SOLID:

עקרונות עיצוב בתכנות object-oriented.
המילה solid מסמלת את 5 עקורונות העיצוב הבאים:

1. **S**ingle responsibility principle
2. **O**pen/Closed principle
3. **L**iskov substitution principle
4. **I**nterface segregation principle
5. **D**ependency inversion

עקרונות אלו מאפשרים לפתח קוד מובן יותר וקל לתחזוק והרחבה לאורך זמן.

coupling-צימוד: מייצג את רמת התלות בין מודולים שונים באותה מערכת.
בבניית מערכת נעדיף להגיע לרמת צימוד **נמוכה**. כדי להשיג את המטרה נצטרך לתכנן כל מודל כיחידה עצמאית ככל הניתן, אשר תלוי כמה שפחות במודולים נוספים במערכת. ברמת צימוד נמוכה, כשנצטרך לעשות שינוי במודל, השינוי יהיה פשוט כיוון שלא נצטרך לבצע שינויים בכלל המערכת.

cohesion-לכידות: חוזק הקשר הפונקציונלי בין פעולות שונות תחת אותו מודל. בבניית מערכת נעדיף להגיע לרמת לכידות **גבוה**ה.
סיווג רמות לכידות: (מהטובה ביותר אל הגרועה ביותר)

1. לכידות פונקציונלית (Functional Cohesion) - פעולות במודול תורמות כולן למשימה בודדת ומוגדרת היטב של המודול.
2. לכידות סדרתית (Sequential Cohesion) - פעולות במודול מקובצות תחתיו כיוון שהפלט של פעולות מסוימות הוא הקלט של פעולות אחרות.
3. לכידות תקשורתית (Communicational Cohesion) - פעולות במודול מקובצות תחתיו כיוון שהן פועלות על אותו המידע.
4. לכידות פרוצדורלית (Procedural Cohesion) - פעולות במודול מקובצות תחתיו כיוון שהן תמיד מבוצעות יחד כחלק מהליך של המערכת (לדוגמה, פעולת בקשת הרשאות ופעולת פתיחת קובץ המבוצעות בזו אחר זו).
5. לכידות טמפורלית (Temporal Cohesion) - כאשר הפעולות במודול מקובצות תחתיו כיוון שהן מבצעות את פעולותיהן במהלך פרק זמן מוגדר במהלך ריצת התוכנית.
6. לכידות לוגית (Logic Cohesion) - כאשר הפעולות במודול מקובצות תחתיו כיוון שהן מבצעות משימות דומות, גם כאשר הן שונות מטבען (לדוגמה, קיבוץ כל פעולות הקלט תחת אותו מודול).
7. לכידות מקרית (Coincidental Cohesion) - הפעולות במודול מקובצות תחתיו באופן שרירותי, כאשר הקשר היחיד בין הפעולות הוא בכך שקובצו יחדיו תחת אותו מודול.

Single responsibility principle:
משפט שמתאר את עקרון זה: "למחלקה צריכה להיות סיבה אחת, ואחת בלבד, בשביל להשתנות". כלומר מחלקה בעלת אחריות יחידה תשתנה רק אם יהיה שיוני בקשר לאחריות שתחתיה.
איך זה עוזר בפיתוח תוכנה?:

- testing: למחלקה עם אחריות אחת יהיה מספר מצומצם של טסטים.

בנוסף מקל על הגדרת mock למחלקה.

- lower coupling: פחות אחריות במחלקה אחת, פחות תלות במחלקות אחרות.
- organization: במחלקות קטנות ומסודרות, קל יותר לחפש את המידע מאשר מחלקה אחת גדולה שמכילה הכל.

דוגמא:

מחלקת book, המחלקה מכילה את השדות שמתארות את הספר ואת הפונקציונליות החיפוש בספר.

כעת נרצה להוסיף פונקציה להדפיס את הספר, הוספת הפונקציה תגדיל את האחריות של הספר ובכך תפר את ה single responsibility principle.

הסתרון הוא לבנות מחלקה BookPrinter שתהיה אחראית על הדפסת ספר

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word){
        return text.replaceAll(word, text);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}

public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

Open/Closed principle

מחלקות צריכות להיות פתוחות להוספות וסגורות לשינויים.

איך זה עוזר בפיתוח תוכנה?:

- מונע שינויים בקוד קיים ועובד, ובכך מונע הוספה פוטנציאלית של בגים חדשים לקוד קיים.

דוגמא:

מחלקת Guitar, המחלקה מייצגת גיטרה.

כעת נרצה לשדרג את הגיטרה ולהוסיף לה ציור של אש. אנחנו יכולים להוסיף את הפיצר החדש למחלקה של Guitar אבל לא נדע האם השינוי יגרום לבאגים חדשים בקוד. לכן אם נוסיף מחלקה חדשה CoolGiuitar שתירש מ Guitar, נוכל להוסיף את ההתנהגות שנרצה, בלי לפגוע בקוד הקיים.

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}


public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Liskov substitution principle

אן המחלקה A היא מחלקת בן של B, אז נוכל להחליף את B ב A בכל מקום שנירצה בלי להשפיע על ההתנהגות של המערכת.

Interface segregation principle

Dependency inversion