תוכן העניינים

[**מבוא:** 1](#_Toc145945017)

[**Single Responsibility Principle (SRP)** 3](#_Toc145945018)

[דוגמה של קוד לפני מימוש ה Single Responsibility Principle 4](#_Toc145945019)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - SRP 5](#_Toc145945020)

[**Open-Closed Principle (OCP)** 6](#_Toc145945021)

[דוגמה של קוד לפני מימוש ה Open-Closed Principle 7](#_Toc145945022)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - OCP 7](#_Toc145945023)

[**Liskov Substitution Principle (LSP)** 9](#_Toc145945024)

[דוגמה של קוד לפני מימוש הLiskov Substitution Principle - 10](#_Toc145945025)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - LSP 11](#_Toc145945026)

[הנה דוגמה נוספת שמדגימה הפרה של SLP 12](#_Toc145945027)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - LSP 13](#_Toc145945028)

[**Interface Segregation Principle (ISP)** 14](#_Toc145945029)

[דוגמה של קוד לפני מימוש הInterface Segregation Principle 15](#_Toc145945030)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - ISP 15](#_Toc145945031)

[**Dependency Inversion Principle (DIP)** 16](#_Toc145945032)

[דוגמה של קוד לפני מימוש ה- Dependency Inversion Principle 17](#_Toc145945033)

[עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - DIP 18](#_Toc145945034)

[**סיכום עקרונות S.O.L.I.D:** 19](#_Toc145945035)

[**תוספות שאינן חלק מ SOLID:** 20](#_Toc145945036)

[**DRY** 20](#_Toc145945037)

[**KISS** 21](#_Toc145945038)

[**YAGNI** 21](#_Toc145945039)

# מבוא:

עקרונות SOLID הם קבוצה של חמישה עקרונות עיצוב שהוצגו בשנת 2000 על ידי רוברט סי מרטין הידוע גם בשם ,Uncle Bob כדי להנחות מפתחים בכתיבת קוד נקי, בר תחזוקה וניתן להרחבה. עקרונות אלו משמשים בסיס לבניית מערכות תוכנה חזקות וגמישות.

מאז, עקרונות SOLID זכו להכרה רחבה ואומצו על ידי קהילת פיתוח התוכנה.

במצגת קצרה זו, נחקור מהם עקרונות SOLID ומדוע הם חשובים כל כך.

1. :Single Responsibility Principle (SRP)

עיקרון זה, (האחריות היחידה) קובע של class צריכה להיות רק סיבה אחת לשינוי. על ידי הקפדה על עיקרון זה, מתכנתים ייצרו מחלקות ממוקדות ומלוכדות יותר שאחראיות למשימה או פונקציונליות בודדת. מה שיוביל לכתיבת קוד שיותר קל להבין, לבדוק ולתחזק.

1. Open-Closed Principle (OCP):

עיקרון זה, (הפתוח/סגור) מדגיש שישויות תוכנה (מחלקות, מודולים, פונקציות וכו') צריכות להיות פתוחות להרחבה אך סגורות לשינוי. על ידי עיצוב הקוד שלנו באופן המאפשר הרחבה קלה מבלי לשנות את הקוד הקיים, נוכל להימנע מהכנסת באגים ותופעות לוואי לא מכוונות. מה שמקדם שימוש חוזר בקוד, מדרגיות וגמישות.

1. Liskov Substitution Principle (LSP):

עקרון זה, (ההחלפה שלLiskov ) קובע שאובייקטים של מחלקת-על צריכים להיות ניתנים להחלפה באובייקטים של תת-המחלקות שלה מבלי להשפיע על נכונות התוכנית. על ידי הקפדה על עיקרון זה, מתכנתים מבטיחים שניתן להרחיב את הקוד שלהם בקלות וניתן להוסיף פונקציונליות חדשה מבלי לשבור את הפונקציונליות הקיימת. מה שמקדם מודולריות ויכולת פעולה הדדית.

1. Interface Segregation Principle (ISP):

עקרון זה, (הפרדת הממשק) מציע שאין לאלץ לקוחות להיות תלויים בממשקים שהם לא משתמשים בהם. במילים אחרות, ממשקים צריכים להיות דקיקים ומותאמים לצרכי הלקוח הספציפיים. על ידי ביצוע עיקרון זה, מתכנתים יוצרים ממשקים ממוקדים ומלוכדים, מצמצמים את הצימוד בין הרכיבים ומגדילים את יכולת התחזוקה והשימוש החוזר של הקוד.

1. :Dependency Inversion Principle (DIP)

עקרון זה, (היפוך התלות) קובע שמודולים ברמה גבוהה לא צריכים להיות תלויים במודולים ברמה נמוכה, שניהם צריכים להיות תלויים בהפשטות (abstract). עיקרון זה מקדם צימוד רופף (loose coupling) ומאפשר בדיקה קלה יותר, גמישות ומודולריות. על ידי הסתמכות על הפשטות (abstract). ולא על יישומים קונקרטיים, מתכנתים יכולים להחליף תלות בקלות ולשפר את העיצוב הכולל של הקוד שלהם.

**היתרונות בשימוש בעקרונות SOLID:**

השקעת זמן בלימוד עקרונות SOLID מביאה מספר יתרונות:

1. **קוד נקי וניתן לתחזוקה:** על ידי ביצוע עקרונות SOLID, מתכנתים יכתבו קוד שקל יותר להבין, לשחזר ולתחזק. זה מוביל להפחתת באגים ולמחזורי פיתוח מהירים יותר.
2. **מדרגיות(Scalability) והרחבה:** עקרונות SOLID מעודדים קוד מודולרי וגמיש, המאפשר הוספה, שינוי והסרה קלה של תכונות. מה שמקדם מדרגיות והתאמה לדרישות העסקיות המשתנות.
3. **שימוש חוזר בקוד:** על ידי הקפדה על עקרונות SOLID המפתחים יוצרים קוד שניתן לשימוש חוזר. זה מפחית כפילות, משפר את הפרודוקטיביות ומבטיח עקביות בכל בסיס הקוד.
4. **שיתוף פעולה ופרודוקטיביות:** עקרונות SOLID מספקים קבוצה משותפת של קווים מנחים המאפשרים תקשורת טובה יותר, שיתוף פעולה והבנה משותפת בין מפתחים. מה שמוביל לפרודוקטיביות מוגברת ועבודת צוות חלקה יותר.

סיכום:

שילוב עקרונות SOLID בפרקטיקות הפיתוח שלנו היא השקעה משתלמת. על ידי לימוד העקרונות הללו, אנו מתכנתים מרגילים את עצמנו בכתיבת קוד נקי, בר תחזוקה, ניתן להרחבה וניתן לשימוש חוזר. מה שיביא בסופו של דבר לפיתוח יעיל יותר, צמצום בתקלות ומוצרי תוכנה באיכות גבוהה יותר עבור הארגון שלנו.

**CLASS DESIGN PRINCIPLES**

**ראשי התיבות SOLID**

* **S**RP: The **S**ingle Responsibility Principle
* עקרון האחריות היחידה
* **O**CP: The **O**pen-Closed Principle
* עקרון פתוח-סגור
* **L**SP: The **L**iskov Substitution Principle
* עקרון ההחלפה של ליסקוב
* **I**SP: The **I**nterface Segregation Principle
* עקרון הפרדת הממשק
* **DIP**: The **D**ependency Inversion Principle
* עקרון היפוך תלות

# Single Responsibility Principle (SRP)

עיקרון זה, הוא עיקרון עיצוב תוכנה הקובע ש

“**A class should have one, and only one, reason to change**”

**"ל** class **צריכה להיות סיבה אחת, ויחידה, לשנוי"**

המילה אחריות אינה אומרת פונקציה, זה לא אומר מה זה עושה, המילה אחריות משמעותה **שינוי** שתהיה סיבה אחת לשינוי

זאת אומרת שלכלclass במערכת תוכנה צריכה להיות רק עבודה אחת ספציפית לעשות.

ה class צריך להתמקד באחריות אחת ולעשות אותה היטב.

תאר לעצמך שיש לך ארגז כלים עם כלים שונים. לכל כלי יש מטרה מסוימת והוא מבצע משימה מסוימת. לדוגמה: פטיש מיועד לנעיצת מסמרים, בעוד מברג משמש לסיבוב ברגים.

כל class צריך להיות כמו כלי בארגז הכלים שלך, עם מטרה ואחריות ברורה. הוא צריך לבצע משימה או עבודה ספציפית אחת מבלי לנסות לעשות יותר מדי דברים.

היתרונות של ביצוע ה SRP הם:

1. **קל להבנה:** כאשר ל class יש אחריות אחת, קל יותר להבין את מטרתו וכיצד הוא משתלב במערכת הכוללת. זה כמו להשתמש בכלי המתאים לעבודה ספציפית, מה שמקל על ההבנה והתחזוקה.
2. **פשוט לשינוי:** אם ל class יש אחריות אחת בלבד, פשוט יותר לבצע שינויים. המפתח/ת מתמקד בפונקציונליות הספציפית הזו מבלי לדאוג להשפעתה על חלקים אחרים שאינם קשורים במערכת.
3. **שימוש חוזר:** בדיוק כמו שאתה יכול לעשות שימוש חוזר בכלי ספציפי עבור פרויקטים שונים, ב class עם אחריות יחידה ניתן לעשות שימוש חוזר בקלות, בחלקים אחרים של מערכת התוכנה שלך או בפרויקטים שונים. זה מקדם שימוש חוזר בקוד וחוסך זמן פיתוח.
4. **בדיקה קלה יותר:** הבדיקה הופכת לניתנת לניהול כאשר לכל class יש אחריות יחידה. ניתן לבדוק את הפונקציונליות הזו בצורה מבודדת, לוודא שהיא פועלת כמצופה מבלי להיות מושפעת מחלקים אחרים שאינם קשורים.

בסיכום: אל תשים פונקציות שונות שעושות דברים שונים באותו ה class

בהקפדה על עיקרון האחריות היחידה, אתה יוצר בסיס קוד מאורגן וניתן לתחזוקה, על ידי שמירה על מיקוד מחלקות/מודולים, פשוטים וקלים להבנה. מה שעוזר בבניית מערכות תוכנה גמישות, ניתנות לשימוש חוזר ולהרחבה.

## דוגמה של קוד לפני מימוש ה Single Responsibility Principle

// Example without SRP

public class User

{

public string Name {get; set;}

public string Email {get; set;}

// Method to send email to user

public void SendEmail(string message)

{

// Code to send email

Console.WriteLine($"Sending email to {Email}: {message}");

}

// Method to save user to the database

public void SaveToDatabase()

{

// Code to save user to the database

Console.WriteLine($"Saving user {Name} to the database");

}

}

בדוגמה שלמעלה, המחלקה **User** מפרה את עיקרון האחריות היחידה מכיוון שיש לה מספר תחומי אחריות. יש לה מאפיינים הקשורים לישות User, ויש לה גם שתי מתודות: לשליחת מיילים ושמירת משתמשים במסד הנתונים.

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - SRP

**דוגמה מתוקנת:**

// SRPדוגמא עם

public class User

{

public string Name {get; set;}

public string Email {get; set;}

}

public class EmailSender

{

public void SendEmail(User user, string message)

{

// Code to send email

Console.WriteLine($"Sending email to {user.Email}: {message}");

}

}

public class UserRepository

{

public void SaveUser(User user)

{

// Code to save user to the database

Console.WriteLine($"Saving user {user.Name} to the database");

}

}

בקוד המחודש, הפרדנו את האחריות למחלקות שונות. המחלקה **User** מייצגת כעת רק את ישות המשתמש עם מאפיינים הקשורים אליה.

המתודה **EmailSender** אחראית על שליחת מיילים. ומשתמשת באובייקט **User** ובמשתנה **message**

כפרמטרים בכדי לשלוח דואר אלקטרוני למשתמש.

המחלקה **UserRepository** אחראית על שמירת משתמשים במסד הנתונים. ומשתמשת באובייקט **User** כפרמטר ושומרת אותו במסד הנתונים.

על ידי הפרדת האחריות למחלקות שונות ולפונקציות שונות, אנו דבקים בעקרון האחריות היחידה. לכל class יש אחריות אחת ויחידה והוא ממוקד בביצוע המשימה הספציפית שהוגדרה לו.

הקפדה על עקרון זה תסייע לכתיבת קוד שניתן לתחזוקה, לבדיקה ולשימוש חוזר.

# Open-Closed Principle (OCP)

עיקרון זה, הוא עיקרון עיצוב תוכנה הקובע ש

**“Modules should be open for extension, but closed for modification”**

- Bertrand Meyer

**"מודולים צריכים להיות פתוחים להרחבה, אך סגורים לשינוי"**

במילים אחרות, ההתנהגות של ישות תוכנה צריכה להיות ניתנת להרחבה בקלות מבלי לשנות את הקוד הקיים שלה, מודולים צריכים להיות פתוחים להרחבה, אך סגורים לצורך שינוי.

הרעיון הוא, לעצב תוכנה באופן המאפשר הוספה קלה של תכונות או התנהגויות חדשות ללא צורך לשנות את בסיס הקוד הקיים.

## דוגמה של קוד לפני מימוש ה Open-Closed Principle

// Shape class representing a rectangle

public class Rectangle

{

public double Width { get; set; } // Width of the rectangle

public double Height { get; set; } // Height of the rectangle

}

// Shape class representing a circle

public class Circle

{

public double Radius { get; set; } // Radius of the circle

}

// Area Calculator class that violates the Open-Closed Principle

public class AreaCalculator

{

// Method to calculate the area of a shape

public double CalculateArea(object shape)

{

// Check the type of the shape

if (shape is Rectangle)

{

var rectangle = (Rectangle)shape; // Cast the shape to Rectangle type

return rectangle.Width \* rectangle.Height; // Calculate area of the rectangle

}

else if (shape is Circle)

{

var circle = (Circle)shape; // Cast the shape to Circle type

return Math.PI \* circle.Radius \* circle.Radius; // Calculate area of the circle

}

throw new ArgumentException("Invalid shape type."); // Throw an exception for unsupported shape types

}

}

בקוד זה, יש לנו מחלקה **Rectangle** ומחלקה **Circle** המייצגות צורות שונות. למחלקה **AreaCalculator** יש מתודה **CalculateArea** שלוקחת פרמטר **object** המייצג את הצורה. עם זאת, יישום זה מפר את עקרון ה Open-Closed Principle מכיוון שהוא אינו סגור לשינוי, כי כאשר אנו רוצים להוסיף תמיכה בצורות גאומטריות חדשות, כמו מעגלים או משולשים, נצטרך לשנות את קוד ה **AreaCalculator**.

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - OCP

**דוגמה מתוקנת:**

// Interface for shapes

public interface IShape

{

double CalculateArea();

}

// Shape class representing a rectangle

public class Rectangle : IShape

{

public double Width { get; set; } // Width of the rectangle

public double Height { get; set; } // Height of the rectangle

public double CalculateArea()

{

return Width \* Height; // Calculate area of the rectangle

}

}

// Shape class representing a circle

public class Circle : IShape

{

public double Radius { get; set; } // Radius of the circle

public double CalculateArea()

{

return Math.PI \* Radius \* Radius; // Calculate area of the circle

}

}

// Area Calculator class that adheres to the Open-Closed Principle

public class AreaCalculator

{

public double CalculateArea(IShape shape)

{

return shape.CalculateArea(); // Calculate area using the shape's implementation

}

}

Circle circle = new Circle { Radius = 5.0 };

Rectangle rectangle = new Rectangle { Height = 20, Width = 30 };

AreaCalculator calculator = new AreaCalculator();

double circleArea = calculator.CalculateArea(circle);

double rectangleArea = calculator.CalculateArea(rectangle);

Console.WriteLine($"The area of the circle is: {circleArea}, The area of the rectangle is: {rectangleArea}");

זו תהיה התוצאה שנקבל:

The area of the circle is: 78.53981633974483, The area of the rectangle is: 600

בקוד המעודכן הזה, אנו מציגים את ה interface **IShape** המגדיר התודה **CalculateArea(),** המחלקות **Rectangle** ו- **Circle** מיישמות interface זה, ומספקות יישום משלהן של מתודת **CalculateArea()** בהתבסס על חישובי הצורה הספציפיים שלהן.

המחלקה **AreaCalculator** מקבלת כעת אובייקט **IShape** כפרמטר במתודה **CalculateArea()** וזה מאפשר ל-**AreaCalculator** לעבוד עם כל צורה המיישמת את interface **IShape** ללא צורך בשינוי.

וכמובן זה עומד ב Open-Closed Principle מכיוון שהמחלקה **AreaCalculator** סגורה כעת לשינוי אך פתוחה להרחבה על ידי הצגת מחלקות צורה חדשות המיישמות את ה interface **IShape**.

# Liskov Substitution Principle (LSP)

עיקרון זה, הוא עיקרון עיצוב תוכנה הקובע ש... ואני מצטט את המילים של ברברה ליסקוב

**“Subtype should behave like a super type as far you can tell, by using the super type methods.”**

<https://www.youtube.com/watch?v=-Z-17h3jG0A>

והיא מסבירה, שימוש במושג " Type hierarchy", הרעיון הוא ש- superclass יגדיר טיפוס על ואז תת-מחלקה תרחיב את זה כדי להפוך לתת-סוג.

אז זה לא שהsubtype לא יכול היה להתנהג אחרת, אלא שכל עוד תגביל את האינטראקציה שלך עם האובייקט לשיטות העל, תקבל את ההתנהגות שאתה מצפה לה.

ועל פי ההסבר של Uncle Bob,

**"מחלקות נגזרות חייבות להיות ניתנות לשימוש דרך ממשק המחלקה הבסיסית, ללא צורך שהמשתמש יידע את ההבדל"**

מושג זה מתייחס לעקרון הפולימורפיזם ולעקרון ההחלפה של ליסקוב (LSP) הוא מדגיש כי מחלקות נגזרות צריכות להיות מסוגלות להחליף את מחלקת הבסיס ולספק יישום משלהן תוך הקפדה על הממשק של מחלקת הבסיס.

והוא מסביר:

"פונקציות המשתמשות במצביעים או הפניות למחלקה בסיסית חייבות להיות מסוגלות להשתמש באובייקטים מכל מחלקה נגזרת מבלי לדעת זאת."

במילים פשוטות יותר, אם יש לך מחלקה בסיסית ואתה יוצר ממנה מחלקות נגזרות, אתה אמור להיות מסוגל להשתמש במחלקות הנגזרות בכל מקום שבו צפויה מחלקת הבסיס, מבלי לגרום לבעיות או לשבור את הפונקציונליות.

הדוד בוב מסביר עוד שהקפדה על הLSP מחייבת שמחלקות נגזרות לא ישנו את ההתנהגות של מחלקת הבסיס. הם יכולים רק לשפר או להרחיב את הפונקציונליות, אבל הם לא צריכים לשנות את ההתנהגות הצפויה שהוגדרה על ידי מחלקת הבסיס.

## דוגמה של קוד לפני מימוש הLiskov Substitution Principle -

public class Rectangle

{

public virtual double Width { get; set; }

public virtual double Height { get; set; }

public double CalculateArea()

{

return Width \* Height;

}

}

public class Square : Rectangle

{

public override double Width

{

get { return base.Width; }

set

{

base.Width = value;

base.Height = value;

}

}

public override double Height

{

get { return base.Height; }

set

{

base.Width = value;

base.Height = value;

}

}

}

הבעיה עם הקוד הנתון היא שהוא מפר את עקרון הLSP על ידי שבירת ההתנהגות הצפויה של מחלקת הבסיס (**Rectangle**) במחלקה הנגזרת (**Square**).

בקוד, המחלקה **Square** נגזרת מהמחלקה **Rectangle** ו override מחדש את המאפיינים **Width** ו- **Height.** המאפיינים ש override במחלקה **Square** אוכפים הם: שה- **Width** ו- **Height** תמיד שווים. משמעות הדבר היא שכאשר אתה מגדיר את **Width** או **Height** של אובייקט **Square**  הוא יתאים אוטומטית את הממד השני כך שיתאים לו.

עם זאת, זה מפר את ה LSP מכיוון שהוא משנה את ההתנהגות של מחלקת הבסיס. במחלקת הבסיס ((**Rectangle**  ל- **Width** ו- **Height** יכולים להיות ערכים שונים, המייצגים מלבן בעל ממדים שונים. על ידי הפיכת המחלקה **Square** לאכוף ממדים שווים, היא מגבילה את ההתנהגות של מחלקת הבסיס ומציגה התנהגות בלתי צפויה כאשר מתייחסים לאובייקט **Square** כאל **Rectangle** .

לדוגמה, אם יש לך מתודה שמצפה לאובייקט **Rectangle** ואתה מעביר אליו אובייקט **Square**  המתודה עשויה להניח שהיא יכולה להגדיר באופן עצמאי את ה- **Width** ו- **Height**  אבל עם הקוד הנתון, שינוי מאפיין אחד ישנה אוטומטית את השני, מה שעלול להוביל לתוצאות בלתי צפויות ולחוסר עקביות.

כדי לדבוק בעקרון החלפת ליסקוב, המחלקה הנגזרת (**Square**) לא צריכה לשנות את ההתנהגות של מחלקת הבסיס (**Rectangle**) במקום זאת, זה צריך רק לשפר או להרחיב אותו. במקרה זה, התייחסות לאובייקט **Square** כאובייקט **Rectangle** אמור לשמור על היכולת להגדיר ערכים בלתי תלויים עבור ה- **Width** ו- **Height.**

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - LSP

**דוגמה מתוקנת:**

public abstract class Shape

{

public abstract double CalculateArea();

}

public class Rectangle : Shape

{

public double Width { get; set; }

public double Height { get; set; }

public override double CalculateArea()

{

return Width \* Height;

}

}

public class Square : Shape

{

public double SideLength { get; set; }

public override double CalculateArea()

{

return SideLength \* SideLength;

}

}

בקוד הזה, יש לנו **abstract class** בשם **Shape** המספק חוזה לחישוב השטח של צורות שונות.

ה - **abstract class** מכריז על מתודת **abstract** **CalculateArea()** שכל מחלקה נגזרת חייבת ליישם.

המחלקה **Rectangle** והמחלקה **Square** שתיהן יורשות ממחלקת הבסיס **Shape** . כל מחלקה מספקת יישום משלה של שיטת **CalculateArea()** בהתבסס על הלוגיקה הספציפית של חישוב הצורה שלה.

על ידי שימוש ב **abstract class** והפעלת כל מחלקה נגזרת של שיטת חישוב השטח, אנו מבטיחים שההתנהגות של מחלקת הבסיס לא תשתנה על ידי המחלקות הנגזרות.

הקפדה זו על עקרון ההחלפה של Liskov מאפשרת לנו להשתמש באובייקטים מכל מחלקה נגזרת (**Rectangle** או **Square**) בכל מקום בו צפויה מחלקת הבסיס **Shape** , מבלי לגרום לבעיות בלתי צפויות או לשבור פונקציונליות.

## הנה דוגמה נוספת שמדגימה הפרה של SLP

דמיינו לעצמכם היררכיית מחלקות המייצגת סוגים שונים של ציפורים: מחלקת בסיס בשם **Bird** ושתי תת מחלקות,  **Duck** ו **Ostrich** , שתיהן יורשות מ **Bird**  לכל ציפור יש מתודה הנקראת **Fly()** שתפקידה לייצג את התנהגות הטיסה שלה.

public class Bird

{

// Common bird properties and methods

public virtual void Fly()

{

Console.WriteLine("Flying...");

}

}

public class Duck : Bird

{

public override void Fly()

{

Console.WriteLine("Flying like a duck...");

}

}

public class Ostrich : Bird

{

public override void Fly()

{

throw new NotSupportedException("Ostriches cannot fly!");

}

}

בדוגמה זו, המתודה**Fly()** נכתבת מחדש במחלקה **Duck** כדי לייצג את התנהגות הטיסה הספציפית לברווזים. עם זאת, המחלקה **Ostrich** כותבת מחדש המתודה  **Fly()**אך זורקת **NotSupportedException** מכיוון שהיענים הם ציפורים חסרות מעוף.

כעת, אם נתבונן בקוד הבא:

Bird bird = new Ostrich();

bird.Fly();

על פי עקרון SLPקוד זה אמור לעבוד כהלכה מכיוון ש **Ostrich** הוא תת-מחלקה של  **Bird** עם זאת, במקרה זה, המתודה **Fly()**במחלקה  **Ostrich**זורקת **NotSupportedException** במקום לספק התנהגות משמעותית.

זה מפר את עקרון ההחלפה של Liskov מכיוון שכאשר אנו מחליפים אובייקט **Ostrich** באובייקט **Bird** ההתנהגות של התוכנית נשברת ויוצאת חריגה. ההתנהגות הצפויה, שהיא לבצע התנהגות תעופה הגיונית עבור הציפור הספציפית, אינה נשמרת.

דוגמה זו ממחישה הפרה של עקרון ההחלפה של Liskov על ידי אי שמירה על ההתנהגות הצפויה בעת החלפת מחלקה נגזרת (**Ostrich**) במחלקה הבסיסית שלה.(**Bird**)

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - LSP

public interface IFlyable

{

void Fly();

}

public class Bird : IFlyable

{

// Common bird properties and methods

public void Fly()

{

Console.WriteLine("Flying...");

}

}

public class Duck : IFlyable

{

public void Fly()

{

Console.WriteLine("Flying like a duck...");

}

}

public class Ostrich : Bird

{

public new void Fly()

{

Console.WriteLine("Ostriches cannot fly!");

}

}

בפתרון זה, אנו מציגים interface **IFlyable** המגדיר את התנהגות הטיסה. כל מחלקת ציפורים שמסוגלת לעוף יכולה ליישם את הממשק הזה.

המחלקה **Bird** מיישמת את ממשק **IFlyable** ומספקת יישום נפוץ להתנהגות טיסה. המחלקה **Duck** מיישמת גם את ממשק **IFlyable** ומספקת התנהגות טיסה ספציפית משלה.

המחלקה **Ostrich**  שהיא ציפור חסרת מעוף, יורשת מהמחלקה **Bird** אך מסתירה את המתודה **Fly()** באמצעות מילת המפתח **new** כדי לציין שיענים לא יכולים לעוף.

מילת המפתח **new** מציינת ומדגישה שאני רוצה לציין התנהגות שונה מהמצופה מהמתודה Fly() והיא: לספר למשתמש ש "יענים לא יכולים לעוף".

כעת, בואו נבחן את הקוד המתוקן:

IFlyable bird = new Ostrich ();

bird.Fly();

במקרה זה, הפלט יהיה "יענים לא יכולים לעוף!", המייצג נכון את התנהגותו של יען.

על ידי שימוש ויישום interface **IFlyable**, נוכל לנתק את התנהגות הטיסה מההיררכיה של המחלקות. זה מאפשר לנו גמישות רבה יותר ולהימנע מהפרת עקרון החלפת ליסקוב.

בנוסף, הוא מספק פתרון יעיל יותר וניתן לתחזוקה מכיוון שהוא מקדם שימוש חוזר בקוד.

# Interface Segregation Principle (ISP)

**"Design interfaces in such a way that the classes that implement those interfaces does not have many unused functions."**

עקרון זה, מתרכז בעקרון הסֶגְרֵגַצְיָה (הַפְרָדָה, הַבְדָּלָה) של ממשקים. העיקרון מתייחס לאופן שבו ממשקים (interfaces) מוגדרים וכיצד מחלקות מממשות אותם.

בצורה פשוטה: עקרון זה אומר שמחלקה לא צריכה להיות מחויבת לממש פעולות מ interface שהיא לא צריכה או לא משתמשת בהן.

**דוגמה :**נניח יש לנוinterface **IWorker**  שמגדיר שתי פעולות **Work()** וְ  **Rest()**וכעת יש לנו שתי מחלקות:  **Manager** שבה ממומשים שני הפעולות ו **Intern** שבה ממומשת רק הפעולה **Work()**

עקרון ה ISP אומר שאנחנו לא צריכים להכריח את ה **Intern** לממש את הפעולה**Rest()** שהוא לא משתמש בה. אנחנו יכולים לפצל את ה interface **IWorker** לכמה interfaces יחידים, לדוגמה **IWorker** ו **IRestable** כך שה **Intern** יממש רק את  **IWorker** ולא את **IRestable**.

בסיכום ISP מתייחס לכך שמחלקות לא יכולות להיות תלויות בפעולות שהן לא צריכות או שאינן משתמשות בהן. זה מקנה גמישות ואחריות בתכנון כדי למנוע תלויות מיותרת ובעיות פוטנציאליות.

## דוגמה של קוד לפני מימוש הInterface Segregation Principle

נניח שיש לנו ממשק **IWorker** שמגדיר שתי פעולות **Work()** וְ **Rest()** ונרצה לממש אותו ממשק במחלקה **Manager**

public interface IWorker

{

void Work();

void Rest();

}

public class Manager : IWorker

{

public void Work()

{

Console.WriteLine("The manager is working.");

}

public void Rest()

{

Console.WriteLine("The manager is resting.");

}

}

בדוגמה זו, המחלקה  **Manager** מממשת את ה interface **IWorker** שמגדיר שתי פעולות **Work()** וְ **Rest()** אך הבעיה היא שרק מנהלים יכולים להשתמש בפעולה  **Rest()**והיא לא רלוונטית לכל העובדים באופן כללי.

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - ISP

אנו יכולים לפצל את ה interface **IWorker** לשני ממשקים **IWorker** וְ **IRestable** ולממש רק את ה interface שרלוונטי לכל מחלקה. הנה דוגמה כיצד ניתן לעשות זאת:

**דוגמה מתוקנת:**

public interface IWorker

{

void Work();

}

public interface IRestable

{

void Rest();

}

public class Manager : IWorker, IRestable

{

public void Work()

{

Console.WriteLine("The manager is working.");

}

public void Rest()

{

Console.WriteLine("The manager is resting.");

}

}

public class Intern : IWorker

{

public void Work()

{

Console.WriteLine("The intern is working.");

}

}

בדוגמה המתוקנת, יש לנו שני interfaces **IWorker** ו **IRestable** המחלקה **Manager** מממשת את שני ה interfaces , והמחלקה **Intern** מממשת רק את ה interface **IWorker** באופן הזה אנחנו מתקנים את הקוד ומתעלים את הבעיה שבה מחלקות מממשות פעולות שאינן רלוונטיות להן.

# Dependency Inversion Principle (DIP)

**"High level modules should not depend on low level modules.**

**Both should depend on abstractions.**

**Abstraction should not depend on implementation.**

**Implementation should depend on abstraction."**

עיקרון זה, אומר:

**תכנת בצורה שתאפשר לך להתמקד במה שחשוב במערכת שלך ולא בפרטים טכניים חסרי חשיבות.**

במילים אחרות, כשאתה כותב קוד, נסה להשאיר את החלקים המורכבים והטכניים המשמשים אותך בהיררכיית קוד ברמה גבוהה. זה יאפשר לך לשנות חלקים מסוימים של הקוד מבלי לשנות את האחרים.

דוגמה פשוטה היא של מכונית: כשאתה נוהג במכונית, אתה לא צריך לדעת בדיוק איך המנוע עובד או איך פועלים ההילוכים. אתה פשוט לוחץ על דוושת ההאצה, והמכונית זזה. הפרטים הטכניים של המנוע והגלגלים מוסתרים מאחורי הקלעים כדאי שתדעו - אלה הם הפרטים הטכניים שמתמקדים בהעברה ממקום למקום.

בעולם התכנות, אם אתה מתכנת נכון, התוכנית שלך צריכה לעבוד בצורה דומה: התוכנית הראשית צריכה להתמקד במשימות ליבה, בעוד שפרטים טכניים צריכים להיות בחלקים אחרים של הקוד שאינם משפיעים על התוכנה הראשית.

במונחים פשוטים, זה אומר שאנחנו לא צריכים לסמן קשרים ישירים בין חלקי הקוד שלנו לחלקים אחרים, אלא נשתמש ב interfaces , כך שאם נרצה לשנות חלק מהקוד, נוכל לעשות זאת מבלי לשנות את שאר הקוד.

העיקרון מציין שעדיף להיתלות בממשקים או ממשקי אבסטרקט, במקום בממשקי פרטיים או פרטים ספציפיים.

לדוגמה, אם יש לנו מחלקה **Car** שמשתמשת במחלקה **Engine** ניתן לתת ממשק אבסטרקטי שמגדיר את הפונקציות הנדרשות למנוע (**Start()**, **Stop()**) ולתת למנועים השונים לממש אותו הממשק. בכך, אפשר להחליף מנוע בקלות מבלי לשנות את קוד הרכב עצמו.

המטרה היא ליצור קוד גמיש, נגיש ופחות תלוי בפרטים ספציפיים ובכך נקל על תחזוקה, ניהול ופיתוח בעתיד.

## דוגמה של קוד לפני מימוש ה- Dependency Inversion Principle

בדוגמה הזו, יש לנו מחלקה  **Car** שמשתמשת במנוע מסוג שתי פעימות בצורה ישירה

// Car class represents a car with a specific type of engine.

public class Car

{

// Private instance of GasolineEngine, directly coupled to this car.

private GasolineEngine engine = new GasolineEngine();

// Method to start the car.

public void Start()

{

// Starting the engine.

engine.Start();

Console.WriteLine("Car is running.");

}

}

// GasolineEngine class represents a gasoline engine.

public class GasolineEngine

{

// Method to start the gasoline engine.

public void Start()

{

Console.WriteLine("Gasoline engine started.");

}

}

בקוד הזה שאינו מתייחס לעקרון Dependency Inversion Principle (DIP) ישנם כמה חסרונות:

1. **קשר ישיר בין המחלקות:** למחלקה  **Car** יש תלות ישירה במחלקת המנוע  **GasolineEngine** וזה אומר שאם תרצה להשתמש בסוג אחר של מנוע (למשל מנוע חשמלי או מנוע דיזל) תצטרך לשנות את הקוד במחלקת **Car**.
2. **Low abstraction:** המחלקה  **Car** נמצאת ברמה נמוכה בהיררכיה הטכנית והיא יודעת יותר מידי על הפרטים הטכניים שבמנוע. וזה עשוי לגרום לקושי בתחזוקה ובשינויים בעתיד.
3. **קושי בבדיקות (Unit Testing)** בגלל התלות הישירה במחלקת  **GasolineEngine** ייתכן שיהיה קשה יותר לבדוק את מחלקת  **Car** ביחידות כאשר נרצה לכתוב לה בדיקות יחידות(unit tests).

באופן כללי החסרונות בקוד זה הם קשיים בתחזוקה, חוסר גמישות בהרחבה, וקושי ב Unit Testing.

כאשר מיישמים את עקרון Dependency Inversion Principle (DIP) הקוד יהיה יותר גמיש, קל יותר לתחזק ולבצע Unit Testing יעילות יותר.

## עכשיו, נראה איך נוכל לתקן זאת בעזרת ה - DIP

בדוגמה הזו, אנו משתמשים בממשק **IEngine** כדי לייצר סוגי מנועים שונים ולהשתמש בהם בתחום רכבים:

**דוגמה מתוקנת:**

// Define an interface for engines.

public interface IEngine

{

void Start();

}

// GasolineEngine class implementing the IEngine interface.

public class GasolineEngine : IEngine

{

// Start method for gasoline engine.

public void Start()

{

Console.WriteLine("Gasoline engine started.");

}

}

// ElectricEngine class implementing the IEngine interface.

public class ElectricEngine : IEngine

{

// Start method for electric engine.

public void Start()

{

Console.WriteLine("Electric engine started.");

}

}

// Car class represents a car that uses an engine.

public class Car

{

private IEngine engine;

// Constructor to set the engine for the car.

public Car(IEngine engine)

{

this.engine = engine;

}

// Start method for the car.

public void Start()

{

engine.Start();

Console.WriteLine("Car is running.");

}

}

בדוגמה החדשה, המחלקה **Car** אינה תלויה בסוג המנוע הספציפי (שתי פעימות או חשמלי), והיא משתמשת בממשק  **IEngine** כדי להפעיל את המנוע. השימוש ב DIP מאפשר גמישות רבה יותר ואפשרות לשנות את סוג המנוע בקלות מבלי לשנות את הקוד של הרכב (**Car**)

# סיכום עקרונות S.O.L.I.D:

עקרונות S.O.L.I.D לא נועדו לפתור בעיה ספציפית בעצמם.

הם מספקים הנחיות ושיטות עבודה מומלצות לעיצוב תוכנה הניתנת לתחזוקה, לבדיקה, להרחבה ולשימוש חוזר.

עקרונות אלו מתייחסים לבעיות נפוצות שעומדות בפני מפתחים בעת פיתוח ותחזוקה של מערכות תוכנה, כגון מורכבות קוד, Tight coupling וחוסר גמישות.

כמה בעיות נפוצות שעקרונות S.O.L.I.D עוזרים לפתור כוללות:

1. **מורכבות הקוד:** קוד מורכב יכול להיות קשה להבנה, לשינוי ולתחזוקה. עיקרון האחריות היחידה (SRP) עוזר להתמודד עם זה על ידי קידום הרעיון של קיום מחלקות או מודולים עם אחריות אחת ויחידה שמוגדרת היטב. הקפדה על עיקרון תעזור להפחית את מורכבות הקוד ותקל על ההבנה, התחזוקה והשינוי.
2. **Tight** **coupling (צימוד הדוק):** צימוד הדוק בין רכיבים יכול להפוך את המערכת לפחות גמישה וקשה לשינוי או הרחבה. עקרון ההיפוך התלות (DIP) ועקרון הפרדת הממשק (ISP) עוזרים לטפל בכך על ידי קידום צימוד רופף והיפוך תלות. על ידי הסתמכות על הפשטות וממשקים במקום יישומים קונקרטיים, ניתן לנתק רכיבים, מה שמאפשר שינוי והרחבה קלים יותר.
3. **חוסר יכולת בדיקה:** קוד שקשה לבדוק יכול להוביל לאיכות קוד נמוכה יותר ולסיכון מוגבר לבאגים. עקרון ה (DIP) ועיקרון ה (SRP) תורמים לשיפור יכולת הבדיקה על ידי קידום קוד מודולרי, מקושר באופן רופף, שניתן לבודד ולבדוק בקלות במנותק.
4. **חוסר הרחבה:** מערכות שקשה להרחיב יכולות לעכב את היכולת להסתגל לדרישות המשתנות. העיקרון ה (OCP) עוזר להתמודד עם זה על ידי קידום קוד שפתוח להרחבה אך סגור לשינוי. על ידי עיצוב קוד שניתן להרחבה בקלות מבלי לשנות את הקוד הקיים, המערכת יכולה להיות גמישה וניתנת להתאמה.
5. **שכפול קוד וחוסר שימוש חוזר:** קוד משוכפל עלול להוביל לבעיות תחזוקה וחוסר עקביות. עקרון הDon't Repeat Yourself (DRY) - למרות שאינו חלק מעקרונות ה S.O.L.I.D המקוריים, מוזכר לעתים קרובות יחד איתם. עקרון זה מעודד מפתחים להימנע משכפול קוד ומקדם שימוש חוזר בקוד. על ידי ביצוע עקרון האחריות היחידה (SRP) ועקרונות S.O.L.I.D אחרים, ניתן לעצב קוד בצורה מודולרית הניתן לשימוש חוזר.

בסך הכל עקרונות SOLID אינם פותרים בעיות ספציפיות באופן ישיר, הם מספקים הדרכה ושיטות עבודה מומלצות לתכנון מערכות תוכנה שקל יותר לתחזק, לבדוק, להרחיב ולהבין. על ידי יישום העקרונות הללו, מפתחים יכולים להתמודד עם אתגרים נפוצים וליצור פתרונות תוכנה חזקים ומתאימים יותר.

# תוספות שאינן חלק מ SOLID:

## DRY

מייצג "Don't repeat yourself", "אל תחזור על עצמך". זהו עיקרון המדגיש הימנעות משכפול קוד ומעודד מפתחים לכתוב קוד לשימוש חוזר וניתן לתחזוקה.

עקרון ה- DRY מציע שקוד צריך להיות מודולרי ומאורגן באופן המאפשר שימוש חוזר קל. במקום לשכפל קוד במספר מקומות, יש לפשט את הפונקציונליות הנפוצה לפונקציות, מחלקות או מודולים הניתנים לשימוש חוזר.

גישה זו מקדמת יעילות קוד, מפחיתה את הסבירות להכנסת באגים ומפשטת תחזוקה ושינויים עתידיים.

על ידי הקפדה על עקרון DRY, מפתחים שואפים לכתוב קוד תמציתי, יעיל וניתן לתחזוקה על ידי ביטול יתירות וקידום שימוש חוזר בקוד.

## KISS

עקרון KISS, הידוע גם בשם "Keep It Simple, Stupid" או "Keep It Short and Simple", הוא עיקרון עיצובי המצביע על כך שפשטות צריכה להיות מטרת מפתח בעת תכנון מערכות או פתרונות.

העיקרון מדגיש כי יש להימנע ממורכבות מיותרת, ולאמץ פשטות כדי להבטיח בהירות, יעילות וקלות שימוש.

עקרון זה, מיושם לעתים קרובות בתחומים שונים, כולל הנדסה, פיתוח תוכנה, עיצוב ממשק משתמש ופתרון בעיות באופן כללי.

על ידי שמירה על דברים פשוטים, קל יותר להבין, לתחזק ולתקשר רעיונות או פתרונות.

## YAGNI

מייצג "You Ain't Gonna Need It" זהו עיקרון בפיתוח תוכנה שמייעץ שלא ליישם פונקציונליות או תכונות עד שהם באמת נחוצים.

העיקרון מציע שמפתחים צריכים להימנע מהוספת מורכבות למערכת על ידי ציפייה לדרישות עתידיות או פונקציונליות שאולי לא יהיה צורך או בשימוש.

במקום זאת, ההתמקדות צריכה להיות ביישום רק של התכונות שבאמת נחוצות בכדי לעמוד בדרישות הנוכחיות.

עיקרון זה, מעודד מפתחים לתעדף פשטות ולהימנע מפתרונות הנדסת יתר.

על ידי הקפדה על עיקרון זה, מפתחים יכולים לחסוך זמן ומאמץ על ידי אי עבודה על תכונות מיותרות והפחתת הסיכון להכנסת באגים או שמירת קוד שאינו בשימוש.

עם זאת, חשוב להגיע לאיזון ולשקול יכולת הרחבה והרחבה עתידיים בעת יישום עקרון YAGNI, מכיוון שרמה מסוימת של גמישות עשויה להיות נחוצה כדי להתאים לשינויים עתידיים.