תוכן

[**Adapter (מתאם)** 2](#_Toc96515358)

[**תיאור במשפט קצר** 2](#_Toc96515359)

[**הבעיה התכנותית** 2](#_Toc96515360)

[**דוגמא** 2](#_Toc96515361)

[**הפתרון** 3](#_Toc96515362)

[**תרשים UML** 3](#_Toc96515363)

[**דוגמת קוד** 3](#_Toc96515364)

[**דוגמאות נוספות לשימוש** 5](#_Toc96515365)

[**יתרונות** 5](#_Toc96515366)

[**חסרונות** 5](#_Toc96515367)

[**Bridge (גשר)** 5](#_Toc96515368)

[**תיאור במשפט קצר** 5](#_Toc96515369)

[**הבעיה התכנותית** 5](#_Toc96515370)

[**דוגמא** 5](#_Toc96515371)

[**הפתרון** 6](#_Toc96515372)

[**תרשים UML** 6](#_Toc96515373)

[**דוגמאות נוספות לשימוש** 8](#_Toc96515374)

[**יתרונות** 8](#_Toc96515375)

[**חסרונות** 8](#_Toc96515376)

[**Composite (הרכבה)** 8](#_Toc96515377)

[**תיאור במשפט קצר** 8](#_Toc96515378)

[**הבעיה התכנותית** 8](#_Toc96515379)

[**דוגמא** 8](#_Toc96515380)

[**הפתרון** 9](#_Toc96515381)

[**תרשים UML** 9](#_Toc96515382)

[**דוגמת קוד** 9](#_Toc96515383)

[**דוגמאות נוספות לשימוש** 12](#_Toc96515384)

[**יתרונות** 12](#_Toc96515385)

[**חסרונות** 12](#_Toc96515386)

[**Decorator (מקשט)** 12](#_Toc96515387)

[**תיאור במשפט קצר** 12](#_Toc96515388)

[**תבנית עיצוב** 12](#_Toc96515389)

[**הבעיה התכנותית** 12](#_Toc96515390)

[**דוגמא** 12](#_Toc96515391)

[**הפתרון** 12](#_Toc96515392)

[**תרשים UML** 13](#_Toc96515393)

[**דוגמת קוד** 13](#_Toc96515394)

[**דוגמאות נוספות לשימוש** 15](#_Toc96515395)

[**יתרונות** 15](#_Toc96515396)

[**חסרונות** 15](#_Toc96515397)

[**Facade (חזית, חלק חיצוני)** 15](#_Toc96515398)

[**תיאור במשפט קצר** 15](#_Toc96515399)

[**הבעיה התכנותית** 15](#_Toc96515400)

[**דוגמא** 15](#_Toc96515401)

[**הפתרון** 16](#_Toc96515402)

[**תרשים UML** 16](#_Toc96515403)

[**דוגמת קוד** 16](#_Toc96515404)

[**דוגמאות נוספות לשימוש:** 18](#_Toc96515405)

[**יתרונות** 18](#_Toc96515406)

[**חסרונות** 18](#_Toc96515407)

[**Flyweight (משקל זבוב)** 18](#_Toc96515408)

[**Proxy (נציג, מיופה כח)** 19](#_Toc96515409)

**Structural Design Patterns**

קבוצה זו של Design Pattern מתייחסת לפתרונות הקשורים במבנה האובייקטים בתוכנית.

בקבוצה זו נכללים:

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

**פירוט:**

# **Adapter (מתאם)**

## **תיאור במשפט קצר**

תבנית עיצוב שמאפשרת לאובייקטים מסוגים שונים לעבוד בשיתוף פעולה.

## **הבעיה התכנותית**

מחלקה X מבצעת קוד, שעובד עם אובייקט מסוג מסוים או עם מבנה מסוים, ונרצה להשתמש באותה מחלקה אבל עם אובייקט מסוג אחר. בדרך כלל מדובר במחלקה שאין לנו אפשרות לשנותה, או ששינוי שלה בעייתי כי ישבור קוד אחר שמשתמש בה.

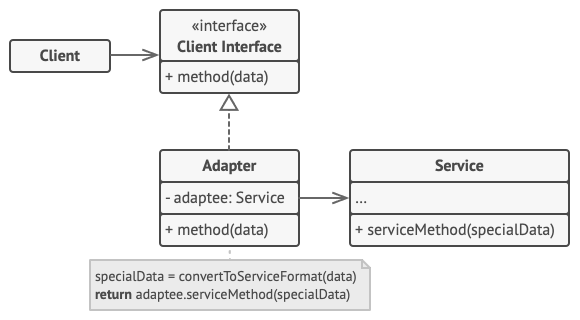
## **דוגמא**

קיימת תוכנה שמציגה נתוני מניות ממספר גורמים. הנתונים מתקבלים בפורמט של XML. כעבור תקופה, משדרגים את התוכנה כך שתציג סטטיסטיקות שונות. חישוב הסטטיסטיקות מתבצע באמצעות קוד של ספריה מוכנה שהורדנו מ npm. הספרייה מספקת את כל הפונקציונליות הדרושה לנו, אבל מקבלת נתונים בפורמט json ולא xml. גם אם הספרייה היא ספריית קוד פתוח, לא מעוניינים להשקיע זמן בשינוי הקוד של הספרייה.

## **הפתרון**

יצירת מתאם – אובייקט שלישי שיבצע את המעבר מפורמט XML לפורמט JSON. זימון הקוד של הספרייה יתבצע דרך המתאם. המתאם יעביר את הנתונים מ XML ל JSON ואז יזמן את הקוד של הספרייה. בקיצור: מתאם לשקע – מסוג אמריקאי לסוג אירופאי וכדו'.

## **תרשים UML**



## **דוגמת קוד**

**public** **interface** ILightningPhone

{

**void** ConnectLightning();

**void** Recharge();

}

**public** **interface** IUsbPhone

{

**void** ConnectUsb();

**void** Recharge();

}

**public** **sealed** **class** **AndroidPhone** : IUsbPhone

{

**private** bool isConnected;

**public** **void** ConnectUsb()

{

**this**.isConnected = **true**;

Console.WriteLine("Android phone connected.");

}

**public** **void** Recharge()

{

**if** (**this**.isConnected)

{

Console.WriteLine("Android phone recharging.");

}

**else**

{

Console.WriteLine("Connect the USB cable first.");

}

}

}

**public** **sealed** **class** **ApplePhone** : ILightningPhone

{

**private** bool isConnected;

**public** **void** ConnectLightning()

{

**this**.isConnected = **true**;

Console.WriteLine("Apple phone connected.");

}

**public** **void** Recharge()

{

**if** (**this**.isConnected)

{

Console.WriteLine("Apple phone recharging.");

}

**else**

{

Console.WriteLine("Connect the Lightning cable first.");

}

}

}

**public** **sealed** **class** **LightningToUsbAdapter** : IUsbPhone

{

**private** **readonly** ILightningPhone lightningPhone;

**private** bool isConnected;

**public** LightningToUsbAdapter(ILightningPhone lightningPhone)

{

**this**.lightningPhone = lightningPhone;

**this**.lightningPhone.ConnectLightning();

}

**public** **void** ConnectUsb()

{

**this**.isConnected = **true**;

Console.WriteLine("Adapter cable connected.");

}

**public** **void** Recharge()

{

**if** (**this**.isConnected)

{

**this**.lightningPhone.Recharge();

}

**else**

{

Console.WriteLine("Connect the USB cable first.");

}

}

}

**public** **void** Main()

{

ILightningPhone applePhone = **new** ApplePhone();

IUsbPhone adapterCable = **new** LightningToUsbAdapter(applePhone);

adapterCable.ConnectUsb();

adapterCable.Recharge();

}

פלט:

Apple phone connected.

Adapter cable connected.

Apple phone recharging.

## **דוגמאות נוספות לשימוש**

1. בורג עגול או מרובע – צריך מתאם בין העיגול למרובע.
2. אובייקט ציפור וברווז משחק – שניהם מצייצים, אבל כל אחד בצורה אחרת. בציפור יש פונקציה makeSound ולברווז משחק יש פונקציה squeak – ותכל'ס צריכים להפעיל קול, צריך מתאם שיפעיל את ה makeSound בציפור.([מימוש מלא של דוגמא זו](https://www.geeksforgeeks.org/adapter-pattern/))

## **יתרונות**

1. open\closed principle – אפשר להוסיף adapters נוספים בלי לשנות את הקיימים.
2. single responsibility principle – כל אחד מבצע משהו אחד. ה adapter – רק אחראי להמרה מסג אחד לסוג אחר, ה class שממנו מקבלים את השירות – מקבל את הנתון כמו שהוא מצפה לו, ללא תוספת עבודה של המרת הנתונים.

## **חסרונות**

1. מסבך קצת את הקוד – מתווספות מחלקות או ממשקים רק כדי לאפשר שימוש במחלקה אחרת. לפעמים פשוט יותר לשנות את המחלקה שבה רוצים להשתמש ולצורכה מבצעים את ההתאמה. לכן בדרך כלל נשתמש רק כשאין אפשרות לשנות את המחלקה.

# **Bridge (גשר)**

## **תיאור במשפט קצר**

הפרדה בין ההפשטה למימוש – הפרדה בין abstraction ל implementation

נשים לב: בהקשר של bridge לא מתייחס ל abstraction במשמעות של Interface או מחלקה אבסטרקטית, אלא למושג רעיוני.

## **הבעיה התכנותית**

קיימת מחלקת בסיס שמעונינים להרחיב אותה בשתי בחינות לפחות. בהורשה זה יתבטא בשתי מחלקות – אחת שמרחיבה מבחינה אחת ואחת שמרחיבה מבחינה אחרת.

## **דוגמא**

מחלקת צורה ומחלקות ספציפיות של צורות שיורשות ממנה, ריבוע ועיגול. בנוסף, קיימות גם מחלקות ריבוע מלא ועיגול מלא, שמבטאות צורה שיש לה מילוי (רקע). כאשר נרצה להוסיף משולש, נצטרך להוסיף מחלקה למשולש רגיל ומחלקה למשולש מלא, וכך עבור כל צורה שנוסיף. מהר מאד עץ המחלקות יגדל בצורה אקספוננציאלית.

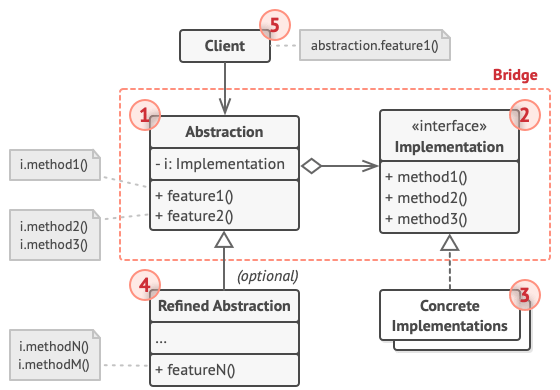
## **הפתרון**

שימוש ב composition (הרכבה) במקום ב Inheritance - הורשה.

מבצעים הפרדה בין שתי המהויות שבהן המחלקות מתרחבות. למעשה, זה אומר שעץ מחלקות אחד בדרך כלל יבטא את ההגדרה הרעיונית, והשני יבטא את המימוש בפועל.

1. ה abstraction מספק את ההגדרה של מה רוצים שיקרה (צורה)
2. ה implementation מספק את הפעילות עצמה – צורה ריקה, צורה מלאה וכו'
3. ב abstraction יהיה ייחוס לאובייקט מסוג ה Implementation, ודרכו תופעל הפונקציונליות בפועל.
4. ה client בדרך כלל יתעסק רק עם ה implementation – מה רוצה לעשות, ופחות – איך.

## **תרשים UML**



**דוגמת קוד**

**using** System;

**namespace** **RefactoringGuru**.DesignPatterns.Bridge.Conceptual

{

// The Abstraction defines the interface for the "control" part of the two

// class hierarchies. It maintains a reference to an object of the

// Implementation hierarchy and delegates all of the real work to this

// object.

**class** **Abstraction**

{

**protected** IImplementation \_implementation;

**public** Abstraction(IImplementation implementation)

{

**this**.\_implementation = implementation;

}

**public** **virtual** **string** Operation()

{

**return** "Abstract: Base operation with:\n" +

\_implementation.OperationImplementation();

}

}

// You can extend the Abstraction without changing the Implementation

// classes.

**class** **ExtendedAbstraction** : Abstraction

{

**public** ExtendedAbstraction(IImplementation implementation) : **base**(implementation)

{

}

**public** **override** **string** Operation()

{

**return** "ExtendedAbstraction: Extended operation with:\n" +

**base**.\_implementation.OperationImplementation();

}

}

// The Implementation defines the interface for all implementation classes.

// It doesn't have to match the Abstraction's interface. In fact, the two

// interfaces can be entirely different. Typically the Implementation

// interface provides only primitive operations, while the Abstraction

// defines higher- level operations based on those primitives.

**public** **interface** **IImplementation**

{

**string** OperationImplementation();

}

// Each Concrete Implementation corresponds to a specific platform and

// implements the Implementation interface using that platform's API.

**class** **ConcreteImplementationA** : IImplementation

{

**public** **string** OperationImplementation()

{

**return** "ConcreteImplementationA: The result in platform A.\n";

}

}

**class** **ConcreteImplementationB** : IImplementation

{

**public** **string** OperationImplementation()

{

**return** "ConcreteImplementationA: The result in platform B.\n";

}

}

**class** **Client**

{

// Except for the initialization phase, where an Abstraction object gets

// linked with a specific Implementation object, the client code should

// only depend on the Abstraction class. This way the client code can

// support any abstraction-implementation combination.

**public** **void** ClientCode(Abstraction abstraction)

{

Console.Write(abstraction.Operation());

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

Client client = **new** Client();

Abstraction abstraction;

// The client code should be able to work with any pre-configured

// abstraction-implementation combination.

abstraction = **new** Abstraction(**new** ConcreteImplementationA());

client.ClientCode(abstraction);

Console.WriteLine();

abstraction = **new** ExtendedAbstraction(**new** ConcreteImplementationB());

client.ClientCode(abstraction);

}

}

}

פלט:

Abstract: Base operation with:

ConcreteImplementationA: The result in platform A.

ExtendedAbstraction: Extended operation with:

ConcreteImplementationA: The result in platform B.

## **דוגמאות נוספות לשימוש**

1. GUI שאמור להיות מוצג בכמה מערכות הפעלה. יכול להתרחב בשתי צורות:

* ממשקי משתמש שונים – נניח מותאמות ללקוח רגיל או למנהל
* מערכות הפעלה שונות – windows, mac, Linux וכו'

אפשר לעשות מחלקות שבהן כל הזמן תהיה בדיקה איזו מערכת הפעלה זו או איזה משתמש ולפי זה מחליטים להציג\להסתיר חלקים מסוימים. אבל אם משתמשים ב bridge זה יהיה הרבה יותר מסודר.

החלק של ה abstraction – אחראי למה שרוצים להיות מוצג.

החלק של ה Implementation – אחראי לאופן בו הממשק מוצג בפועל במערכת הפעלה מסוימת.

## **יתרונות**

1. אפשר ליצור תוכנות שיכולות לעבוד על מערכות הפעלה שונות
2. ה client שצורך את הקוד לא חשוף לפרטי המימוש שלו, אלא רק לרמת ההגדרה של הקוד.
3. עקרון Open\Closed – אפשר להוסיף רמות שונות של הפשטה ומימוש ללא תלות זו בזו.
4. עקרון Single Responsibility – מחלקה אחת מגדירה מה צריך, מחלקות אחרות מסבצעות את המימוש לסוג מסוים – כל אחת סוג אחד.

## **חסרונות**

1. הקוד עלול להיות מסובך מדי עבור מחלקות מגובשות מאד.

# **Composite (הרכבה)**

## **תיאור במשפט קצר**

תבנית המאפשרת לך לחבר אובייקטים למבני עצים ולאחר מכן לעבוד עם מבנים אלה כאילו היו אובייקטים בודדים, גם אם הם עצמם מסתעפים.

## **הבעיה התכנותית**

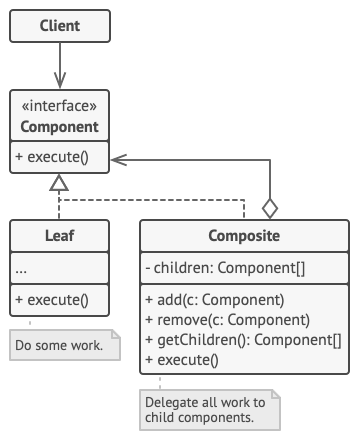
* יש לייצג היררכיה של חלק-שלם כך שלקוחות יוכלו להתייחס לאובייקטים חלקים ושלמים באופן אחיד.
* היררכיה חלקית-שלמה צריכה להיות מיוצגת כמבנה עץ.

## **דוגמא**

עובדים במפעל – ישנם כאלה שיש תחתיהם עובדים נוספים

## **הפתרון**

## **תרשים UML**



## **דוגמת קוד**

**using** System;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.Composite.Conceptual

{

// The base Component class declares common operations for both simple and

// complex objects of a composition.

**abstract** **class** **Component**

{

**public** Component() { }

// The base Component may implement some default behavior or leave it to

// concrete classes (by declaring the method containing the behavior as

// "abstract").

**public** **abstract** **string** Operation();

// In some cases, it would be beneficial to define the child-management

// operations right in the base Component class. This way, you won't

// need to expose any concrete component classes to the client code,

// even during the object tree assembly. The downside is that these

// methods will be empty for the leaf-level components.

**public** **virtual** **void** Add(Component component)

{

**throw** **new** NotImplementedException();

}

**public** **virtual** **void** Remove(Component component)

{

**throw** **new** NotImplementedException();

}

// You can provide a method that lets the client code figure out whether

// a component can bear children.

**public** **virtual** **bool** IsComposite()

{

**return** **true**;

}

}

// The Leaf class represents the end objects of a composition. A leaf can't

// have any children.

//

// Usually, it's the Leaf objects that do the actual work, whereas Composite

// objects only delegate to their sub-components.

**class** **Leaf** : Component

{

**public** **override** **string** Operation()

{

**return** "Leaf";

}

**public** **override** **bool** IsComposite()

{

**return** **false**;

}

}

// The Composite class represents the complex components that may have

// children. Usually, the Composite objects delegate the actual work to

// their children and then "sum-up" the result.

**class** **Composite** : Component

{

**protected** List<Component> \_children = **new** List<Component>();

**public** **override** **void** Add(Component component)

{

**this**.\_children.Add(component);

}

**public** **override** **void** Remove(Component component)

{

**this**.\_children.Remove(component);

}

// The Composite executes its primary logic in a particular way. It

// traverses recursively through all its children, collecting and

// summing their results. Since the composite's children pass these

// calls to their children and so forth, the whole object tree is

// traversed as a result.

**public** **override** **string** Operation()

{

**int** i = 0;

**string** result = "Branch(";

**foreach** (Component component **in** **this**.\_children)

{

result += component.Operation();

**if** (i != **this**.\_children.Count - 1)

{

result += "+";

}

i++;

}

**return** result + ")";

}

}

**class** **Client**

{

// The client code works with all of the components via the base

// interface.

**public** **void** ClientCode(Component leaf)

{

Console.WriteLine($"RESULT: {leaf.Operation()}\n");

}

// Thanks to the fact that the child-management operations are declared

// in the base Component class, the client code can work with any

// component, simple or complex, without depending on their concrete

// classes.

**public** **void** ClientCode2(Component component1, Component component2)

{

**if** (component1.IsComposite())

{

component1.Add(component2);

}

Console.WriteLine($"RESULT: {component1.Operation()}");

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

Client client = **new** Client();

// This way the client code can support the simple leaf

// components...

Leaf leaf = **new** Leaf();

Console.WriteLine("Client: I get a simple component:");

client.ClientCode(leaf);

// ...as well as the complex composites.

Composite tree = **new** Composite();

Composite branch1 = **new** Composite();

branch1.Add(**new** Leaf());

branch1.Add(**new** Leaf());

Composite branch2 = **new** Composite();

branch2.Add(**new** Leaf());

tree.Add(branch1);

tree.Add(branch2);

Console.WriteLine("Client: Now I've got a composite tree:");

client.ClientCode(tree);

Console.Write("Client: I don't need to check the components classes even when managing the tree:\n");

client.ClientCode2(tree, leaf);

}

}

}

**פלט**

Client: I get a simple component:

RESULT: Leaf

Client: Now I've got a composite tree:

RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:

RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)

## **דוגמאות נוספות לשימוש**

1. UI שמוצגים בו פקדים, יש פקדים שבתוכם קיימים עוד פקדים. רוצים לדעת לצייר את כולם, לא משנה אם יש פקדים נוספים בתוכם או אם אין.
2. תיקיות וקבצים

## **יתרונות**

1. יוצרים דרך אחידה להתייחס לאובייקטים שונים באמצעות פולימורפיזם ורקורסיה פנימית.
2. Open\Close principle – אפשר להוסיף אלמנטים נוספים להירארכיה בלי לשבור את הקוד הקיים.

## **חסרונות**

1. לעיתים קשה למצוא דרך להתייחס בצורה אחדיה לאובייקטים השונים שמרכיבים את ההירארכיה

# **Decorator (מקשט)**

## **תיאור במשפט קצר**

## **תבנית עיצוב**

## **הבעיה התכנותית**

קיים אובייקט כלשהו, שיכול לקבל תוספות. התוספות אינן משנות את האובייקט עצמו, אלא מוסיפות לו יכולות – משדרגות אותו.

ביצוע באמצעות הורשה מסורבל מאד. פירוט:

אופציה א – לא מומלצת:

ליצור מחלקות נפרדות לאובייקט עם כל אפשרויות השדרוגים שלו (פיצה, פיצה עם זיתים, פיצה עם פטריות, פיצה עם פטריות וזיתים, וכו' עם מרגריטה וזיתים, מרגריטה פטריות וכו')

ברור שהאופציה לא נוחה, המון מחלקות, ולא גמישה, לא נוחה לניהול

אופציה ב' –

להוסיף למחלקת הבסיס מאפיינים שיציינו את המידע הנוסף בפונקציה שצריכה להתייחס למידע הנוסף, להתייחס למאפיינים האלו ולקבוע את הפעולה שלה (למשל בדוגמא של הפיצה, יהיה hasMashrooms, hasOlive והפונקציה getCost בודקת אם הערך true אז היא מוסיפה במחיר

החסרון:

נוגד את Open-closed Principle כי:

1. אם מוסיפים תוספת חדשה – צריכים ל"פתוח" שוב את הקוד של מחלקת הבסיס
2. יכול להיות שיש מחלקות שיורשות מהבסיס והתוספת אינה רלוונטית עבורן
3. עדיין זה לא מספיק גמיש – כי רוצים כל מיני קומבינציות – זיתים כפול וכדו'

## **דוגמא**

פיצה עם תוספות

## **הפתרון**

נוסיף מחלקה חדשה שיורשת ממחלקת הבסיס, ממנה יורשות מחלקות ספציפיות כפי מספר התוספות.

למחלקה לא יהיה ctor ריק, וכל ה ctor שלה יקבלו מופע של מחלקת הבסיס. שממנה היא יורשת.

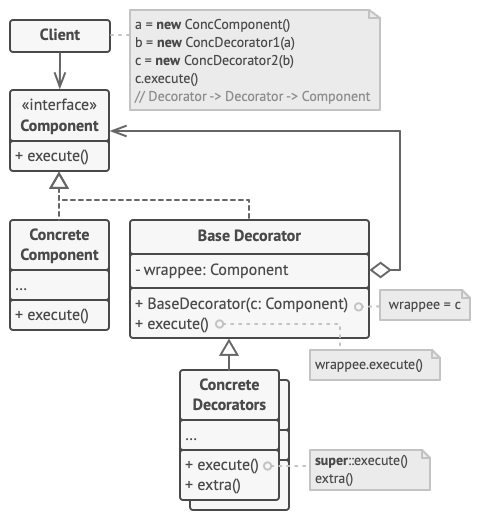
ב decorators הספציפיים |(זיתים, בצל וכדו'), נדרוס את הפונקציה הרלוונטית, נקרא לה Do שמושפעת מהתוספת (אם זה ציור של מלבן עם מילוי – הפונקציה שמושפעת היא פונקצית draw אם זה חישוב מחיר – הפונקציה המושפעת היא הפונקציה של getCost)

בפונקציה Do נקרא ל מופע שקבלנו ב ctor נקודה Do, + מה שאנחנו רוצים לבצע.

אחרי שמימשנו את הרצויים לנו, אפשר ב main ליצור מופעים של הרצויים לנו, ואפשר לקשט אותם – להשתמש ב decorator.

נדגיש, שאם רוצים לקשט מספר פעמים – תמיד נשלח ל decorator הבא – את המופע שחזר מה decorator הקודם. (הפיצה עם הבצל שנשלחת לזיתים – נשלח לזיתים את הפיצה עם הבצל ולא את הפיצה הבסיסית).

## **תרשים UML**



## **דוגמת קוד**

**namespace** **WikiDesignPatterns**;

**public** **interface** IBike

{

string GetDetails();

double GetPrice();

}

**public** **class** **AluminiumBike** : IBike

{

**public** double GetPrice() =>

100.0;

**public** string GetDetails() =>

"Aluminium Bike";

}

**public** **class** **CarbonBike** : IBike

{

**public** double GetPrice() =>

1000.0;

**public** string GetDetails() =>

"Carbon";

}

**public** **abstract** **class** **BikeAccessories** : IBike

{

**private** **readonly** IBike \_bike;

**public** BikeAccessories(IBike bike)

{

\_bike = bike;

}

**public** **virtual** double GetPrice() =>

\_bike.GetPrice();

**public** **virtual** string GetDetails() =>

\_bike.GetDetails();

}

**public** **class** **SecurityPackage** : BikeAccessories

{

**public** SecurityPackage(IBike bike):**base**(bike)

{

}

**public** **override** string GetDetails() =>

**base**.GetDetails() + " + Security Package";

**public** **override** double GetPrice() =>

**base**.GetPrice() + 1;

}

**public** **class** **SportPackage** : BikeAccessories

{

**public** SportPackage(IBike bike) : **base**(bike)

{

}

**public** **override** string GetDetails() =>

**base**.GetDetails() + " + Sport Package";

**public** **override** double GetPrice() =>

**base**.GetPrice() + 10;

}

**public** **class** **BikeShop**

{

**public** **static** **void** UpgradeBike()

{

var basicBike = **new** AluminiumBike();

BikeAccessories upgraded = **new** SportPackage(basicBike);

upgraded = **new** SecurityPackage(upgraded);

Console.WriteLine($"Bike: '{upgraded.GetDetails()}' Cost: {upgraded.GetPrice()}");

}

}

פלט:

Bike: 'Aluminium Bike + Sport Package + Security Package' Cost: 111

## **דוגמאות נוספות לשימוש**

1. לבישת בגדים – בחורף, אפשר ללבוש גם סוודר, גם ג'קט, גם צעיף וכו'. הכל לפי הצורך. לא חובה ללבוש סוודר כלל, ואפשר גם ללבוש שניים.
2. קבצים – יש אפשרות להצפין קובץ, יש אפשרות לכווץ קובץ, אפשר גם וגם.

## **יתרונות**

1. ניתן להרחיב מחלקה מבלי לרשת אותה.
2. ניתן להוסיף או להסיר אחריות לאובייקט בזמן ריצה.
3. ניתן לשלב מספר התנהגויות על אותו אובייקט, באמצעות שימוש במספר decorators.
4. Single Responsibility principle – חלוקה של מספר התנהגויות למחלקות נפרדות, כל אחת היא decorator נפרד.

## **חסרונות**

1. קשה לממש decorator כך שיהיה עצמאי לחלוטין ולא תלוי בסדר ה decorators.
2. הקוד שעוטף אובייקט ב decorator לא נראה כל כך יפה.

# **Facade (חזית, חלק חיצוני)**

## **תיאור במשפט קצר**

שער אחד למערכת מורכבת, ממשק אחיד לאוסף של ממשקים בתת-מערכת

## **הבעיה התכנותית**

קיימת מערכת מורכבת שמבצעת כמה פעולות ובכל שלב צריכים לדעת איזה פעולה לבצע, או לאן להפנות.

נניח מצרפים לקוד קבוצה רחבה של אובייקטים השייכים לספרייה או framework מתוחכמת. בדרך כלל, צריך לאתחל את כל האובייקטים האלה, לעקוב אחר תלויות ביניהם, לבצע פונקציות בסדר הנכון, וכן הלאה.

כתוצאה מכך, הלוגיקה העסקית של ה classים שלך תתחבר באופן הדוק לפרטי היישום של classים של צד שלישי, מה שמקשה על ההבנה והתחזוקה.

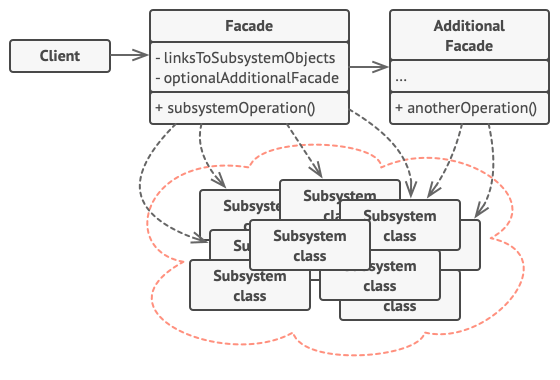
## **דוגמא**

שיחת טלפון לחברה שמספקת שירותים כלשהם. בדרך כלל יהיה נציג שירות לקוחות, שיפנה לגורם המתאים או ינחה מהי סדרת הפעולות שצריכה להתבצע כדי להשיג תוצאה מסוימת.

## **הפתרון**

ניצור מחלקה שתהיה הממשק החיצוני, והיא תדע לגשת לכל הפעולות של "המערכת" המורכבת שלנו.

## **תרשים UML**



## **דוגמת קוד**

**using** System;

**namespace** **RefactoringGuru**.DesignPatterns.Facade.Conceptual

{

// The Facade class provides a simple interface to the complex logic of one

// or several subsystems. The Facade delegates the client requests to the

// appropriate objects within the subsystem. The Facade is also responsible

// for managing their lifecycle. All of this shields the client from the

// undesired complexity of the subsystem.

**public** **class** **Facade**

{

**protected** Subsystem1 \_subsystem1;

**protected** Subsystem2 \_subsystem2;

**public** Facade(Subsystem1 subsystem1, Subsystem2 subsystem2)

{

**this**.\_subsystem1 = subsystem1;

**this**.\_subsystem2 = subsystem2;

}

// The Facade's methods are convenient shortcuts to the sophisticated

// functionality of the subsystems. However, clients get only to a

// fraction of a subsystem's capabilities.

**public** **string** Operation()

{

**string** result = "Facade initializes subsystems:\n";

result += **this**.\_subsystem1.operation1();

result += **this**.\_subsystem2.operation1();

result += "Facade orders subsystems to perform the action:\n";

result += **this**.\_subsystem1.operationN();

result += **this**.\_subsystem2.operationZ();

**return** result;

}

}

// The Subsystem can accept requests either from the facade or client

// directly. In any case, to the Subsystem, the Facade is yet another

// client, and it's not a part of the Subsystem.

**public** **class** **Subsystem1**

{

**public** **string** operation1()

{

**return** "Subsystem1: Ready!\n";

}

**public** **string** operationN()

{

**return** "Subsystem1: Go!\n";

}

}

// Some facades can work with multiple subsystems at the same time.

**public** **class** **Subsystem2**

{

**public** **string** operation1()

{

**return** "Subsystem2: Get ready!\n";

}

**public** **string** operationZ()

{

**return** "Subsystem2: Fire!\n";

}

}

**class** **Client**

{

// The client code works with complex subsystems through a simple

// interface provided by the Facade. When a facade manages the lifecycle

// of the subsystem, the client might not even know about the existence

// of the subsystem. This approach lets you keep the complexity under

// control.

**public** **static** **void** ClientCode(Facade facade)

{

Console.Write(facade.Operation());

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

// The client code may have some of the subsystem's objects already

// created. In this case, it might be worthwhile to initialize the

// Facade with these objects instead of letting the Facade create

// new instances.

Subsystem1 subsystem1 = **new** Subsystem1();

Subsystem2 subsystem2 = **new** Subsystem2();

Facade facade = **new** Facade(subsystem1, subsystem2);

Client.ClientCode(facade);

}

}

}

#### פלט:

Facade initializes subsystems:

Subsystem1: Ready!

Subsystem2: Get ready!

Facade orders subsystems to perform the action:

Subsystem1: Go!

Subsystem2: Fire!

## **דוגמאות נוספות לשימוש:**

1. הדלקה של מחשב – העלאה של CPU, אתחול הזכרון, וכו' – נבצע הכל ב facade אחד שיהיה הפונקציה של ה startup.המשתמש לא צריך לדעת אילו בידוק פעולות מבצעים בהדלקה של מחשב – הוא פשוט ידליק והכל יתבצע כראוי.
2. מחלקה שמקבלת קלט מחלון ה console ומעבירה אותו לתוך משתנה בתוכנית. אנחנו לא צריכים להתעסק עם החלקים של התממשקות עם התקן המקלדת, קליטת נתון מנו, זיהוי וכו'. זה מערכת ההפעלה תבצע. לנו רק יש את הפונקציה Console.ReadLine.
3. מלון עם מספר סוגי מסעדות, לכל מסעדה התפריט שלה – יהיה מישהו שמקבל את האורח בפתח, ומגיש לו את התפריט בו הוא מעונין. מה חסכנו לאורח? להסתובב בכל המסעדות, למצוא אותן, ולבקש מכל אחת את התפריט שלה.

## **יתרונות**

1. הפרדת הקוד מקוד מורכב של קוד צד שלישי

## **חסרונות**

1. במהלך הזמן facade עלול להפוך ל God object – אובייקט שמכיל פונקציות שער ליותר מדי תהליכים, שלאו דווקא קשורים ביניהם.

# **Flyweight (משקל זבוב)**

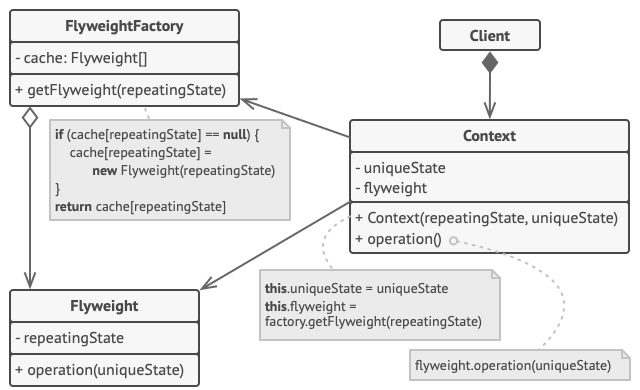
שימושי כאשר מתמודדים עם מספר רב של אובייקטים עם אלמנטים חוזרים ונשנים פשוטים שישתמשו בכמות גדולה של זיכרון אם יאוחסנו בנפרד. מקובל להחזיק נתונים משותפים במבני נתונים חיצוניים ולהעביר אותם לאובייקטים באופן זמני בעת השימוש בהם.

דוגמה קלאסית הם מבני הנתונים המשמשים המייצגים תווים במעבד תמלילים. באופן נאיבי, לכל תו במסמך עשוי להיות אובייקט המכיל את מתאר הגופן, מדדי הגופן ונתוני עיצוב אחרים. עם זאת, זה ישתמש במאות או אלפי בתים של זיכרון עבור כל תו. במקום זאת, לכל תו יכולה להיות הפניה לאובייקט המשותף לכל מופע של אותו תו במסמך. בדרך זו, רק המיקום של כל דמות צריך להיות מאוחסן פנימי.

כתוצאה מכך, עצמים בעלי משקל זבוב יכולים:

* לאחסן מצב מהותי שהוא בלתי משתנה, בלתי תלוי הקשר וניתן לשיתוף (לדוגמה, הקוד של התו 'A' בקבוצת תווים נתונה)
* לספק ממשק למעבר במצב חיצוני שהוא וריאנט, תלוי הקשר ואינו יכול להיות משותף (לדוגמה, המיקום של התו 'A' במסמך טקסט)

clients יכולים לעשות שימוש חוזר בחפצי Flyweight ולעבור במצב חיצוני לפי הצורך, ולהפחית את מספר האובייקטים שנוצרו פיזית.



# **Proxy (נציג, מיופה כח)**

מיופה כח, נמצא "לפני" השירות שאליו רוצים להגיע ומחליט מה לבצע. לפעמים יעביר הלאה את הבקשה למישהו נוסף ולפעמים יטפל בעצמו. – ולכן הוא נקרא מיופה כח, כי הוא יכול לבצע פעולה של מישהו אחר, כביכול.

דוגמא:

ניצור InternetProxy שיש לו מאפיין שמכיל רשימה של אתרים שאנו רוצים לחסום.

ניצור Internet שהוא כביכול בעצמו האינטרנט

שניהם יממשו interface, שיש לו פונקציה Search שמקבלת url (כמחרוזת) ומחזירה מחרוזת

משתמש רוצה לבצע חיפוש בפונקציה Search, הוא לא יפנה ישירות לאינטרנט, אלא יעבור דרך ה proxy.

ה proxy יבצע בדיקה:

* אם האתר לא נמצא ברשימת החסומים – יעביר הלאה את הבקשה לאינטרנט שיחזיר גלשת לאתר XXX
* אם נמצא באתרים החסומים – יחזיר הודעה access denied

