

# הנדסת תוכנה

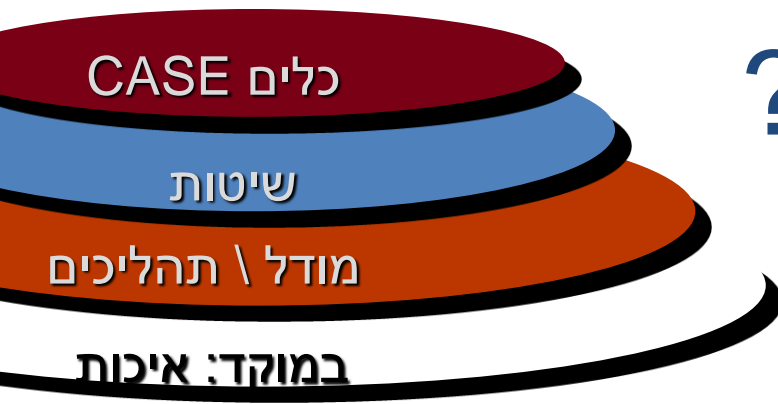
## 12. עקרונות תיכון מונחה עצמים

### SOLID OODP

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"  
- Martin Fowler



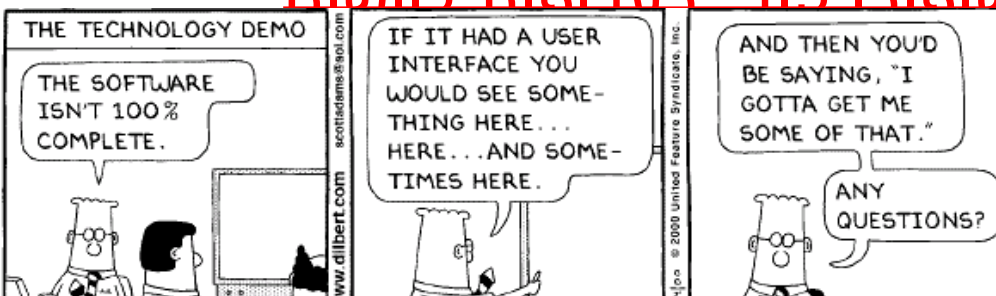
# מה היום?



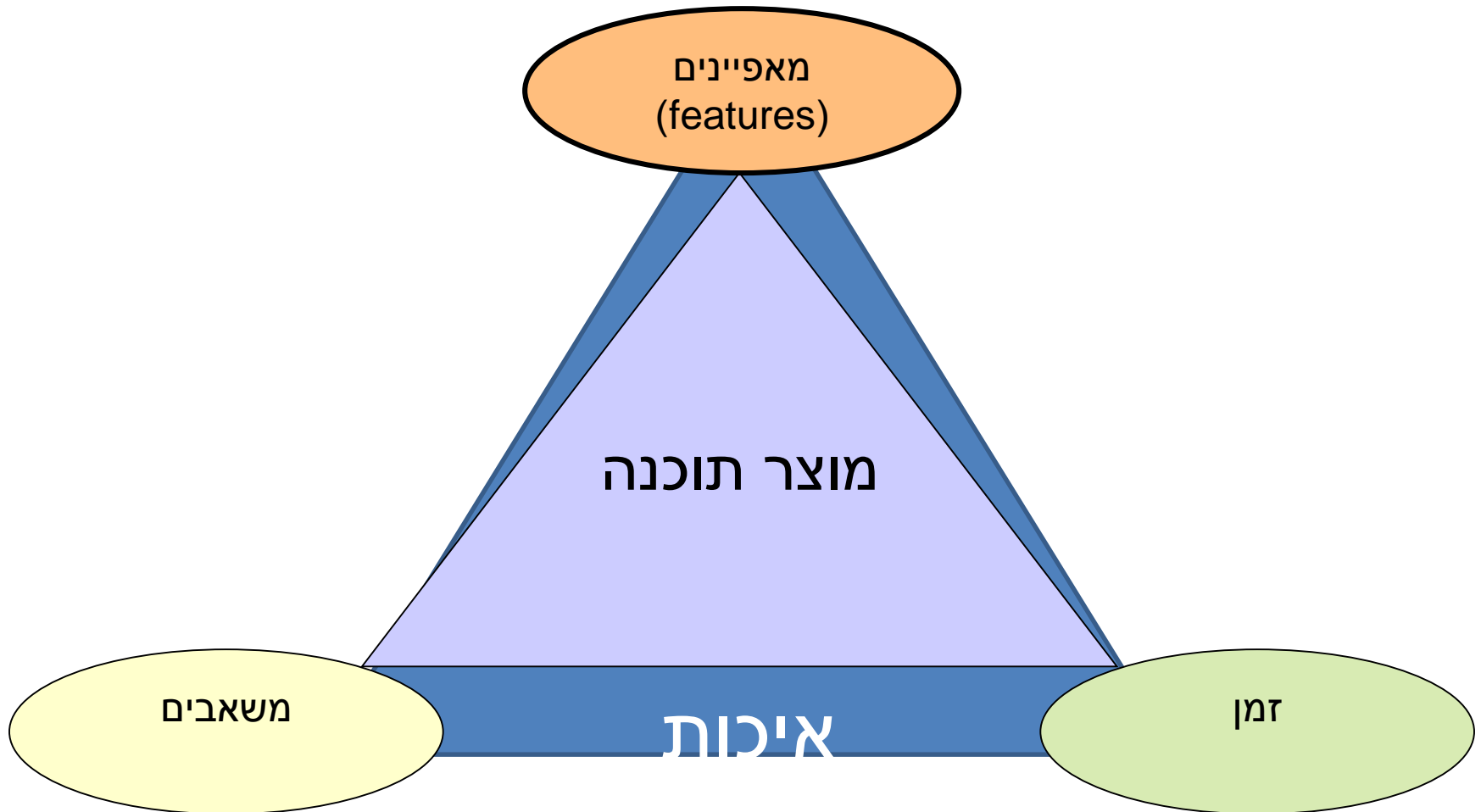
- ראינו:

- ארכיטקטורה, עקרונות-על
- בדיקות, תיכון מתמשך ושיפרוק בעיקר ברמת הקוד
- תבניות תיכון
- $\leq$  המטרה: מוצר איכותי שיכול להתפתח
- עוד על **איכות תוכנה**: עקרונות תיכון מונחה עצמים  
Object Oriented Design Principles  
הדגמת העקרונות (כולל בדיקות ותבניות תיכון)
- הרצאה 3\תרגיל:

– בפרויקט – סבב 3, סקר ~~שופטו כוד~~ ~~רעיונות לעשות~~  
~~עקרונות התיכון לסקרים~~



# תזכורת: פרויקט תוכנה:



# אנחנו רוצים להיות מהירים ב...

- הוספת תכונות בעתיד
- ריצה של המוצר
- התכונה שאנחנו עובדים עליה עכשיו
- $\leq$  לא תמיד אפשר הכל ביחד

# שאלות

- מהו קוד איכותי?
- איך כותבים אותו?
- האם בדיקות עוזרות?
- כיצד מגיעים לתיכון בדיק? עמיד לשינויים?
- היום: עקרונות תיכון מונחה עצמים
- רגע – מה זה "עצם"? OOP ? OOD ?

# עקרון על: מודולריות

- חלוקה

- של בעיות גדולות לבעיות קטנות הניתנות לניהול
- של קוד גדול ליחידות קטנות ופשוטות\מפשטות, כל אחת מספקת פונקציונאליות או מאפיין מסויים

- הרכבה

- שימוש חוזר; DRY
- אבל גם הפשטה YAGNI
- עשינו זאת ברמת הפרויקט (ניתוח\תיכון\תכנון)  
עכשיו ברמת הקוד

# Separation of concerns

- Functions
- Modules / Libraries
- Objects
- Layers / Components
- Services / Aspects / ...
- In general:  
[Programming Paradigms](#)



# תזכורת: Beck's Simple Design Rules

1. כל הבדיקות עוברות
2. ללא כפילויות (DRY)
3. מבטא את כוונת המתכנת
4. מינימום של מחלקות ומתודות

```
Collection {  
  ...  
  int size() {...}  
  boolean isEmpty() {...}  
}
```

כיצד נוריד כפילויות?

[New book by Corey Heines](#)



# תזכורת: פיתוח תוכנה מונחה עצמים

- אריזה של נתונים ופעולות יחד – Encapsulation  
– תקשורת דרך שליחת הודעות
- סידור טיפוסים נתונים בהיררכיות – Inheritance
- התנהגויות משתנות לפי הטיפוס – Polymorphism
- Alan Kay: [The big idea is "messaging"](#)

# למה פיתוח מונחה עצמים?

- חלוקה לאובייקטים
  - מיפוי יותר אינטואיטיבי של תחום הבעיה
  - אלו אובייקטים מופעלים ע"י המערכת?
- שימוש חוזר
  - פעולות בד"כ תלויות בנתונים אז בואו נחבר ביניהם
  - מחלקה כיחידת השימוש העיקרית
- הרחבתיות
  - שימוש ברכיבים קיימים
  - היררכיית אובייקטים המתאיצבת במהלך הפיתוח

# מהו קוד שקל לשנות?

- שינויים אינם גורמים לתוצאות בלתי צפויות
- שינוי קטן בדרישות לא מצריך שינוי גדול בקוד
- ניתן לעשות שימוש חזור בקוד קיים
- הדרך הקלה ביותר לשנות היא להוסיף קוד שקל לשנות

# מאפייני קוד קל לשינוי - TRUE [Metz]

- **Transparent** The consequences of change should be obvious in the code that is changing and in distant code relies upon it
- **Reasonable** The cost of any change should be proportional to the benefits the change achieves
- **Usable** Existing code should be usable in new and unexpected contexts
- **Exemplary** The code itself should encourage those who change it to perpetuate these qualities

# סימנים בקוד Martin: Design Smells

- Rigidity (קשיחות) – קשה לשנות
- Fragility (שבירות) – כשמשנים יש בעיות
- Immobility (נייחות) – רוצים לקחת למקום אחר
- Viscosity (צמיגות) – תקועים עם הארכי', סביבה
- Needless complexity – למשל עודף כלליות
- Needless repetition – למשל בהעברת קוד
- Opacity (עמימות) – קוד לא ברור

"ארץ ישראל – היא ארץ של  
סימנים, יש בה סימנים של נפט,  
סימני גז, סימנים של נחושת ועוד  
סימנים מעודדים רבים", לוי אשכול.

# Broken Window Theory





# Broken Window Theory





# SOLID

Software Development is not a Jenga game



# עקרונות תומכים במודולאריות, Martin -

## SOLID

- The Single-Responsibility Principle - **SRP** - A class should have only one reason to change.
- The Open-Closed Principle - **OCP** - A class should be extensible without requiring modification
- The Liskov Substitution Principle - **LSP** - Derived classes should be substitutable for their base classes
- The Dependency Inversion Principle - **DIP** - Depend upon abstractions. Do not depend upon concretions
- The Interface Segregation Principle - **ISP** - Many client specific interfaces are better than one general purpose interface.



# SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

# Single-Responsibility Principle (SRP)

*A class should have only one reason to change (Martin).*

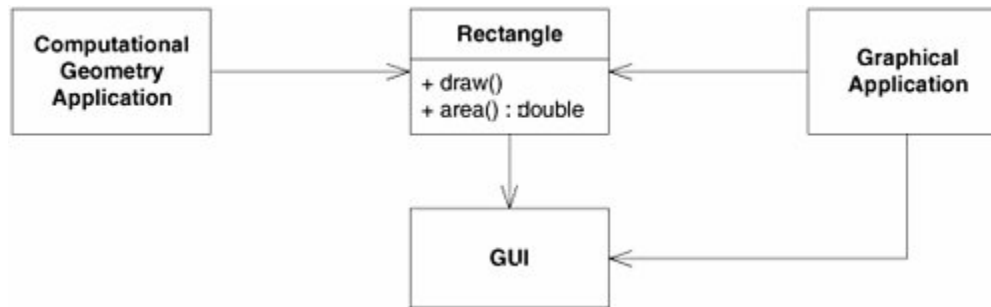
- אחריות היא סיבה לשינוי
- התפקידים שיש למחלקה הם צירי שינוי. אם יש לה שני תפקידים הם **צמודים** ביחד **ומשתנים** ביחד
- עיקרון פשוט אך לא תמיד קל להגיע אליו
- הדרך: האצלה (delegation)



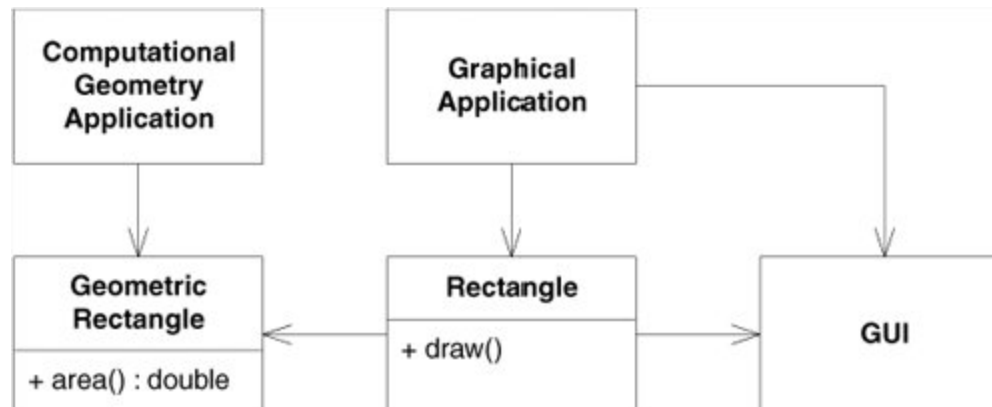
על אילו סימנים התגברנו?

# SRP

• מה הבעיה כאן?



• פתרון:





```
public class PrintServer
{
    public string CreateJob(PrintJob data) { //...
    }
    public int GetStatus(string jobId) { //...
    }
    public void Print(string jobId, int startPage, int endPage) { //...
    }

    public List<Printer> GetPrinterList() { //...
    }
    public bool AddPrinter(Printer printer) { //...
    }

    public event EventHandler<JobEvent> PrintPreviewPageComputed;

    public event EventHandler PrintPreviewReady;

    // ...
}
```

```
public class PrintServer {
```

```
    public string CreateJob(PrintJob data) { //...  
    }
```

```
    public int GetStatus(string jobId) { //...  
    }
```

```
    public void Print(string jobId, int startPage, int endPage) { //...  
    }
```

```
}
```

```
public class PrinterList {
```

```
    public List<Printer> GetPrinterList() { //...  
    }
```

```
    public bool AddPrinter(Printer printer) { //...  
    }  
}
```



# SRP Violation Smells

- שמות עם And
- Manager Class
- מתודות ארוכות ובמיוחד עם קינון ..if, while
- הערות ו- regions
- מחלקה עם רכיבי קשירות
- קשה לבדיקה
- נשברת כשמתחילים לשנות



# DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

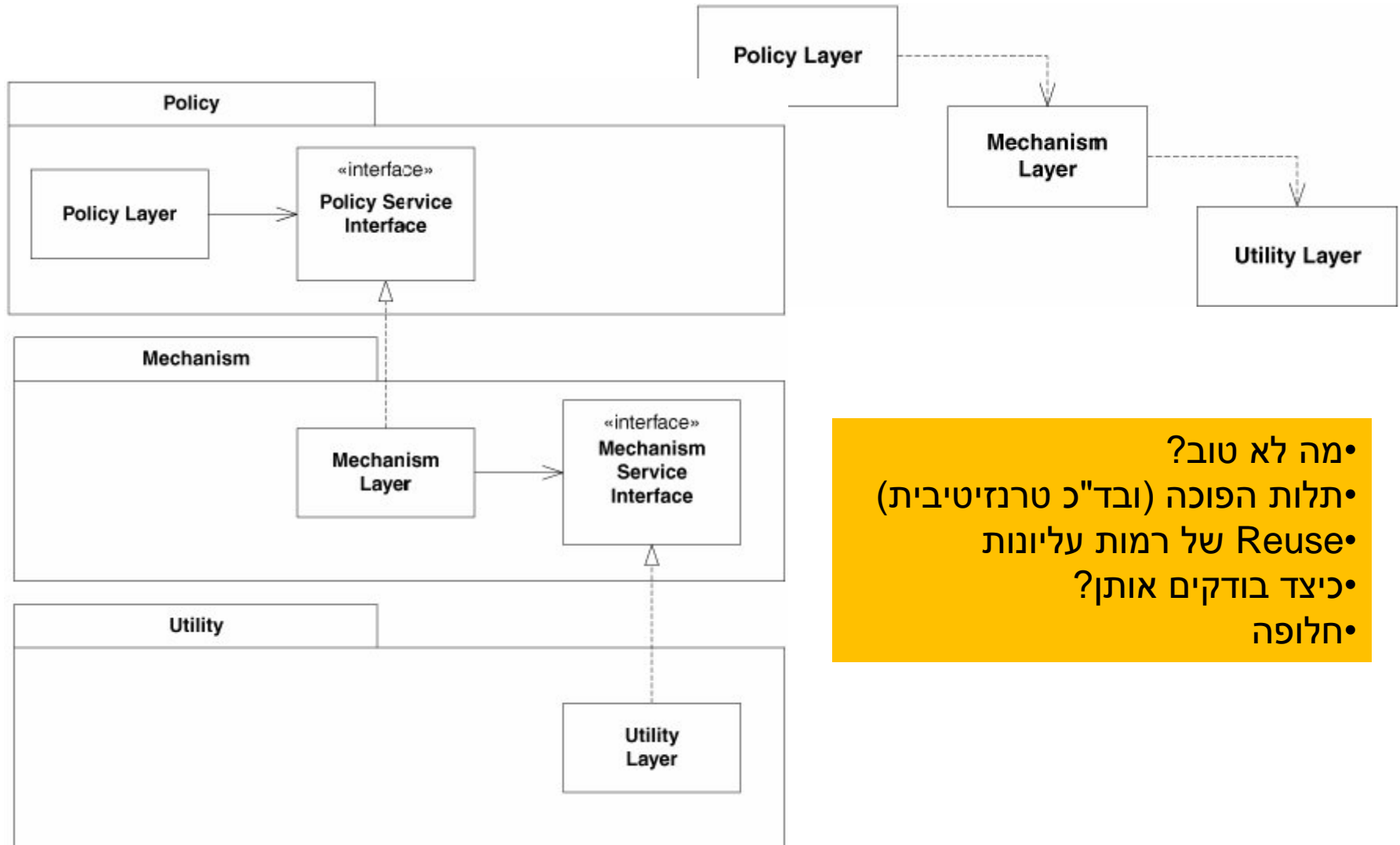


# Dependency Inversion Principle (DIP)

*Depend upon abstractions. Do not depend upon concrete implementations (Martin).*

- מחלקות ב"רמה גבוהה" אינן צריכות להיות תלויות במחלקות ב"רמה נמוכה"
- הפשטות צריכות להיות מנותקות ממימוש מסוים ומפרטים
- אם ההפשטה תלויה במימוש אז יש לנו תלות הפוכה (לא טוב)
- הדרך: גישה לאובייקטים (מופעים) דרך ממשקים ומחלקות מופשטות
- אם הרמות העליונות אינן תלויות במימושים מסוימים, איך בעצם מחברים את הרמות התחתונות?
- מאפשר הפרדה בבדיקות

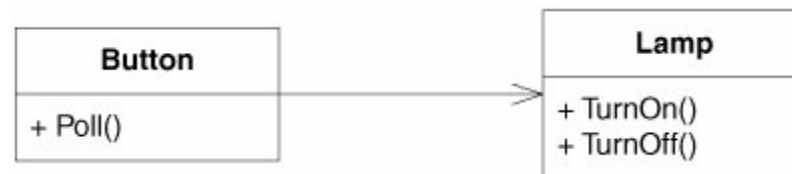
# DIP



- מה לא טוב?
- תלות הפוכה (ובד"כ טרנזיטיבית)
- Reuse של רמות עליונות
- כיצד בודקים אותן?
- חלופה

# DIP - Example

- ברצונינו לכתוב תוכנית שבה "כפתור" מזהה לחיצה ומפעיל מנורה
- איזה תיכון הייתם מציעים?



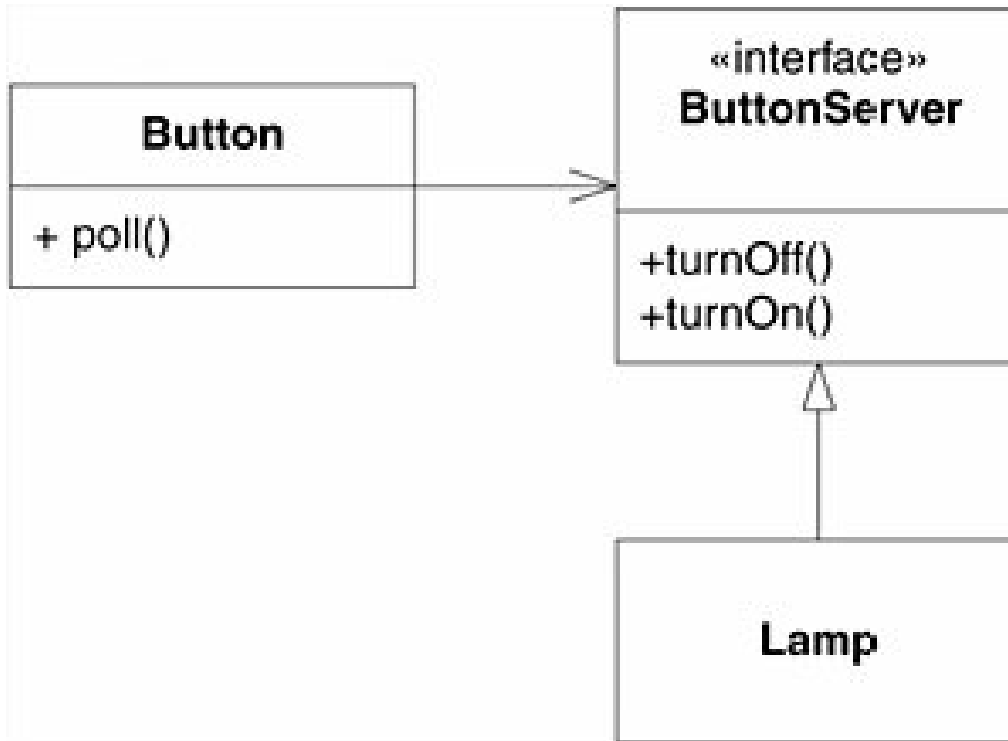
# DIP - Violation

```
public class Button
{
    private Lamp lamp;

    public void Poll()
    {
        if (/*some condition*/)
            lamp.TurnOn();
    }
}
```

- מה לא טוב?
- השרות ברמה הגבוהה, תלוי במימוש ברמה תחתונה, הפרה של DIP!
- מה אם נרצה שהכפתור ישלוח על מנוע?

# DIP - Solution



- עכשיו הכפתור מגדיר את התלות!
- האם לא יצרנו עכשיו תלות של מנורה בכפתור?
- רק תלות בממשק! אפשר לשנות את השם וגם להוציא את הממשק לחבילה נפרדת, למשל **SwitchableDevice**
- אז בסוף כיצד מקשרים את המנורה לכפתור?

# DIP – Frameworks

- התפתח היצע רחב של חבילות תשתית בנושא
- “new” is a smell
- למשל [Guice](#), Spring ל-Java ו-[Autofac](#), MEF ל-.Net
- ```
public class Client {  
  
    public void go() {  
        Service service = ServiceFactory.getInstance();  
        service.go();  
    }  
}
```
- [מצגת Guice \(\\*\)](#)
- הערה: פיתוח בדיקות עם Test double מצריך ומעודד עבודה עם ממשקים

# DIP – Frameworks - Example

```
var builder = new ContainerBuilder();

builder.Register<Straight6TwinTurbo>().As<IEngine>().FactoryScoped();

builder.Register(c => new Skyline(c.Resolve<IEngine>(), Color.Black))
    .As<ICar>()
    .FactoryScoped();

////////////////////////////////////

using (var container = builder.Build())
{
    var car = container.Resolve<ICar>();
    car.DriveTo("Byron Bay");
}
```

• רוב החבילות תומכות גם בקבצי  
קינפוג (XML)  
• כמה רחוק אפשר לקחת את  
העיקרון (string)





# INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?



# Interface Segregation Principle (ISP)

*Many client-specific interfaces are better than one general purpose interface (Martin).*

- אם יש מחלקה שיש לה כמה שימושים, כדאי ליצור ממשק נפרד (ורזה) לכל שימוש.
- אחרת, כל המשתמשים חולקים ממשק גדול, אבל רובם משתמשים בחלק קטן ובכל זאת תלויים בשאר
- החלופה היא שהשימוש במחלקה יאורגן בקבוצות של שיטות שקשורות אחת לשנייה, כל קבוצה כזו היא מחלקה.
- באופן כזה **לקוחות** משתמשים בממשקים קטנים ועקיבים (קוהרנטיים) יותר
- לעומת זאת, ממשקים "שמנים" מובילים לצמידות ותלות אקראית

# ISP

- עדיפות לממשקים "רזים" על פני "שמונים"  
– אם אפשר, מוגדרים ע"י הלקוח
- אם היינו צריכים לכתוב ממשקים (DIP) עבור  
דוגמת המלבנים?  
GraphicRectangle –  
GeometricRectangle –

```

public interface Animal {
    void fly();
    void run();
    void bark();
}

public class Bird implements Animal {
    public void bark() { /* do nothing */ }
    public void run() {
        // write code about running of the bird
    }
    public void fly() {
        // write code about flying of the bird
    }
}

public class Cat implements Animal {
    public void fly() { throw new Exception("Undefined cat property"); }
    public void bark() { throw new Exception("Undefined cat property"); }
    public void run() {
        // write code about running of the cat
    }
}

public class Dog implements Animal {
    public void fly() { }
    public void bark() {
        // write code about barking of the dog
    }

    public void run() {
        // write code about running of the dog
    }
}

```

```

public interface Flyable {
    void fly();
}
public interface Runnable {
    void run();
}
public interface Barkable {
    void bark();
}
public class Bird implements Flyable, Runnable {
    public void run() {
        // write code about running of the bird
    }

    public void fly() {
        // write code about flying of the bird
    }
}
public class Cat implements Runnable{

    public void run() {
        // write code about running of the cat
    }
}
public class Dog implements Runnable, Barkable {

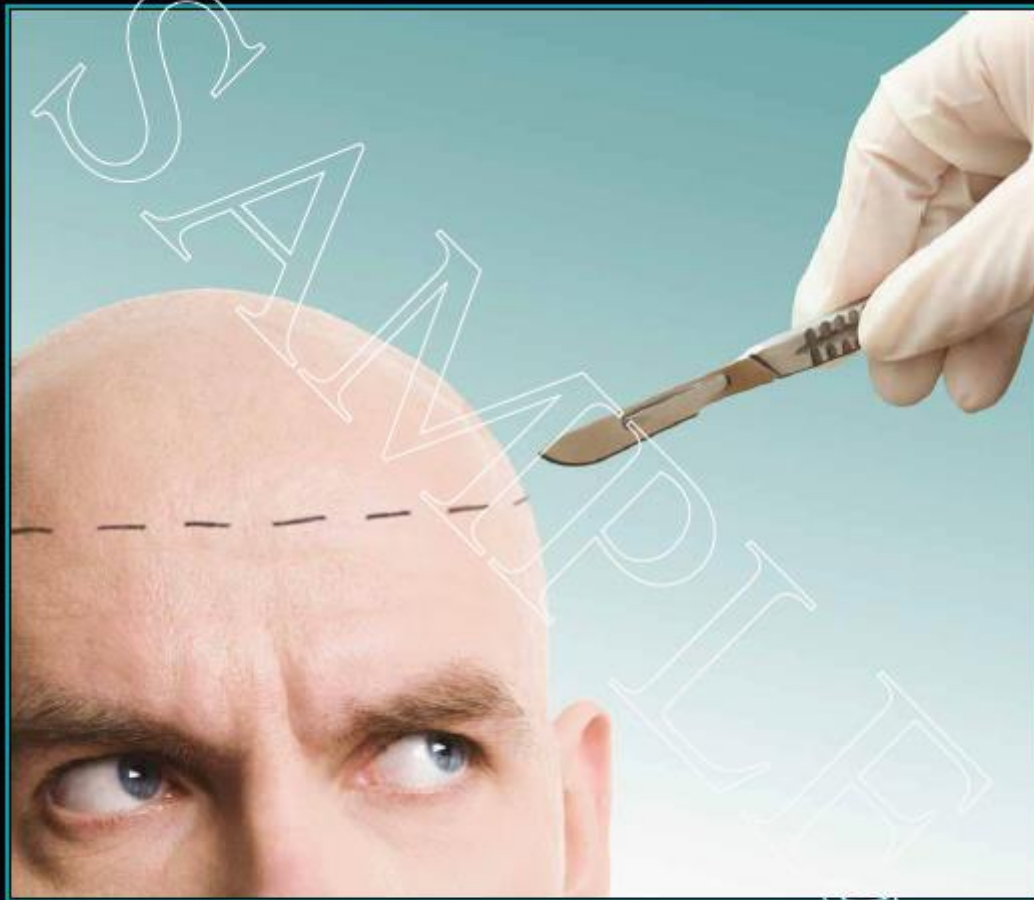
    public void bark() {
        // write code about barking of the dog
    }

    public void run() {
        // write code about running of the dog
    }
}

```

# ISP "Smells"

- NotImplementedException – בעייתי גם עם LSP (בהמשך)
- לקוח שמשתמש בחלק קטן מממשק של מחלקה
- Comments / Comment out / C# Regions
- Testability Issues



# OPEN CLOSED PRINCIPLE

Brain surgery is not necessary when putting on a hat.

# Open-Closed Principle (OCP)

*A module (class) should be open for extension but closed for modification (Bertrand Meyer).*

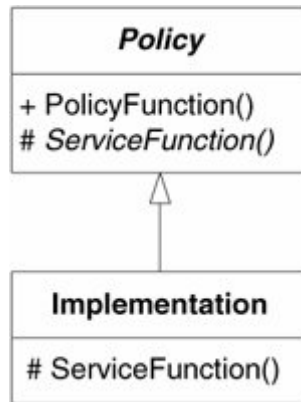
- מחלקות צריכות להיכתב כך שניתן להרחיב אותן ללא צורך לשנות אותן
- כך אפשר לשנות את התנהגות של מחלקה ללא שינויים לקוד המקור וללא פגיעה ב"לקוחות"
- בכדי להרחיב מערכת (כתוצאה משינוי בדרישה), יש להוסיף קוד ולא לשנות את הקיים
- המנגנון העיקרי למימוש: Abstraction (Interface, Abstract Class, ו-subtype polymorphism)

# OCP

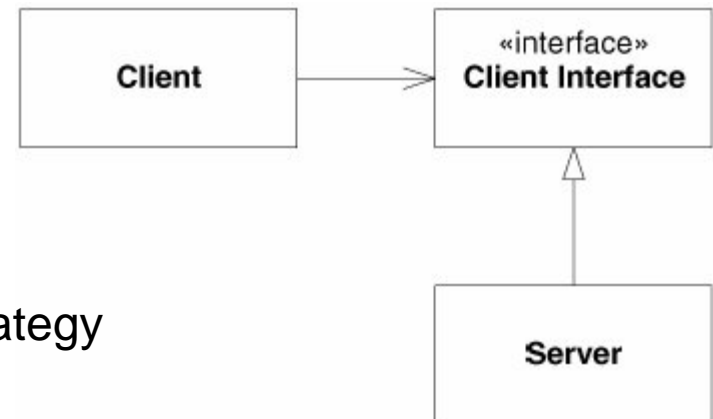
- מה הבעיה בתלות כאן?



- פתרונות OO (שימו לב לשמות):



Design Patterns:  
Template Method <-> Strategy





# Violating OCP

- אפליקציה גרפית פרוצדורלית (C) לציור צורות
- האם תוספת מצריכה שינוי בקוד הספרייה? בקוד הלקוח?



Shapes.c

```

--shape.h-----
enum ShapeType {circle, square};
struct Shape
{
    ShapeType itsType;
};
--circle.h-----
struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};
void DrawCircle(struct Circle*);
--square.h-----
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};
void DrawSquare(struct Square*);

```

```

--drawAllShapes.cc-----
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;

            case circle:
                DrawCircle((struct Circle*)s);
                break;
        }
    }
}

```

היכן ההפרה של OCP?  
אלו סימנים (ריחות) עולים כאן?

# Violating OCP: OOP Solution

```
public interface Shape
{
    void Draw();
}
public class Square : Shape
{
    public void Draw()
    {
        //draw a square
    }
}
public class Circle : Shape
{
    public void Draw()
    {
        //draw a circle
    }
}

public void DrawAllShapes(IList shapes)
{
    foreach(Shape shape in shapes)
        shape.Draw();
}
```

- כיצד תתבצע הרחבה עכשיו?
- האם עמדנו בעיקרון?
- על אלו סימנים התגברנו?

# האם זהו פתרון מושלם?

- מה קורה אם מוסיפים דרישה שכל העיגולים יצוירו לפני המלבנים?
- לעקרון הסגירות יש גבולות!
- כיצב קובעים את הגבולות?
- "Fool me once, shame on you. Fool me twice, shame on me."  
– אם נפגענו פעם אחת, נגן על עצמינו מסוג כזה של פגיעה
- אלו הרגלים יכולים לעזור?
  - בדיקות מקדימות (TDD)
  - מחזורי פיתוח קצרים
  - עדיפות לפיתוח מאפיינים עיקריים (על פני תשתית) והצגה ללקוח
- כיצד נסגור את DrawAllShapes שוב בצורה מופשטת?
- ב-C# יש לנו כבר מנגנון: Comparable

# Closing again (?)

```
public interface Shape : IComparable
{
    void Draw();
}
public class Circle : Shape
{
    public void Draw()
    {
        //draw a circle
    }

    public int CompareTo(object o)
    {
        if(o is Square)
            return -1;
        else
            return 0;
    }
}
public void DrawAllShapes(ICollection shapes)
{
    shapes.Sort();
    foreach(Shape shape in shapes)
        shape.Draw();
}
```

•האם הכל סגור?

# And more...

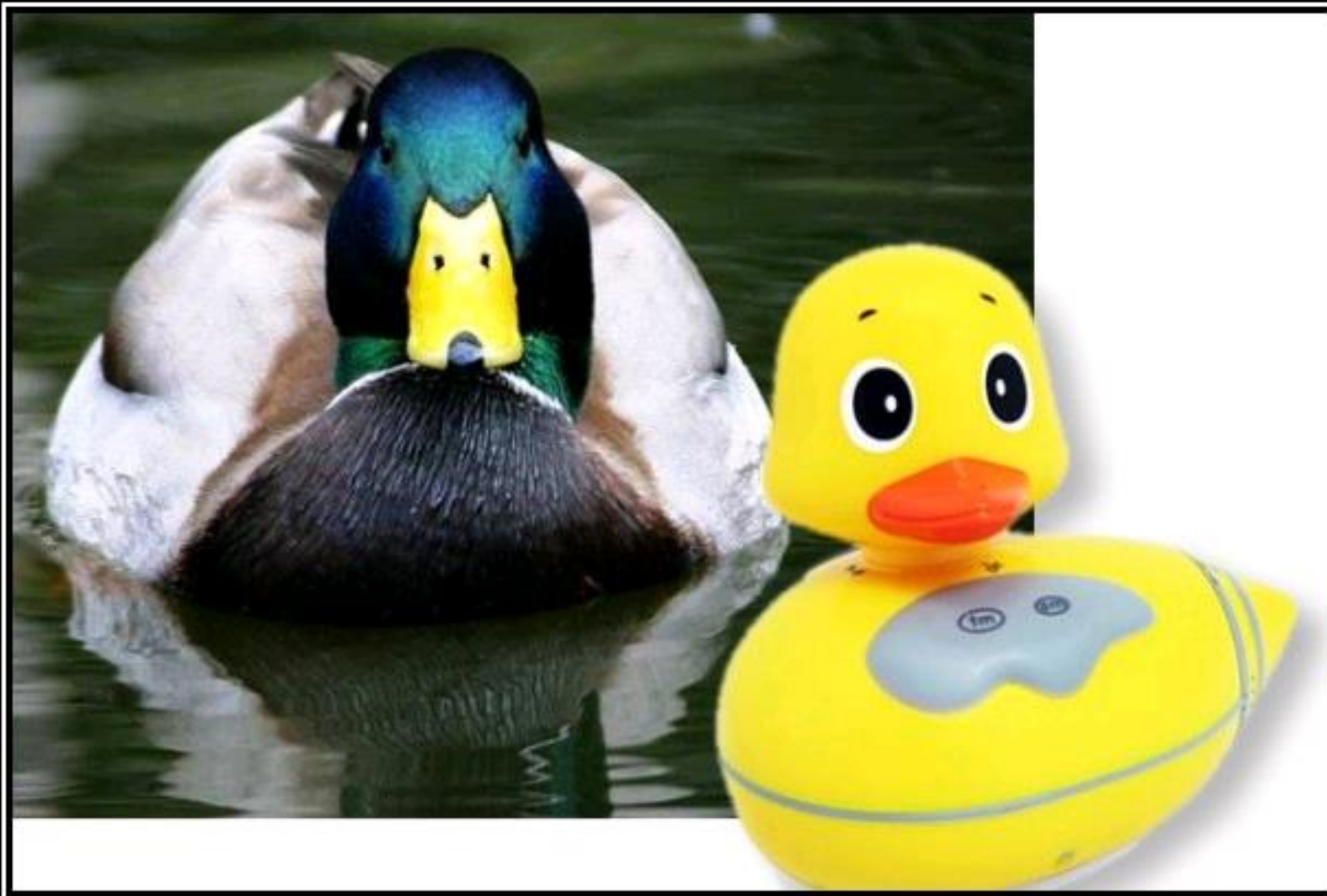
```
public class ShapeComparer : IComparer
{
    private static Hashtable priorities = new Hashtable();
    static ShapeComparer()
    {
        priorities.Add(typeof(Circle), 1);
        priorities.Add(typeof(Square), 2);
    }
    private int PriorityFor(Type type)
    {
        if(priorities.Contains(type))
            return (int)priorities[type];
        else
            return 0;
    }
    public int Compare(object o1, object o2)
    {
        int priority1 = PriorityFor(o1.GetType());
        int priority2 = PriorityFor(o2.GetType());
        return priority1.CompareTo(priority2);
    }
}

public void DrawAllShapes(ArrayList shapes)
{
    shapes.Sort(new ShapeComparer());
    foreach(Shape shape in shapes)
        shape.Draw();
}
```

•האם עכשיו הכל סגור?

# OCP סיכום

- עיקרון מרכזי בתיכון מערכת מונחית עצמים
- הטענה: עוזר להשיג גמישות, שמישות ותחזוקתיות
- ראינו שלא משיגים זאת רק ע"י שימוש בשפה מונחית עצמים
- כנ"ל גם לא ע"י החלת הפשטה ללא אבחנה
- אלא: במקומות שצפויים להשתנות!
- האם זה מספיק?



# LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



# Liskov Substitution Principle (LSP)

*Subclasses should be substitutable for their base classes (Barbara Liskov, 1988)*

- לקוח של מחלקת בסיס צריך להיות מסוגל לעבוד כרגיל גם אם יקבל במקום מחלקה נגזרת
- “A derived class should have some kind of specialized behavior (it should provide the same services as the superclass, only some, at least, are provided differently.)”
- “Let  $q(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $q(y)$  should be provable for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ ”...
- החוזה של מחלקת הבסיס חייב להיות מכובד ע"י מחלקה נגזרת
- הפרה של עיקרון זה מהווה גם הפרה של Open-Closed Principle – מדוע?

# LSP – Violation Example (C#)

```
struct Point {double x, y;}
public enum ShapeType {square, circle};
public class Shape
{
    private ShapeType type;
    public Shape(ShapeType t){type = t;}
    public static void DrawShape(Shape s)
    {
        if(s.type == ShapeType.square)
            (s as Square).Draw();
        else if(s.type == ShapeType.circle)
            (s as Circle).Draw();
    }
}
public class Circle : Shape
{ ...
```

•היכן ההפרה של LSP?  
•Circle ו-Square בעצם לא מחליפים את Shape  
•<= הפרה של LSP גורמת להפרה של OCP ב- DrawShape  
•מדוע תכננו זאת כך? (רמז: ביצועים)

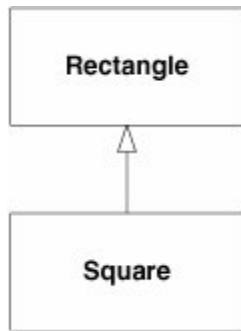
# LSP – A More Subtle Violation

```
public class Rectangle
{
    private Point topLeft;
    private double width;
    private double height;
    public double Width // non virtual, java: final
    {
        get { return width; }
        set { width = value; }
    }
    public double Height
    {
        get { return height; }
        set { height = value; }
    }
}
```

# LSP – A More Subtle Violation

- נניח שהמחלקה Rectangle משמשת באפליקציה כבר זמן רב

- דרישה חדשה: אפשרות להשתמש ב-Square כיצד נממש זאת?



- כידוע Square **IS-A** Rectangle
- האם משהו מריח לא טוב?
- Width ו-Height לא משתנים ביחד!
- פתרון אפשרי...

# LSP – A More Subtle Violation

```
public class Square : Rectangle
{
    public new double Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
    public new double Height
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

```
var s = new Square();
s.Width = 1;
s.Height = 2; // sets width too, hooray...
```

- האם הכל בסדר?
- מה עם הקוד הבא?

```
void f(Rectangle r)
{
    r.Width = 32;
}
```

- הפרה של LSP!
- האם אפשר לתקן?
- Virtual

# LSP – A More Subtle Violation

```
public class Rectangle
{
    ...
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }
}
public class Square : Rectangle
{
    public override double Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
    public override double Height
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

- עכשיו מתמטית ריבוע מתנהג בסדר
- מה עם הקוד הבא:

```
void g(Rectangle r)
{
    r.Width = 5;
    r.Height = 4;
    if (r.Area() != 20)
        throw new Exception("Bad area!");
}
```

- שוב הפרה של LSP!
- איפה טעינו? האם המפתח של g אשם?
- או אולי ההורשה לא מתאימה?
- לפי LSP מודל נכון תלוי בלקוחות שלו!

# LSP – A More Subtle Violation

- האם לא אמרנו ש: Square IS-A Rectangle?
- **ההתנהגות** של ריבוע לא קונסיסטנטית עם הציפיות של g!
- LSP מלמד אותנו שב-ODD היחס IS-A מתייחס להתנהגות סבירה שלקוחות מצפים
- איך יכולנו לחשוב על זה?

# Design By Contract

- שיטה מאת Bertrand Meyer, שפת Eiffel
- מפתח של מחלקה מודיע על חוזה שמכיל תנאי קדם, תנאי סיום ואינווריאנטות
- במקרה שלנו postcondition:
- `assert((width == w) && (height == old.height));`
- "A routine redeclaration [in a derivative] may only replace the original precondition by one **equal or weaker**, and the original postcondition by one **equal or stronger**."  
[Meyer97], p. 573



# Design By Contract

- ב- C# ו-Java אין תמיכה מובנית בשפה
- התפתחו ספריות שונות

iContract –

– spec# הרחבה מאת MSR – בדיקות סטטיות,  
שימשה לכתובת מ"ה Singularity (עם sing#)

– Guard (OSS)

- ספריית Contracts (MSR) + בדיקות אוטומטיות  
(Pex) - חלק מ-.Net 4.0, VS add-in

# .Net Contracts API

- `CodeContract.Requires(parameter >= 0);`
- `CodeContract.Ensures(SomeSharedState != null);`
- `CodeContract.EnsuresOnThrow<IOException>(SomeSharedState != null);`
- `CodeContract.Ensures(CodeContract.Result<Int32>() >= 0);`
- `[ContractInvariantMethod]`  
`void ObjectInvariant() {`  
    `CodeContract.Invariant(Data >= 0);`  
`}`
- [Pexforfun](#) example

# לסיכום LSP

- היורסטיקה: מחלקה יורשת שמורידה התנהגות, אולי מפירה את LSP

```
public class Base
{
    public virtual void f()
    { /*some code*/ }
}
...
public class Derived : Base
{
    public override void f() {}
}
```

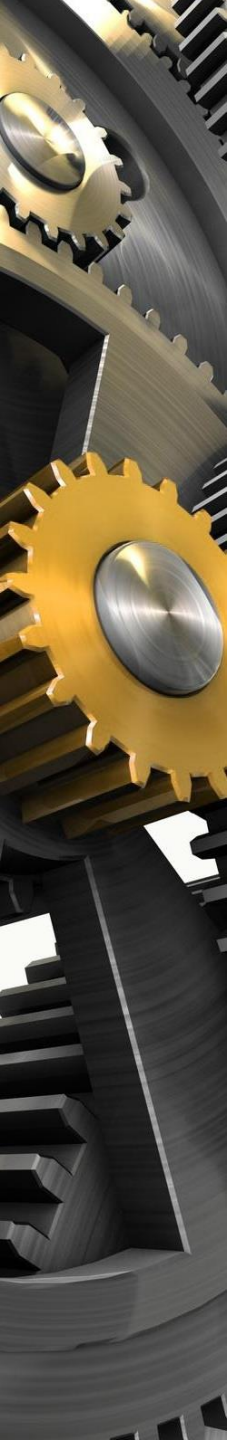
- עוד דוגמא: זריקת חריגה מטיפוס חדש
- שמירה על LSP עוזרת להגיע ל-OCP
- חוזים נותנים משמעות יותר ספציפית לקשרי מחלקות

# Plugin ב-play framework ישנו ממשק המאפשר הרחבה של ה-framework איזה עיקרון אינו מעורב ברעיון זה

1. Single Responsibility
2. Open Close
3. Liskov Substitution
4. Interface Segregation
5. Dependency Inversion ([pull req.](#))

# SOLID & Design Patterns

- חיים מכבי, 6/15, [Clean Code Alliance](#):  
[The SOLID Principles Illustrated by Design Patterns](#)
- (וגם הרצאה מיוחדת ל-SRP)  
<http://eyalgo.com/2015/06/11/srp-as-part-of-solid/>



# ***The SOLID Principles Illustrated by Design Patterns***

Hayim Makabee

<http://EffectiveSoftwareDesign.com>

# About Me:

Education:



Experience:



Current:



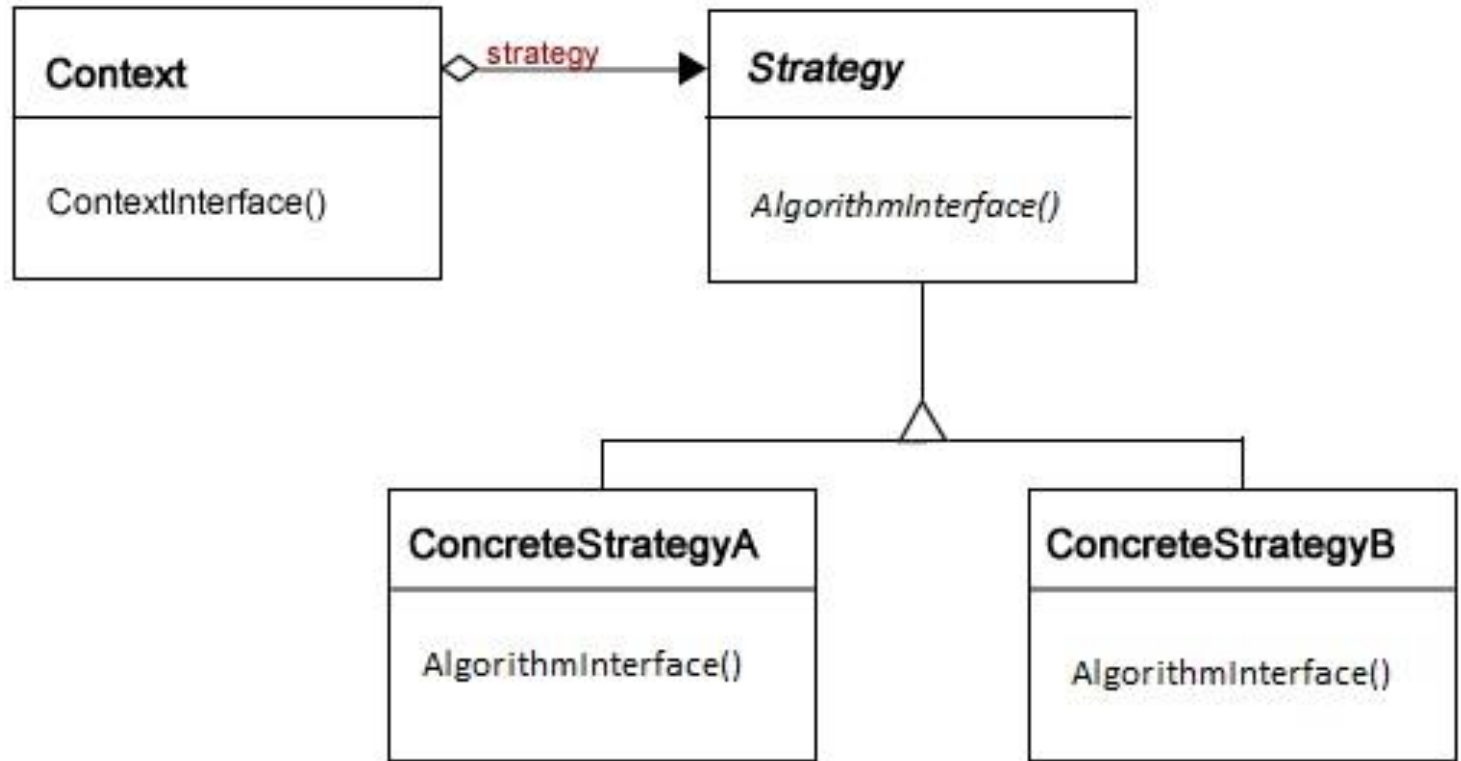


# The SOLID Principles

- **S**ingle Responsibility principle
- **O**pen/Closed principle
- **L**iskov Substitution principle
- **I**nterface Segregation principle
- **D**ependency Inversion principle



# Strategy Design Pattern

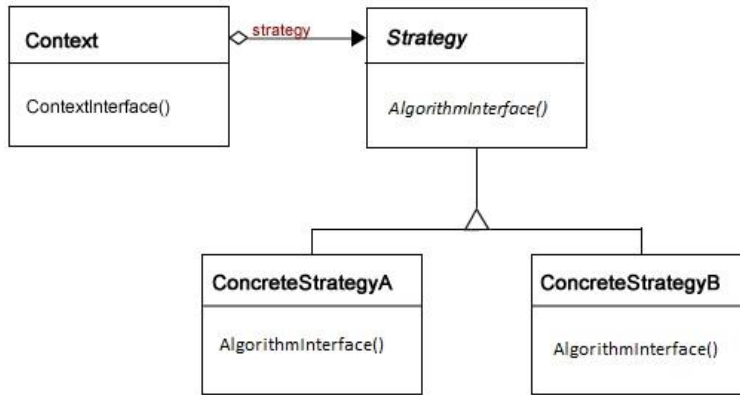




# Single Responsibility Principle

- Each class should have a single responsibility. Only one potential change in the system's specification should affect the implementation of the class.

# Single Responsibility @ Strategy



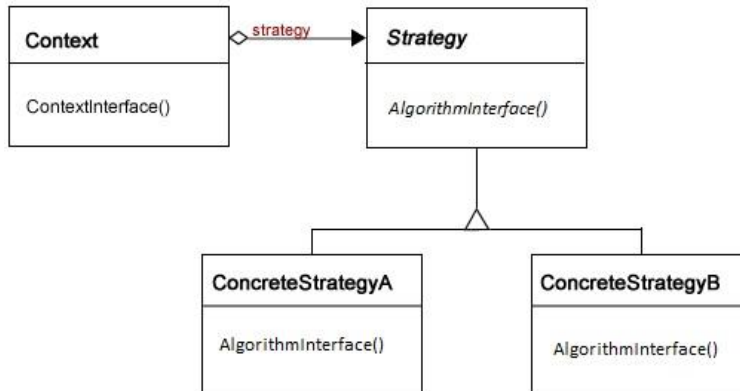
The responsibility for the implementation of a concrete strategy is decoupled from the context that uses this strategy.



# Open/Closed Principle

- “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” - Bertrand Meyer

# Open/Closed @ Strategy



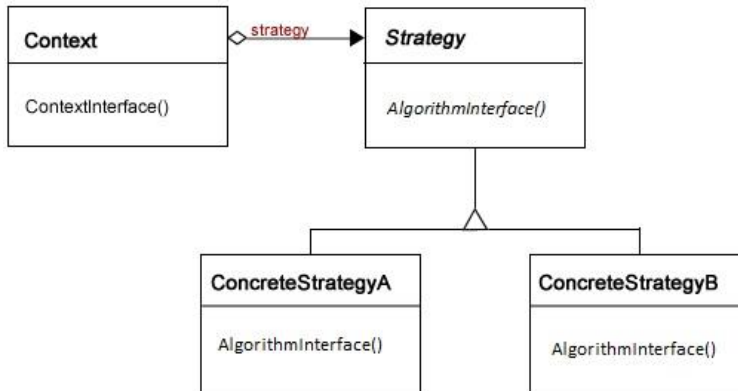
The context is open for extensions and closed for modifications since it does not need to be changed to use new types of strategies.



# Liskov Substitution Principle

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

# Liskov Substitution @ Strategy



All concrete strategies implement the same interface and should be substitutable without affecting the system's correctness.



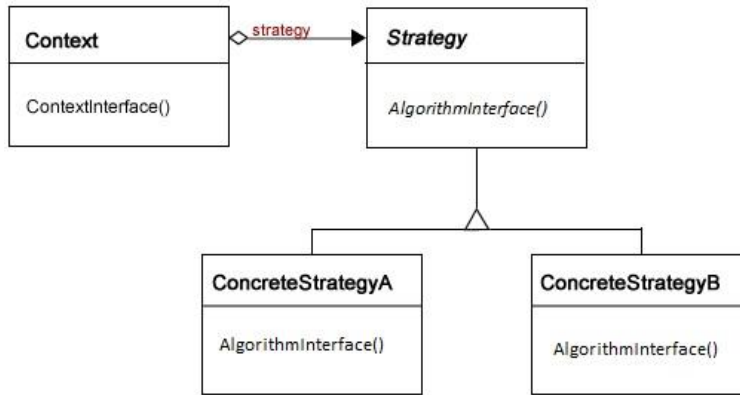


# Interface Segregation Principle

- No client should be forced to depend on methods it does not use. Many client-specific interfaces are better than one general-purpose interface.



# Interface Segregation @ Strategy



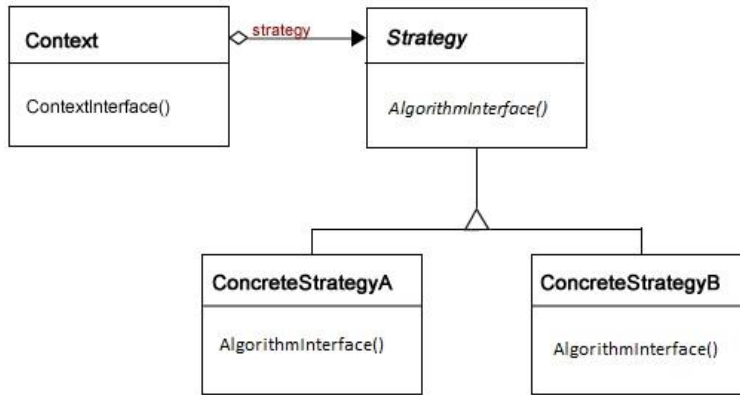
Concrete strategies implement an interface that provides only the specific needs of the context that uses it.



# Dependency Inversion Principle

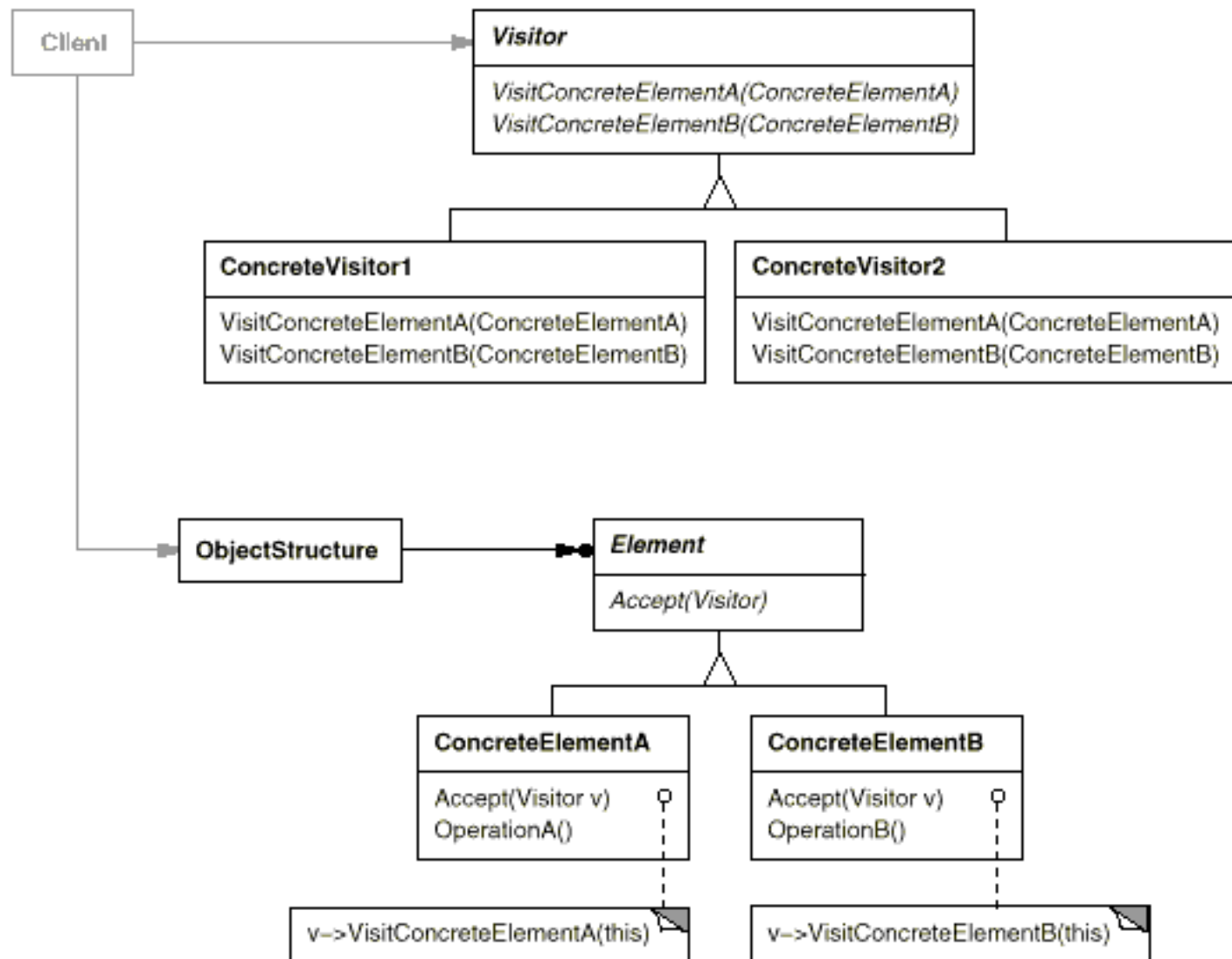
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

# Dependency Inversion @ Strategy

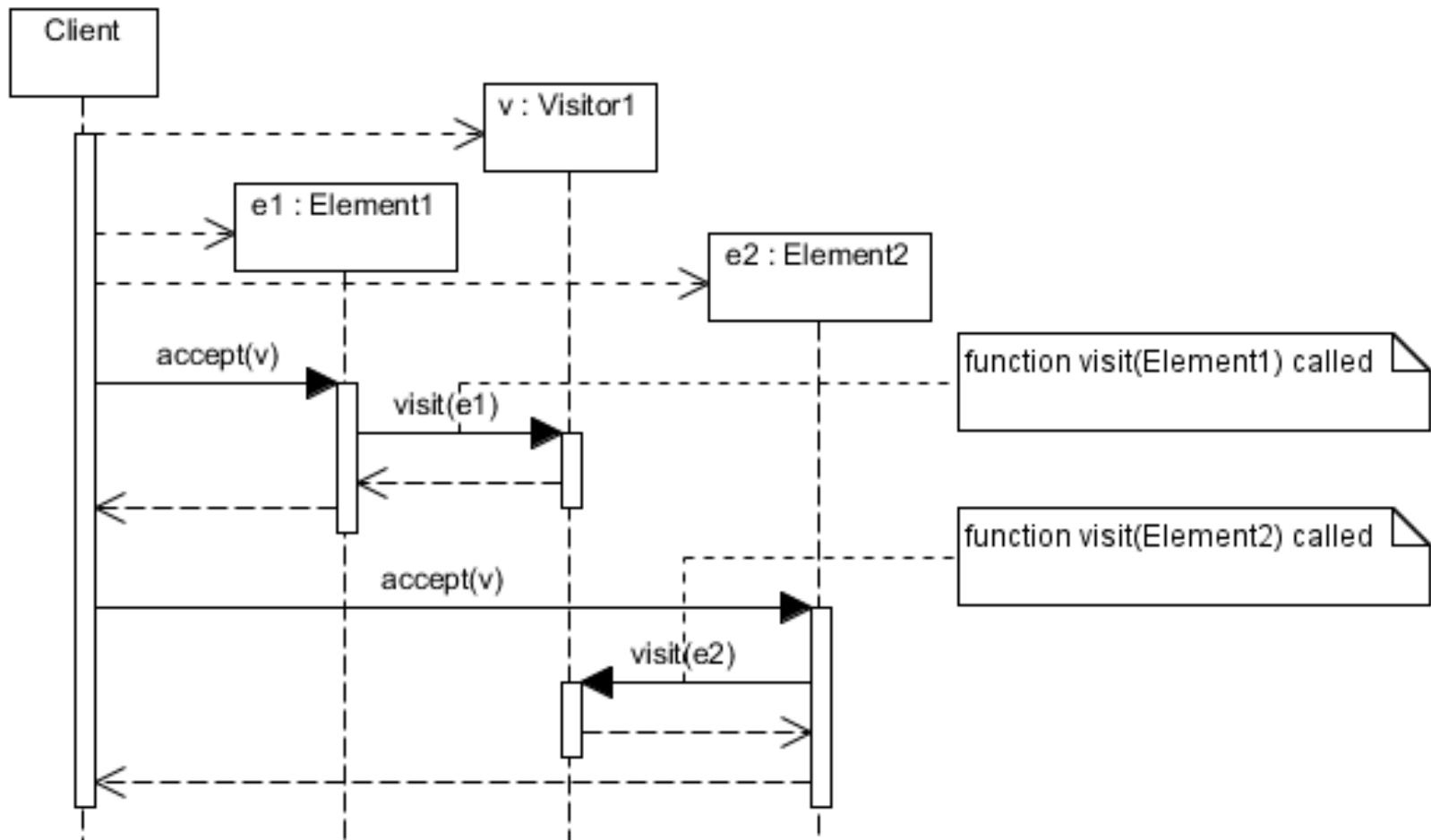


Both the context and the concrete strategies depend on an abstract interface and are not directly coupled.

# Visitor Design Pattern

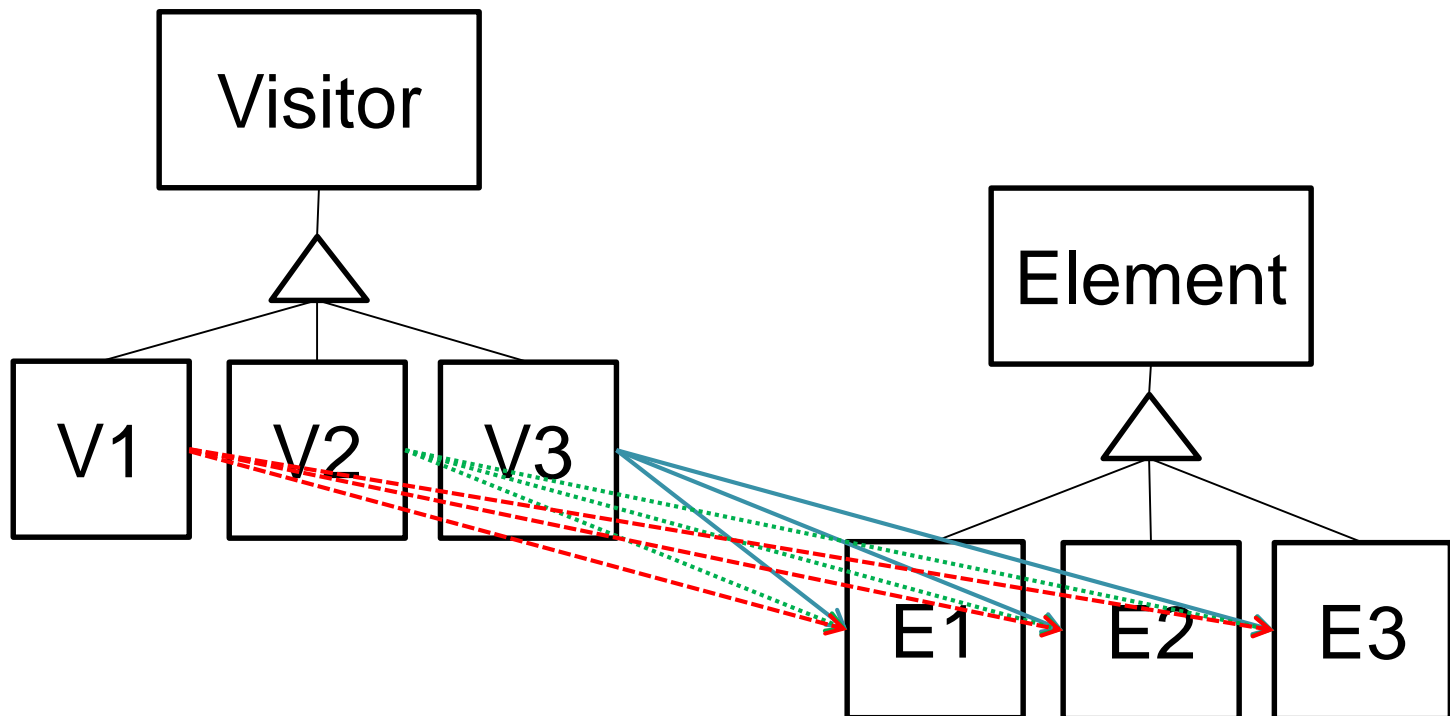


# Visitor: Sequence Diagram



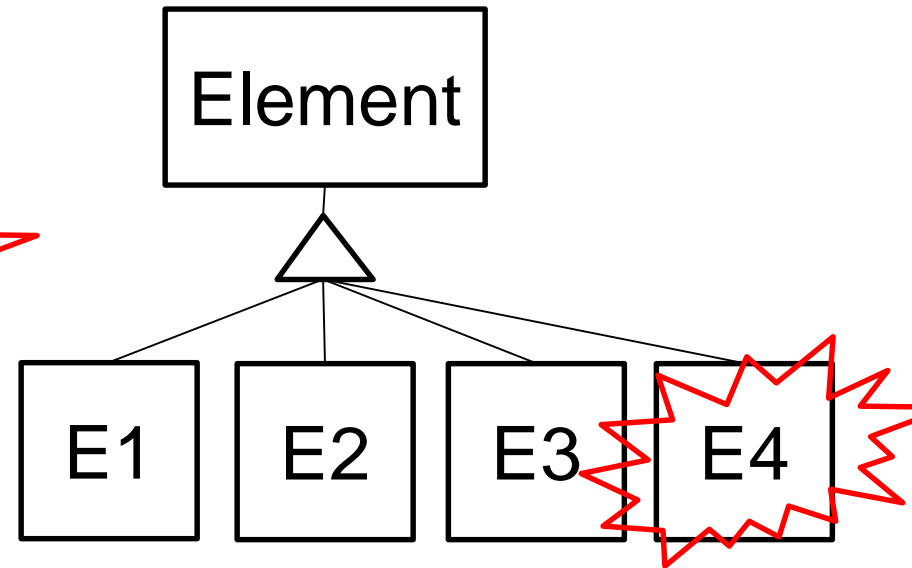
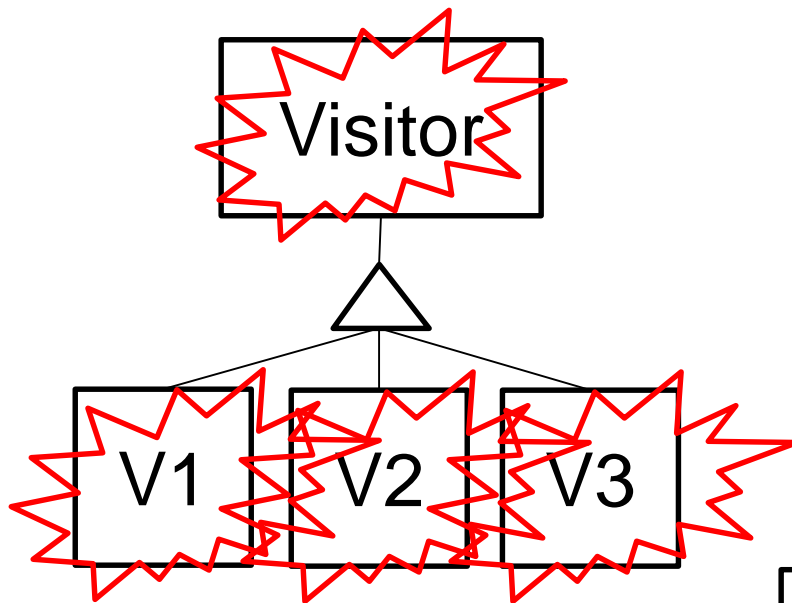
# Is Visitor SOLID?

- Visitor violates the Single-Responsibility principle. Consequence = weak cohesion.



# The Problem with Visitor

- When a new Element is added, all Visitors must be changed to add a new method.







# Single Responsibility

- Ideally: Dynamic binding by both type of Visitor and Element (Double Dispatch).

| Visitor X<br>Element | E1       | E2       | E3       |
|----------------------|----------|----------|----------|
| V1                   | m11(v,e) | m12(v,e) | m13(v,e) |
| V2                   | m21(v,e) | m22(v,e) | m23(v,e) |
| V3                   | m31(v,e) | m32(v,e) | m33(v,e) |





# Limitation of Single Dispatch

- In practice: Dynamic binding only by type of Visitor, the Element is not polymorphic.

| Visitor X<br>Element | E1        | E2        | E3        |
|----------------------|-----------|-----------|-----------|
| V1                   | v.m11(e1) | v.m12(e2) | v.m13(e3) |
| V2                   | v.m21(e1) | v.m22(e2) | v.m23(e3) |
| V3                   | v.m31(e1) | v.m32(e2) | v.m33(e3) |

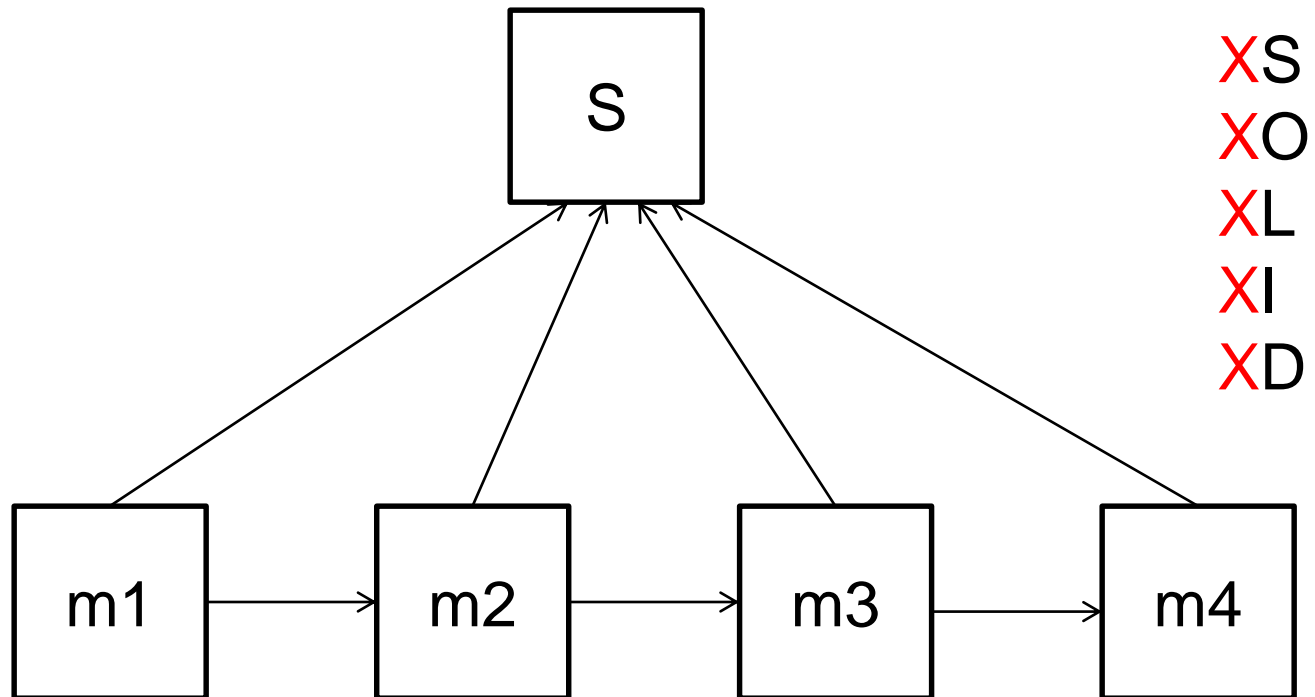


# Singleton Design Pattern

```
public class Singleton {  
    private static final Singleton  
        INSTANCE = new Singleton();  
    private Singleton() {}  
  
    public static Singleton  
        getInstance() {  
            return INSTANCE;  
        }  
}
```

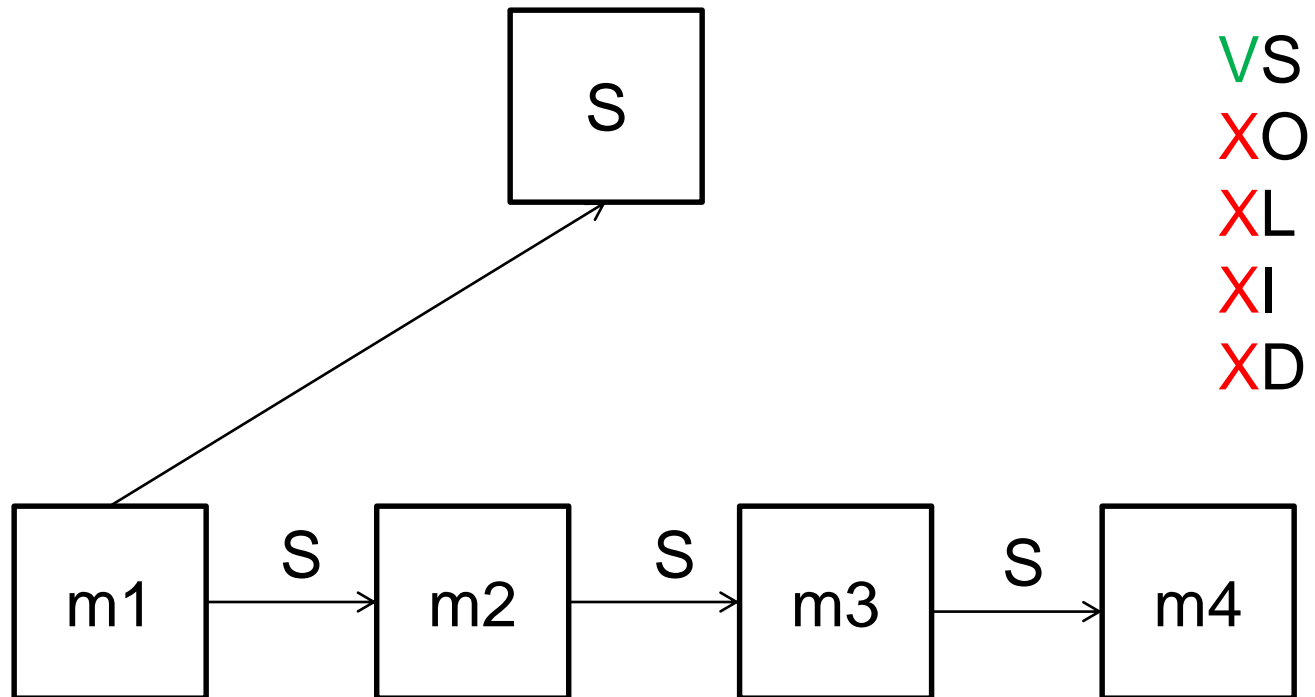
# The Problem with Singleton

- Singleton causes strong coupling:



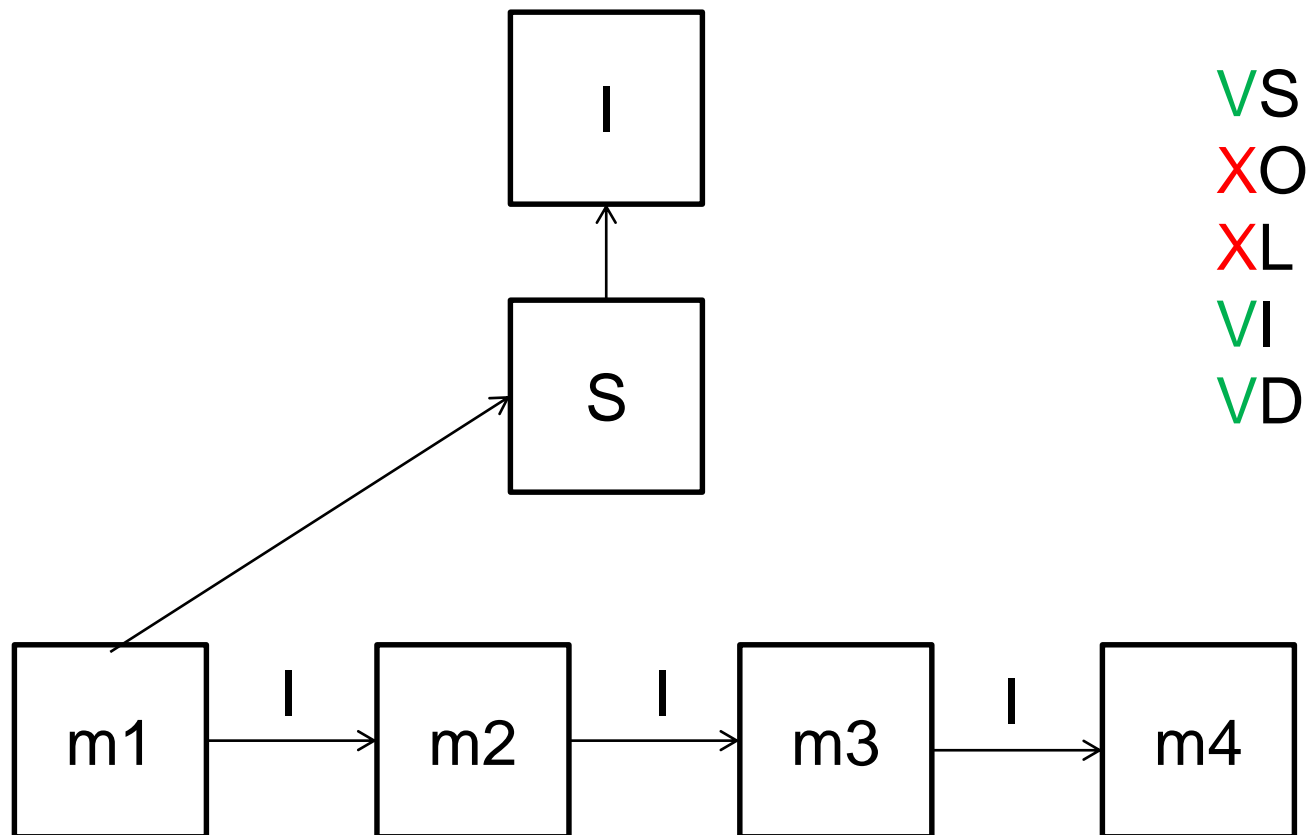
# Refactoring the Singleton – Step I

- Pass the Singleton object as a parameter:



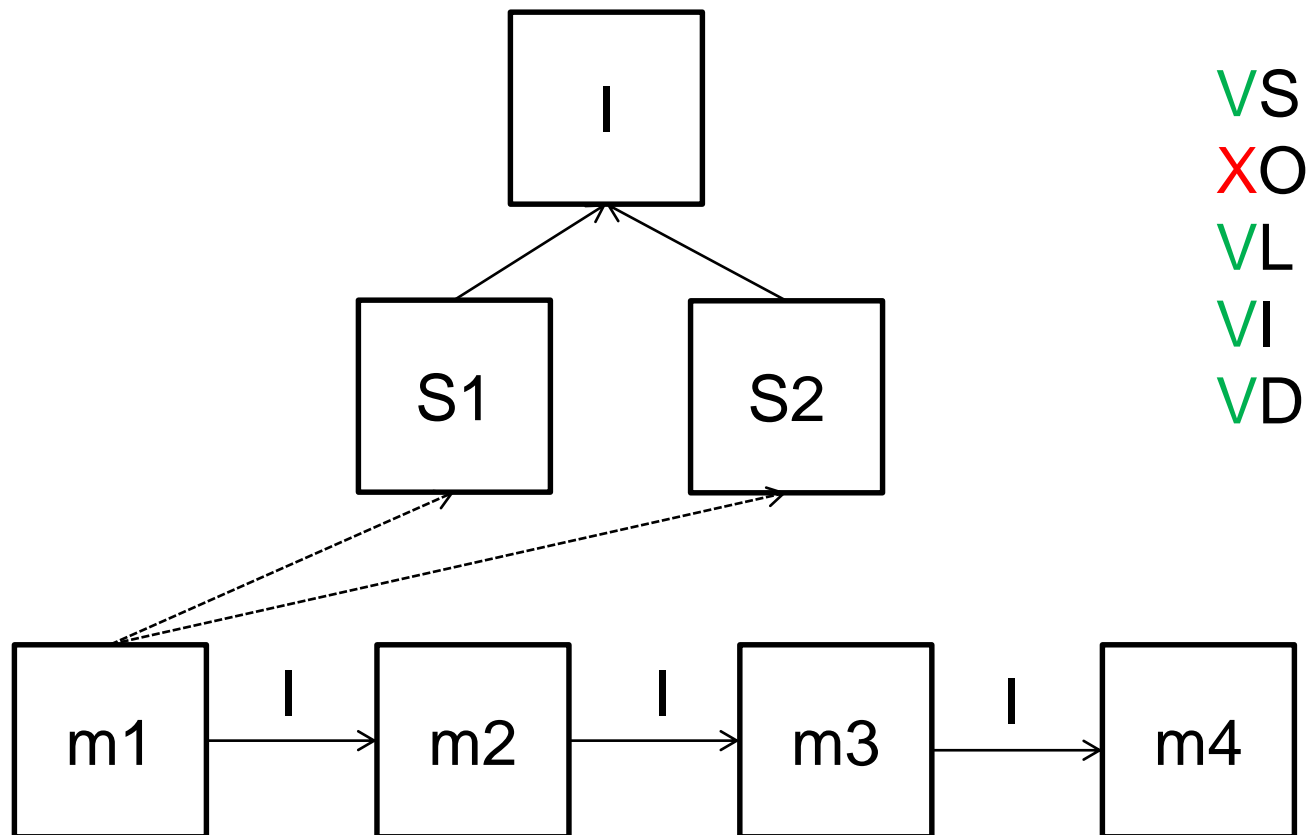
# Refactoring the Singleton – Step 2

- Derive the Singleton from an Interface:



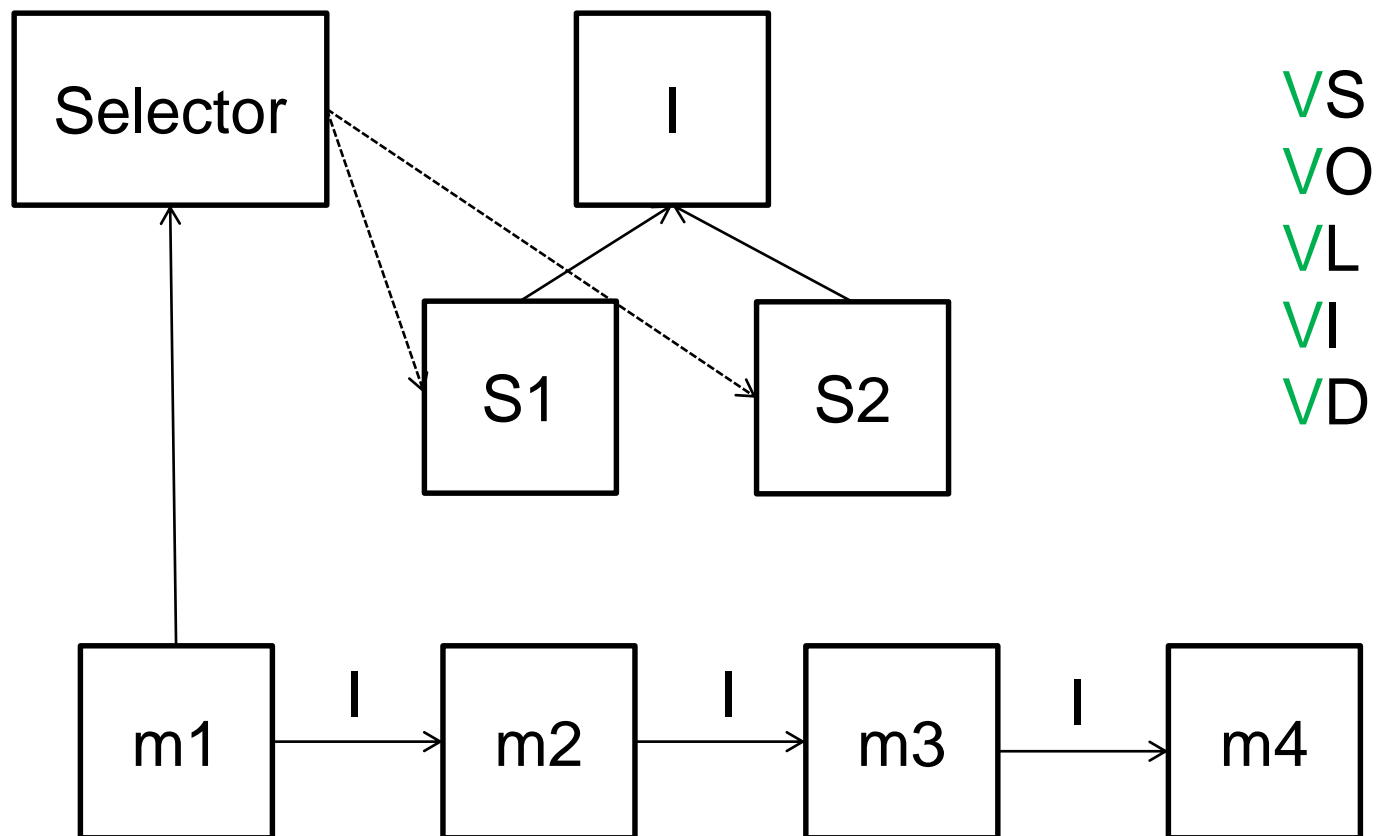
# Refactoring the Singleton – Step 3

- Several concrete Singleton classes:



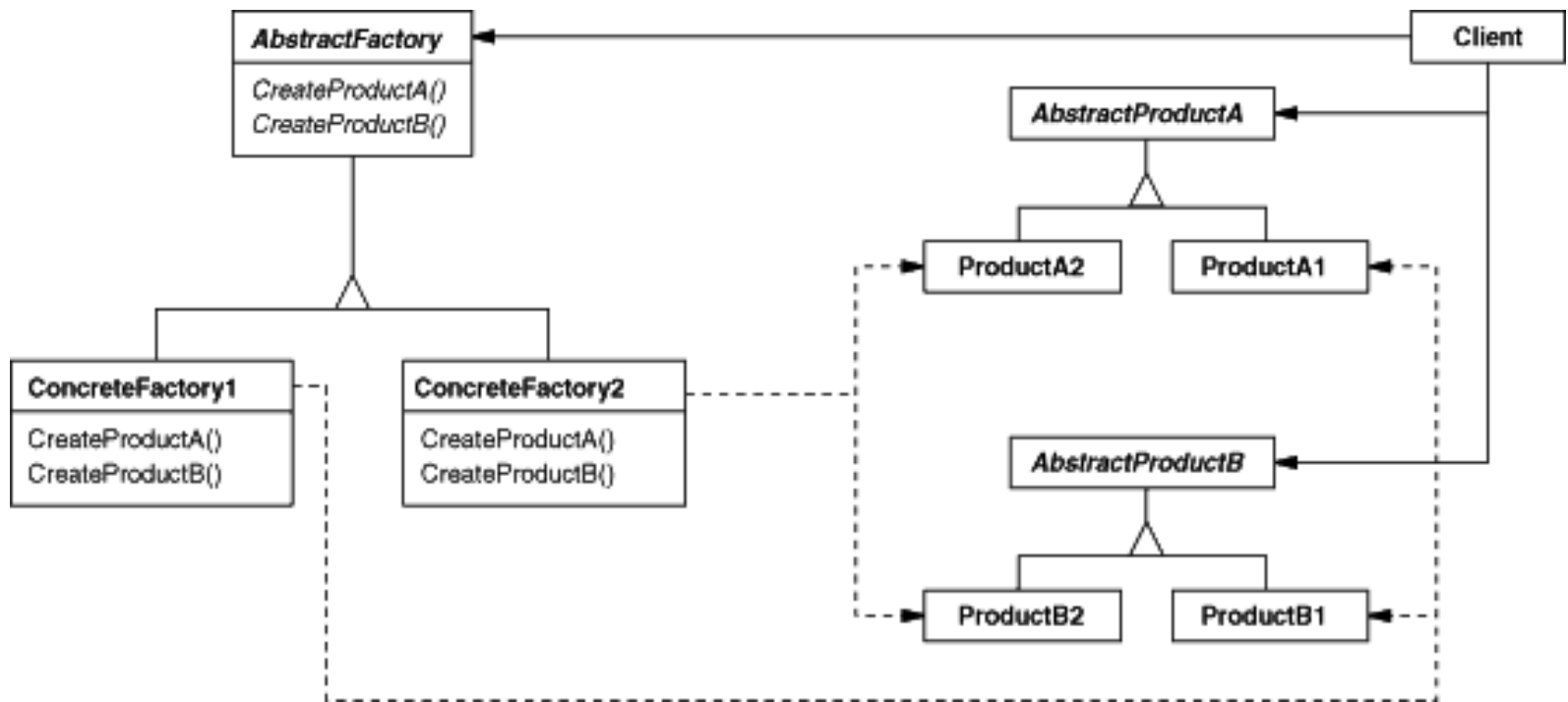
# Refactoring the Singleton – Step 4

- Parameter-based Singleton “selector”:





# Factory Design Pattern







# Pattern Metamorphosis

- Through the application of the SOLID principles, we moved from Singleton to the Factory Design Pattern.
- Singleton is actually a degenerate kind of Factory that always returns the same object of the same type...
- The interface of the Factory (getInstance) is mixed with the interface of the object itself...

A decorative image on the left side of the slide showing several interlocking gears. The gears are rendered in a 3D style with metallic textures. Some gears are yellow, while others are dark grey or black. They are arranged in a way that suggests a complex mechanical system.

# Conclusions

- To understand how to apply the SOLID principles in practice:
  1. Choose a Design Pattern.
  2. Analyze how this pattern makes use of the SOLID principles to reduce coupling and increase cohesion.
- Be aware that some patterns may violate some of the SOLID principles.

# בפעם הבאה

- חווית משתמש
- כלים ושיטות מתקדמים
- הכנה למצגות סיום
  - הצגה אישית
  - סקר

# לסיכום

- מתי לתקן או לשפר? מההתחלה? כשמגלים בעיה?
- לפעמים, מותר גם לתכנן מראש...
- כתיבת בדיקות מראש לגילוי הצרכים
- יש טוענים ש-TDD עם mocks ושות' גורם לעמידה ב-SOLID אפילו כשהמפתחים לא מכירים את העקרונות
- מתפתחים כלים ברמות שונות בנושא איכות הקוד
- נושא ללימוד מתמשך
- עקרונות נוספים במאמרים ובספרים של Martin ואחרים, למשל [The Law Of](#), [The Boy Scout Rule](#), [Tell Don't Ask](#), [Demeter](#) ויש מתנגדים, למשל:
- SOLID Fight: <http://www.artima.com/weblogs/viewpost.jsp?thread=250296>
- Recent SOLID Talks: [MS TechEd2014](#), [Weirich](#), [Clean-code-alliance](#)