S.O.L.I.D principles

SOILD הם ראשי תיבות של 5 עקרונות עיצוב תוכנה שנועדו לסייע לפיתוח תוכנה טובה יותר. העקרונות הללו נועדו להקל על תחזוקת הקוד, להפחית את הקושי בהוספת פיצ'רים חדשים ולהקל על הבנת הקוד. כל עקרון מתמקד בבעיה אחרת ומציע פתרון לבעיה זו.

- S - Single Responsibility Principle
- O - Open/Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Single Responsibility Principle

לכל מחלקה צריכה להיות רק תפקיד אחד מוגדר היטב. במילים אחרות צריך להיות רק סיבה אחת לשינוי במחלקה.

יתרונות

- שינויים בחלק אחד של הקוד ישפיעו פחות על חלקים אחרים.
- קל יותר לעשות טסטים ולדאבג מחלקות עם תפקיד אחד.
- הקוד יהיה יותר קריא וקל להבנה.

Open/Closed Principle

עקרון זה מציין שמחלקות צריכות להיות פתוחות להרחבה וסגורות לשינוי. כלומר, כאשר נרצה להוסיף פיצ'ר חדש למחלקה נצטרך להוסיף קוד חדש ולא לשנות את הקוד הקיים.

יתרונות

- קל יותר להוסיף פיצ'רים חדשים למחלקה.
- אם נשנה קוד קיים, הסיכוי שמשהו יישבר הוא קטן יותר.
- הקוד מתאים יותר לשינויים עתידיים.

Liskov Substitution Principle

העקרון מציין שכל מחלקה שיורשת ממחלקה אחרת צריכה להתנהג כמו המחלקה שהיא יורשת ממנה. כלומר אי אפשר לשנות את ההתנהגות של המחלקה היורשת.

דוגמה ללמה העקרון הזה נחוץ:

נניח שאנחנו רוצים לייצג מלבן וריבוע. נגיד והמתכנת רוצה שריבוע יירש ממלבן (כי ריבוע מקיים את כל התכונות של מלבן).

מצד שני, כדי לשמור על התכונות של מלבן, נצטרך לדאוג שהרוחב והגובה יישארו זהים.

אז הקוד יכול להיראות כך:

```
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getWidth() {
        return width;
```

```
    }

    public int getHeight() {
        return height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

בעצם מה שהמתכנת עשה כאן זה שברגע שנשנה את הרוחב של הריבוע נשנה גם את הגובה.
עכשיו נניח שיש לנו את הקוד הבא:

```
public void testRectangleArea(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.getArea() == 20;
}
```

כאשר נריץ את הקוד עם מלבן כקלט, הקוד יעבוד כמצופה. אבל כאשר נריץ את הקוד עם ריבוע כקלט, הקוד יכשל. זה קורה כי הריבוע לא מתנהג כמו מלבן.

בעצם המתכנת עבר על העקרון של Liskov Substitution Principle. כלומר המתכנת יכול להוסיף פיצ'רים למחלקה ריבוע כל עוד הוא מבטיח שהריבוע יתנהג כמו מלבן, וכמו שניתן לראות המתכנת לא עשה זאת, כי הפונקציות של הריבוע שינו את ההתנהגות של מלבן.

יתרונות של Liskov Substitution Principle

- ניתן השתמש בתתי מחלקות במקום המחלקה האב והקוד יעבוד כרגיל.
- הקוד נשאר עקבי וניתן לחיזוי כשאר נשתמש בתתי מחלקות.
- מבטיח היררכית ירושה נכונה.

Interface Segregation Principle

עקרון הפרדת הממשקים קובע שמחלקה לא אמורה להיות תלויה ברכיבים שהיא לא משתמשת בהם. במילים אחרות לא נרצה לחייב את מי משתמש במחלקה להשתמש בפונקציות שהוא לא צריך.

דוגמת קוד: המתכנת איצטיק רוצה לייצג ממשק עבור חיות. אז הוא הגדיר את הממשק הבא:

```
interface Animal {
    void eat();
    void sleep();
    void fly();
void makeSound();
}
```

איצטיק חשב שעכשיו הוא יוכל לייצג כל חיה בעולם. אבל כאשר הוא רצה לייצג דג, הוא נתקל בבעיה. דג לא יכול לעוף ולא יכול ליצור צלילים (האמנם?). ולכן הממשק של איצטיק לא יתאים בצורה טובה לדג.

בעצם המתכנת עבר על העקרון של Interface Segregation Principle. כלומר המתכנת יכול להפריד את הממשק לממשקים קטנים יותר ולא להכליל את כל הפונקציות בממשק אחד.

פתרון נכון יהיה להפריד את הממשק לממשקים קטנים יותר:

```
interface CanEat {
    void eat();
}

interface CanSleep {
    void sleep();
}

interface CanFly {
    void fly();
}

interface CanMakeSound {
    void makeSound();
}
```

יתרונות של Interface Segregation Principle

- הקוד קטן יותר וקל יותר לתחזוקה, כי קל לזהות ולפתור בעיות קטנות.
- הקוד יותר קריא וקל להבנה.
- קל יותר להוסיף פיצ'רים חדשים לממשקים קטנים יותר.
- אנחנו מקטינים את התלות בין המחלקות כי לכל מחלקה יש רק את הפונקציות שהיא צריכה.

Dependency Inversion Principle

עקרון היפוך התלות מציין שמחלקות צריכות להיות תלויות בממשקים ולא במחלקות ספציפיות. כלומר כאשר מחלקה משתמשת במחלקה אחרת, היא צריכה להשתמש בממשק של המחלקה ולא בה ממשק ספציפי. (כלומר מחלקות ברמה גבוה לא צריכות לדעת על מחלקות ברמה נמוכה).

דוגמת קוד: נניח שיש לנו מחלקה שמייצגת מסעדה (מחלקה ברמה גבוה), בתוך המסעדה יש לנו שף (מחלקה ברמה נמוכה), שמכין את המנות.

דרך אחת (ללא שימוש בעקרון) כדי לכתוב את הקוד של המסעדה תהיה:

```
class Chef {
    public void cook() {
        // קוד לבישול
    }
}

class Restaurant {
    private Chef chef;

    public Restaurant() {
        this.chef = new Chef();
    }

    public void serveFood() {
        chef.cook();
    }
}
```

הבעיה כאן היא שהמסעדה תלויה בשף ספציפי. כלומר אם נרצה להחליף את השף נצטרך לשנות את הקוד של המסעדה.

פתרון לבעיה היא להשתמש בממשק שייצג את השף, וכאשר ניצור את המסעדה נעביר לה את השף דרך הבנאי:

```
interface IChef {
    void cook();
}

class Chef implements IChef {
    public void cook() {
        // קוד לבישול
```

```
    }
}

class Restaurant {
    private IChef chef;

    public Restaurant(IChef chef) {
        this.chef = chef;
    }

    public void serveFood() {
        chef.cook();
    }
}

static void main(String[] args) {
    IChef chef = new Chef();
    Restaurant restaurant = new Restaurant(chef);
    restaurant.serveFood();
}
```

עכשיו המסעדה לא תלויה בשף ספציפי, ואפשר להחליף את השף בקלות, מהמקום שבו נרצה.

יתרונות של Dependency Inversion Principle

- אנחנו מקטינים את התלות בין המחלקות, המחלקה הגבוהה תלויה רק בממשק של המחלקה הנמוכה, ולא במחלקה עצמה.
- הקוד יותר מודולרי - כלומר קל יותר לפיתוח ותחזוקה.
- קל יותר לבצע שינויים, כי אין התלות בין המחלקות קטנה.