בס"ד

סיכום Design Patterns:

כפיר גולדפרב

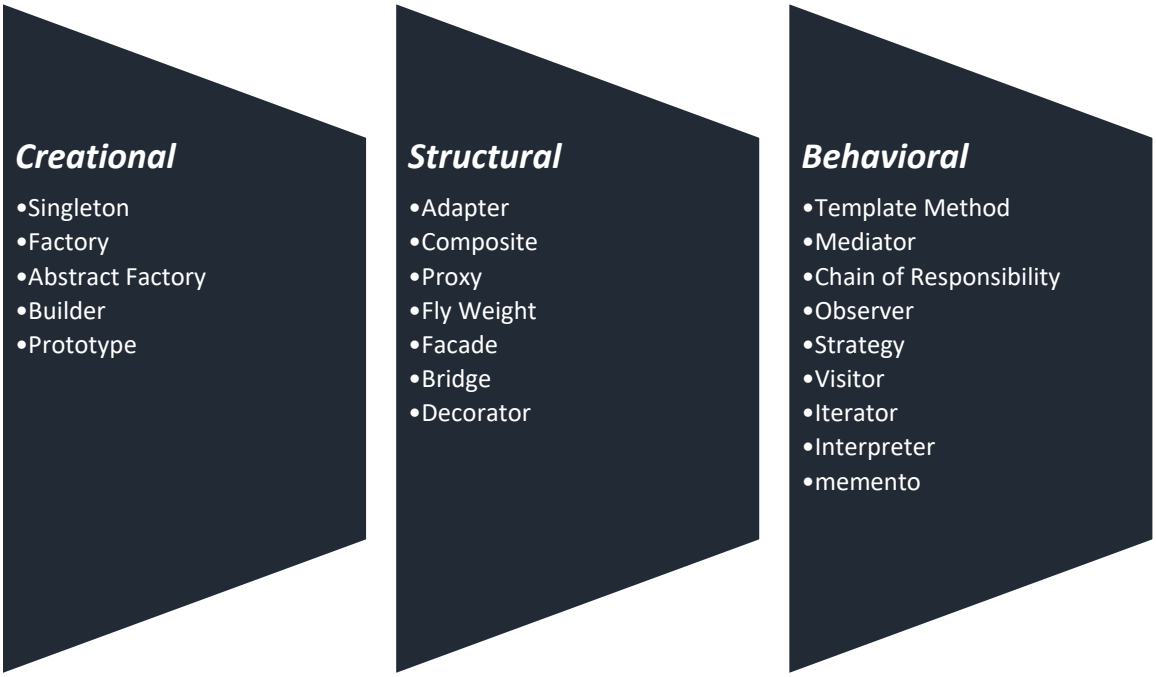
עיצוב תבניות הוא תחום בתכנות מונחה עצמים ומטרתו לתאר פתרונות לבעיות שכיחות בהנדסת תוכנה.

עיצוב תבניות מתחלק לשלושה תחומים שונים:

Creational – דרכים שונות כדי ליצור אובייקטים.

Structural – הקשרים השונים בין האובייקטים.

Behavioral – האינטראקציות השונות והתקשורת בין האובייקטים השונים.



Anti-patterns – מונח קוד גרוע

בס"ד

Creational Patterns

Singleton:

כאשר יש לנו מספר אובייקטים אשר מצביעים לאותו אובייקט (המקום בזכרון) וכאשר נשנה אחד מהם זה ישנה את כולם.

איך בונים את מחלקת Singleton?

נבנה מחלקה שמקבלת אובייקטים, המחלקה תעשה בדיקת getInstacne אם היא תראה שכבר קיים אובייקט כזה, תחזיר את האובייקט (כלומר את המקום בזכרון) במידה והאובייקט לא קיים המחלקה תיצור אותו ותחזיר את המיקום החדש.

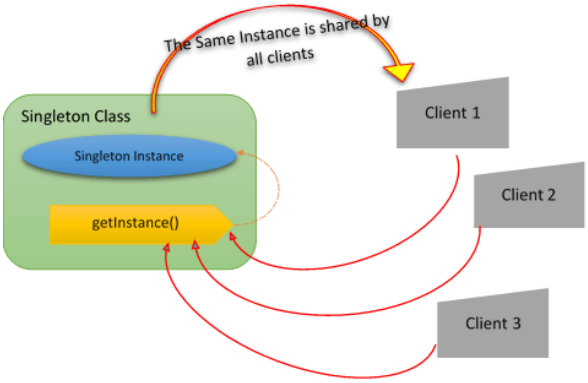


Figure 1: Singleton Design Pattern Overview

מתי משתמשים?

כאשר רוצים להשתמש באותה מחלקה בהרבה מקומות ואנחנו לא רוצים שהיא תתנהג בצורה שונה, לדוגמה מערכת שמבצעת התחברות למשתמשים, דוגמה נוספת שרת.

קוד:

```
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
    private LazyInitializedSingleton(){}
    public static LazyInitializedSingleton getInstance() {
        if(instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

מה קורה כשיש הרבה *threads*?

יכול להיות שיבצר אובייקט מ-*thread* אחד אבל כשסיים לעבוד, נוצר עוד אובייקט במקום אחר בזכרון, עוד אפשרות היא שאולי שני *threads* במקביל פתחו שני אובייקטים בשני מקומות שונים בזכרון.

איך נפתור את הבעיה? ניתן לפתור בעזרת *synchronized* כך:

```
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
    private LazyInitializedSingleton(){}
    public static synchronized LazyInitializedSingleton getInstance() {
        if(instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

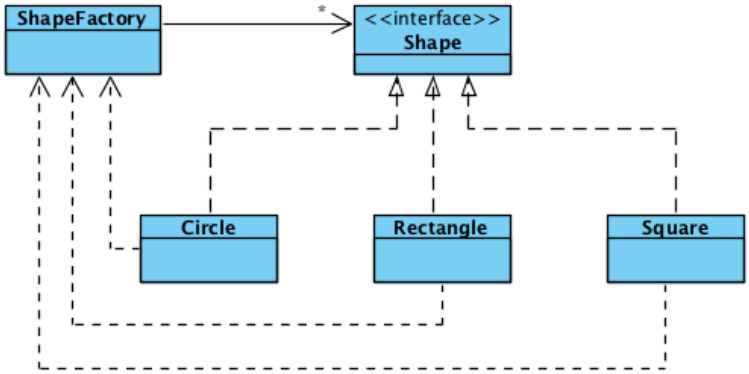
אבל פתרון זה יפגע בזמן ריצה של המערכת כיוון שה-*synchronized* הוא על כל הפונקציה וכל קריאה אליה ייקח הרבה זמן ריצה, ולכן נרצה לפתור את זה בדרך הבאה:

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;
    private ThreadSafeSingleton(){}
    public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {
        if(instance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if(instance == null) {
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
}
```

ובכך פתרנו את הבעיה אבל לא פגענו בזמן ריצה, למה יש שני תנאים של *instance == null*? רק במידה והוא *null* יהיה *synchronized*.

Factory:

כאשר אנחנו לא רוצים שיהיה מבחוץ גישה לבנאי של המחלקה, במקרה זה נגדיר מחלקה חיצונית מסויימת שהוא תבנה את המחלקה הנדרשת (לפי קלט מסויים שתקבל מהמשתמש תדע איזה אובייקט ליצור).



קוד:

```
public class ShapeFactory {

    public Shape getShape(String shapeType) {
        if(shapeType == null) {
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
    }

}
```

Prototype:

כאשר אנחנו רוצים להעתיק אובייקט מסויים מלא פעמים ולא נרצה להשתמש ב-shallow copy או משיכה מ-data base מכיוון שזה דורש הרבה זמן ריצה, נוכל לקחת את אותו אובייקט שכבר יש לנו, לשכפל אותו, לשנות את המיקום שלו בזיכרון ולשנות רק את הנתונים והשדות שאנחנו רוצים לשנות.

קוד:

נתונה מחלקה אבסטראקטית (נועדה רק לירושה) הבאה:

```
public abstract class Tree {
    public abstract Tree copy();
}
```

ונתונות שני מחלקות הבאות שיורשות מ-Tree:

```
public class PlasticTree extends Tree {

    @Override
    public Tree copy() {
        PlasticTree plasticTreeClone = new PlasticTree(this.getMass(), this.getHeight());
        plasticTreeClone.setPosition(this.getPosition());
        return plasticTreeClone;
    }
}
```

```
public class PineTree extends Tree {

    @Override
    public Tree copy() {
        PineTree pineTreeClone = new PlasticTree(this.getMass(), this.getHeight());
        pineTreeClone.setPosition(this.getPosition());
        return pineTreeClone;
    }
}
```

נוכל ליצור אובייקטים בצורה הבאה:

```
PlasticTree plasticTree = new PlasticTree(mass, height);
plasticTree.setPosition(position);
PlasticTree anotherPlasticTree = (PlasticTree) plasticTree.copy();
anotherPlasticTree.setPosition(otherPosition);
```

Structural Patterns

Adapter:

כאשר יש לנו מחלקה בנויה וגדולה שעובדת על תשתית מסויימת (נגיד xml) ואנחנו רוצים להשתמש באותה מחלקה אבל אנחנו עובדים על תשתית אחרת (נגיד json), במקום לשנות את כל המחלקה הקיימת הרבה זמן, נוכל לחסוך זמן וליצור מחלקה נוספת שיודעת להמיר לי בין התשתיות (למשל מ-json ל-xml ולהפך) ובכך נוכל להשתמש במחלקה הגדולה הבנויה מבלי לשנות אותה.

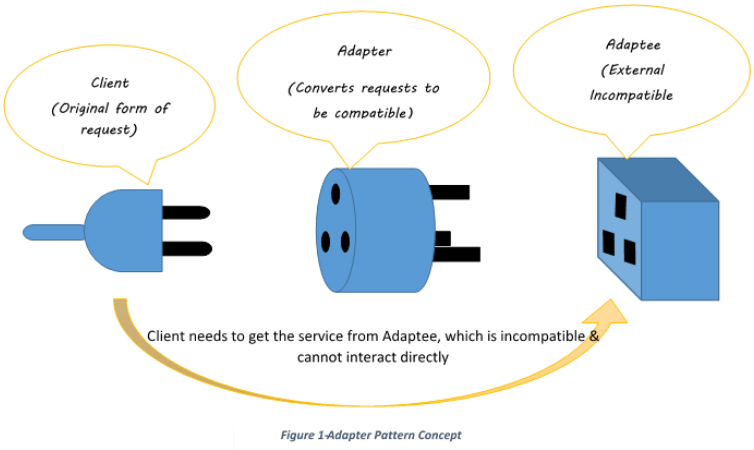
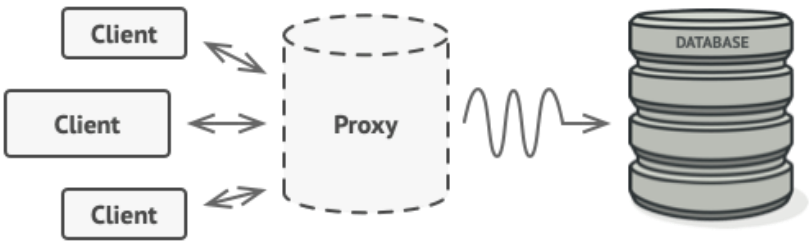


Figure 1-Adapter Pattern Concept

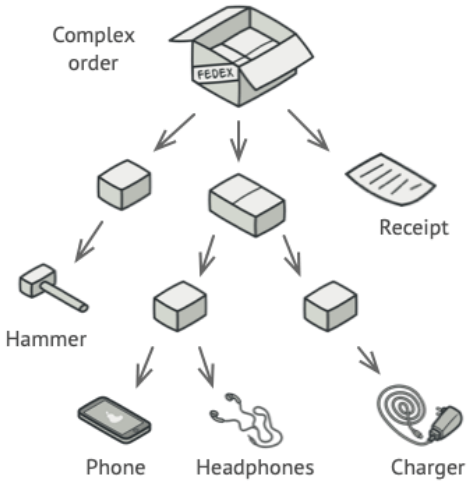
Proxy:

כאשר יש לנו מחלקה שנותנת שירותים מסויימים, ויש עוד הרבה מחלקות ומשתמשים אחרים שרוצים לקבל שירותו מאותה מחלקה אבל אנחנו לא רוצים ליצור עומס ישיר על אותה מחלקה, למשל ניתן לחשוב על שרת שנותן שירות להרבה משתמשים בו זמנית, כאשר יש יותר מידי בקשות לשרת הוא יכול לקרוס מרוב בקשות, ולכן על מנת לפתור את זה ניצור "מתחזה", כלומר מחלקה אחרת שתומכת באותם הפעולות בדיוק של המחלקה המקורית והמחלקה המתחזרת היא תיהיה זאת שמקבלת את הבקשות שירות מהלקוחות ומהמחלקות האחרות, אבל במקום לבצע אותם היא תשים את הבקשות על המתנה (wait) ולתבקש מהמחלקה המקורית לבצע את בקשות השירות ולהחזיר פלט למתחזה, המתחזה יחזיר את הפלט למבקש השירות.



Composite:

כאשר יש לנו מספר מחלקות הקשורות למחלקות אחרות (בצורה כמו של עץ) ואנחנו רוצים לעבור על כולם ולחשב מספר דברים, למשל בסופר יש מספר מחלקות (כלים, חד"פ, קצבייה ועוד) ולכל מחלקה יש את המוצרים שלה, ונרצה למשל לחשב את כל סכומי העלות של כל המוצרים בכל מחלקות אז נצטרך לעבור על כל מחלקה, וכל מחלקה בעצמה עוברת על כל המוצרים שלה וסוכמת את כל העלות של המוצרים, ברגע שסיימה את המעבר המחלקה מעלה את הסכום שקיבלה למחלקה למעלה והמחלקה למעלה (מחלקת האב) תסכום את כל שאר הסכומים שתקבל מכל שאר המחלקות.



Behavioral Patterns

Iterator:

דרך שבה ניתן לעבור על מבני נתונים של מחלקה מבלי לדעת איך המבנה הנתונים של המחלקה בנוי.

Observer:

כאשר אנחנו רוצים לקבל מידע מסויים על מחלקה בכל זמן (ולא לבצע לולאה אינסופית שבודקת סטטוס של המחלקה כל כמה זמן), נרצה להגיד למחלקה להחזיר תשובה אם סטטוס השתנה, דוגמה מהחיים האמיתיים – למשל נרצה לקנות פלאפון מחנות פלאפונים, במקום ללכת כל יום ולבדוק האם הפלאפון הגיע לחנות, פשוט נגיד לחנות להודיע לנו כאשר הפלאפון הגיע, דבר שיחסוך בסופו של דבר כח חישוב וזכרון.

קוד:

```
public interface Channel {
    public void update(Object o);
}
```

כל מחלקה שתרצה להתעדכן תממש את הממשק *Channel* ותממש את הפונקציה *update* בדרך שהיא תבחר וכך תוכל להתעדכן מהמחלקה שנותנת את השירות, המחלקה שנותנת את השירות היא אחראית על שימוש בפונקציה *update* של כל אחת מהמחלקות שקיימות אצלה (שמורות ברשימה),

קוד המחלקה שנותנת את השירות:

```
public class NewsAgency {
    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObServer(Channel channel) {
        this.channels.add(channel);
    }

    private void removeObServer(Channel channel) {
        this.channels.remove(channel);
    }

    public void setNews(String news) {
        this.news = news;
        for(Channel channel : this.channels) {
            channel.update(this.news);
        }
    }
}
```

State:

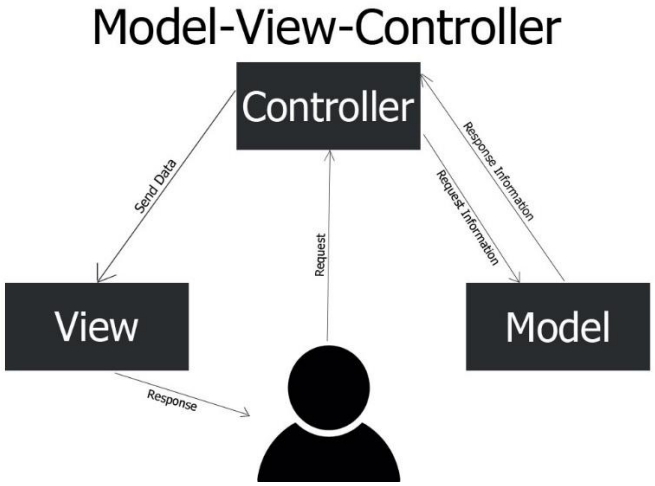
דרך שבה ניתן לשמור מצב מסויים של המערכת שלנו, וכל שאר המחלקות ידעו להתנהג בהתאם לאותו מצב.

Strategy:

כאשר יש לנו מספר דרכים לפתור את אותה הבעיה, ניתן למשל להפעיל שני מחלקות של אלגוריתמים שונות, נרצה להשתמש למשל במחלקה מסויימת אבל אחרי זמן מסויים נרצה להשתמש דווקא במחלקה האחרת, המטרה היא שיהיה אפשר להחליף את מחלקת האלגוריתמים בלי בעיות בכלל וכמעט בלי שינויים, שני המחלקות ממשות את אותם פונקציות ותכונות, אבל יכול להיות שבאלגוריתמים שונים.

עקרון MVC:

עקרון שבו מומלץ לחלק על מערכת לשלושה תחומים: **M**odel, **V**iew, **C**ontroller



מודל - Model:

אלגוריתמים, ייצוגי מידע, מבני נתונים או במילים אחרות "כל הקוד שמריץ את התוכנה, ובעזרתו התוכנה מקיימת את ייעודה".

תצוגה - View:

הדרך שבה מוצגת המידע מהמודל למשתמש, המטרה היא להפריד בין כל קוד של המודל של התוכנה לבין תצוגה שהמשתמש רואה בקוד אחר (לדוגמה - UI).

שליטה - Controller:

דרך שבא אפשר לשלוט על כל המידע ופעולות על המודלים, בעצם לבצע מוניפולציות על המודל ואפשרויות שונות על התצוגה (להחליט מה לראות מהתצוגה).

עקרונות S.O.L.I.D

***S**ingle Responsibility Principle, **O**pen/Closed Principle, **L**iskov Substitution Principle*

***I**nterface Segregation Principle, **D**ependency Inversion Principle*

S.O.L.I.D – עקרון שבו צריך לתכנן את הקוד שיהיה כתוב בצורה גנרית, דינאמית ואבסטרקטית על מנת ששינוי קטן במחלקה ספציפית לא יגרום לכך ששאר המחלקות לא יעבדו אחת עם השנייה.

הסוג המרכזי של הבעיות האלו הוא יותר מידי תלות בתוך הקוד.

עקרונות *S.O.L.I.D* באים לתת קווים מנחים שיגרמו לנו להמנע מלכתוב קוד עם בעיות הנ"ל.

S – Single Responsibility Principle:

למחלקה צריך להיות תחום אחריות אחד במערכת.

דוגמה:

המחלקה הבאה כתובה לא טוב:

```
public class User {
    private String name;
    private String password;
    private String email;

    public boolean setEmail(String email) {
        if(isValidEmail(email)) {
            this.email = email;
            return true;
        }
        return false;
    }

    public boolean setPassword(String password) {
        if(isValidPassword(password)) {
            this.password = password;
            return true;
        }
        return false;
    }

    private boolean isValidPassword(String password) {
        // check if password is valid
        return true;
    }

    private boolean isValidEmail(String email) {
        // check if email is valid
        return true;
    }
}
```

בס"ד

המחלקה הבאה כתובה טובה:

```
public class User {
    private String name;
    private String password;
    private String email;
    private UserFieldValidator userFieldValidator = new UserFieldValidator();

    public boolean setEmail(String email) {
        if(userFieldValidator.isValidEmail(email)) {
            this.email = email;
            return true;
        }
        return false;
    }

    public boolean setPassword(String password) {
        if(userFieldValidator.isValidPassword(password)) {
            this.password = password;
            return true;
        }
        return false;
    }
}

public class UserFieldValidator {
    public boolean isValidPassword(String password) {
        // check if password is valid
        return true;
    }

    public boolean isValidEmail(String email) {
        // check if email is valid
        return true;
    }
}
```

ניתן לראות שבמקום לכתוב שני פונקציות לבדיקת האימייל והסיסמה בתוך המחלקה *User*, בנינו מחלקה אחרת (תחום אחר) שאחראית על בדיקת האימייל והסיסמה וכך גם מחלקות אחרות יוכלו לגשת למחלקה זו במקרה הצורך.

בס"ד

O – Open/Closed Principle:

מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים.

דוגמה:

המחלקה הבאה כתובה לא טובה:

```
public class SumCalculator {

    private List<Shape> shapes;

    public SumCalculator(List<Shape> shapes) {
        this.shapes = shapes;
    }

    public double getSum() {
        double sum = 0;
        for(Shape s : shapes) {
            sum += getArea(s);
        }
        return sum;
    }

    public double getArea(Shape s) {
        if(s instanceof Square) {
            return Math.pow(((Square)s).getLength(), 2);
        } else if(s instanceof Circle) {
            return Math.PI * Math.pow(((Circle) s).getRadius(), 2);
        }
        return 0;
    }
}
```

הבעיה: מה יקרה כשאר נוסיף מחלקה הממשת את *Shape* למשל משולש? הקוד הוא בעייתי.

המחלקה הבאה כתובה טובה:

```
public class SumCalculator {

    private List<Shape> shapes;

    public SumCalculator(List<Shape> shapes) {
        this.shapes = shapes;
    }

    public double getSum() {
        double sum = 0;
        for(Shape s : shapes) {
            sum += s.getArea();
        }
        return sum;
    }
}
```

ונגדיר את האינטרפייס של *Shape* להיות עם הפונקציה *GetArea()*, כלומר:

```
public interface Shape {
    public double getArea();
}
```

בס"ד

L-Liskov Substitution Principle:

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעות בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל.

דוגמה:

נתונה מחלקה למלבן:

```
public class Rectangle implements Shape {

    private double height;
    private double width;

    public Rectangle(double width, double height) {
        this.height = height;
        this.width = width;
    }

    @Override
    public double getArea() {
        return width * height;
    }

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

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

המחלקה הבאה לא כתובה טוב (מחלקה של ריבוע):

```
public class Square extends Rectangle {

    public Square(double length) {
        super(length, length);
    }

    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);

    }

    public void  setWidth(double width) {
        super.setWidth(width);
        super.setHeight(width);
    }
}
public static void foo(Rectangle r) {
    r.setWidth(2);
    r.setHeight(3);
}
```

מה שיקרה בפונקציה *foo* זה שאם נשלח אליה מצביע לריבוע נקבל *bug* בהתבסס על זה שאנחנו מצפים שבהתקבל מלבן, השטח יהיה 6 אבל אם נשלח ריבוע השטח יהיה 4 ואז 9 וזה לא טוב מבחינה לוגית.

הפתרון: לא לעשות ירושה ולבנות מחלקה של ריבוע בפני עצמה בלבד.

I-Interface Segregation Principle:

יש לדאוג לממשקים מצומצמים:

- לא לאלץ למחלקה לממש ממשק שאין לה צורך בו.
- לדאוג לכימוס מרבי של המידע (שנתונים יהיה כמה שיותר ב-*private*).

דוגמה:

נתון הממשק הבא של צורה:

```
public interface Shape {
    public double getArea();
    public double getVolume();
}
```

הממשק הבא (משולש) בנוי בצורה לא טובה:

```
public class Triangle implements Shape {
    @Override
    public double getArea() {
        return 0;
    }

    @Override
    public double getVolume() {
        // ?????????
        return 0;
    }
}
```

מכיוון שהפונקציה *getVolume* אינה מוגדרת, ולכן הדרך הטובה לעשות את זה:

הפתרון:

```
public interface Shape {
    public double getArea();
}
```

```
public interface SolidShape extends Shape {
    public double getVolume();
}
```

ירושה של ממשק מממשק.

בס"ד

:D – Dependency Inversion Principle

מחלקות *height level* לא צריכות להשתמש באופן ישיר במחלקות *low level*.

High level – מחלקות שמשתמשות במחלקות אחרות.

Low level – מחלקות שלא משתמשות במחלקות אחרות.

דוגמה:

המחלקה הבאה כתובה לא טובה:

```
public class WritingManager {
    Printer printer;
    WritingManager(Printer printer) {
        this.printer = printer;
    }
    public void doWriting(String str) {
        printer.print(str);
    }
}
```

```
public class Printer {
    public void print(String str) {
        // print the string
    }
}
```

במקום זה עדיף לכתוב:

```
public interface ICanWrite {
    public void write(String str);
}
```

```
public class WritingManager {
    ICanWrite writeable;
    WritingManager(ICanWrite writeable) {
        this.writeable = writeable;
    }
    public void doWriting(String str) {
        writeable.write(str);
    }
}
```

```
public class Printer implements ICanWrite {
    @Override
    public void write(String str) {
        // print the string
    }
}
```

תכנות על פי חוזה DBC – Design By Contract:

עיצוב תוכנה המבוסס על פי מפרטים פורמלים מדוייקים וניתנים לאימות עבור ממשקים של רכיבים בתוכנה, רכיבי התוכנה הם טיפוסי נתונים אבסטרקטים הדורשים קיום של:

1. קבועים (*invariants*)
2. תנאים מוקדמים (*preconditions*)
3. תנאים מאוחרים (*postconditions*)

מפרטים אלו נקראים גם "חוזים", על פי שיטה זו, רכיבי התוכנה משתפים פעולה זה עם זה על בסיס "מחויבויות" ו"רווחים" הדדיים.

קרא עוד ב- תכנות על פי חוזה - ויקיפדיה.

תכנות הגנתי DP – Defensive Programming:

תכנות הדורש בדיקות של כל המצבים וכל מקרי הקצה, הרעיון הוא להיות מוגן מקלט בלתי צפוי, לכן אין להניח תקינות הקלט.

קיימים 3 כללים עקריים:

1. לעולם לא להניח קלט תקין.
2. להשתמש בסטנדרטים קבועים ומקובלים.
3. קוד פשוט וברור.

קרא עוד ב- תכנות הגנתי - ויקיפדיה.