.1:

הפתרון של "הסטודנט", הוא לא נכון מבחינת חלוקה למחלקות, בפרט הפתרון של לשים את כל הלוגיקה הממשקים הגרפיים והעבודה מול השרת באותה מחלקה אינו עומד בהגדרה של תבנית העיצוב של **MVC**. בתבנית העיצוב של **MVC**, נבצע חלוקה בין הלוגיקה (**Model** - אלגוריתמיקה), תצוגה גרפית (**View** – ממשק גרפי), כאשר ה **Controller** הוא הממשק בין שכבת האלגוריתמיקה לבין התצוגה.
את הפתרון של "צד הלקוח" נשפר באופן הבא: נפריד לשלוש חבילות נפרדות: חבילה שעוסקת אך ורק בתצוגה (מודעת למערכת השליטה) חבילה שעוסקת באלגוריתמיקה: כוללת מחלקות למבני נתונים, אלגוריתמים של הזזת הסוכנים וכו'. חבילה שמחברת בין האלגוריתמיקה ומערכת התצוגה – בפרט כוללת מחלקה מרכזית שיודעת לעבוד מול השרת, ולהתייחס לאירועים ולפי הצורך להפעיל את האלגוריתמיקה ולשלוח את המידע לשכבת התצוגה.

שאלה 2.2.

כיוון שכל בקשה של "צד הלקוח" מ"השרת" עוברת השהייה של 100-500 אלפיות שנייה, קצב התשובות שהלקוח שלנו יקבל עלול להיות בין 2-10 תגובות בשנייה (6 תגובות בממוצע – ללא התייחסות למשך ביצוע הפעולה בשרת). לפיכך נצטרך להריץ ב"צד הלקוח" מספר תהליכים (בגישת **none-blocking**) בפרט אם יהיו לנו בצד הלקוח 5 תהליכים שונים (לפחות) שכולים יפנו לשרת אזי גם אם ההשהייה היא חצי שנייה (500 אלפיות) נקבל שקצב התשובות מהשרת יהיו לפחות 10 פעמים בשנייה.

שאלה 3.1:

שיטה לעדכון משקלות הגרף: משך הזמן (האמיתי) שלוקח לעבור כל "מקטע" – כל צלע:
1. כל עדכון מלקוח כולל: זמן, לקוח, מיקום, ומהירות (על בסיס המסלול לייעד).
2. השרת ישייך כל עדכון שנשלח מלקוח לשרת לצלע (המכוונת) רלוונטית בגרף התנועה בשרת.
3. כל צלע בגרף תכלול "תור" של עדכונים – לפי זמן (נניח רק העדכונים מה 10 דקות האחרונים ישמרו בתור – בהנחה שיש מספיק עדכונים בתור).
4. בכל "תור" נחשב את הממוצע של המהירות (אפשר לסנן חריגים לפי סטיית תקן, או מרחק מערך חציוני)
5. הגרף יעודכן בהתאם לערך המהירות הממוצעת – ובהתאם יעודכן המסלול וזמן הגעה ליעד לכל לקוח.

שאלה 3.2:

הרעיון הכללי הוא להשתמש בתבנית העיצוב של **observer**

1. לקוח (מנווט) שולח לשרת בקשה לחישוב מסלול (ממיקומו הנוכחי) ליעד.
2. "השרת" מחשב ללקוח מסלול מומלץ וזמן הגעה משוער, וכן רושם אותו ואת מאפייני המסלול שנשלח אליו. במילים אחרות: הלקוח נרשם אצל השרת במנגנון **observer**, או **callback**)
3. השרת שכולל גרף תנועה – שמתעדכן מדי פעם (לדוגמא כמו בסעיף 3.1).
4. בהינתן עדכון בגרף התנועה בשרת – השרת יבדוק האם השינוי בגרף גורם לשינוי משמעותי במסלול או בתזמון של כל אחד מהלקוחות:
5. במקרה של שינוי משמעותי במסלול או בתזמון הצפוי של לקוח מסויים: יפנה השרת ללקוח הרלוונטי ויעדכן אותו בשינוי (**callback**).

שאלה 4.1:

```python
class Planet:
    def __init__(self, planet_name: str, pos: Tuple[float, float, float]):
        """
        :param planet_name: the name of the planet
        :param pos: the position of the planet(x,y,z)
        """
        self.name = planet_name
        self.pos = pos

    def __str__(self):
        return f"name:{self.name},pos:{self.pos}"

    def __repr__(self):
        return f"Planet[name:{self.name},pos:{self.pos}]"

    def __sub__(self, other):
        ans = []
        for i in range(len(self.pos)):
            ans.append(self.pos[i] - other.pos[i])
        return tuple(ans)

    def __eq__(self, other):
        if isinstance(other, Planet):
            return self.name == other.name and self.pos == other.pos
        else:
            return False
```

שאלה 4.2.

```python
class OurSolarSystem(SolarSystem):
    def __init__(self):
        self.planets = {}

    def add_planet(self, planet: Planet) -> bool:
        if planet.name in self.planets:
            return False
        else:
            self.planets[planet.name] = planet
            return True

    def remove_planet(self, planet_name: str) -> bool:
        if planet_name not in self.planets:
            return False
        else:
            del self.planets[planet_name]

    def get_planet(self, planet_name: str) -> Planet:
        return self.planets.get(planet_name)
```

שאלה 4.3

תשובה מקובלת:

```python
def closest_planet(self, pos: tuple) -> Planet:
    p: Planet
    min = math.inf
    planet = None
    for p in self.planets.values():
        dist = self._distance(p.pos, pos)
        if dist < min:
            min = dist
            planet = p

    return planet

def _distance(self, from_, to_):
    x = to_[0] - from_[0]
    y = to_[1] - from_[1]
    z = to_[2] - from_[2]
    return math.sqrt(x ** 2 + y ** 2 + z ** 2)
```

פתרון מעט מתוחכם (לא נדרש):

```python
def closest_planet(self, pos: tuple) -> Planet:
    if len(self.planets) > 0:
        return min(self.planets.values(), key=lambda planet: planet.dist(pos))
```

דורש להוסיף את השיטה **distance** למחלקה **Planet**:

```python
def dist(self, other_pos):
    x, y, z = self.pos
    x_o, y_o, z_0 = other_pos
    return math.sqrt((x - x_o) ** 2 + (y - y_o) ** 2 + (z - z_0) ** 2)
```