Structural Design Patterns

תוכן

[**Chain of Responsibility (שרשרת = chain)** 4](#_Toc96207952)

[**תיאור במשפט קצר** 4](#_Toc96207953)

[**הבעיה התכנותית** 4](#_Toc96207954)

[**דוגמא** 4](#_Toc96207955)

[**הפתרון** 4](#_Toc96207956)

[**UML** 5](#_Toc96207957)

[**דוגמת קוד** 5](#_Toc96207958)

[**דוגמאות נוספות** 8](#_Toc96207959)

[**יתרונות** 8](#_Toc96207960)

[**חסרונות** 8](#_Toc96207961)

[**Command (פקודה)** 8](#_Toc96207962)

[**תיאור במשפט קצר** 8](#_Toc96207963)

[**הבעיה התכנותית** 8](#_Toc96207964)

[**דוגמא** 8](#_Toc96207965)

[**הפתרון** 8](#_Toc96207966)

[**UML** 9](#_Toc96207967)

[**דוגמת קוד** 9](#_Toc96207968)

[**דוגמאות נוספות** 11](#_Toc96207969)

[**יתרונות** 11](#_Toc96207970)

[**חסרונות** 11](#_Toc96207971)

[**Iterator (אובייקט למעבר על איברי קבוצה)** 11](#_Toc96207972)

[**תיאור במשפט קצר** 11](#_Toc96207973)

[**הבעיה התכנותית** 11](#_Toc96207974)

[**דוגמא** 12](#_Toc96207975)

[**הפתרון** 12](#_Toc96207976)

[**UML** 13](#_Toc96207977)

[**דוגמת קוד** 13](#_Toc96207978)

[**דוגמאות נוספות** 15](#_Toc96207979)

[**יתרונות** 16](#_Toc96207980)

[**חסרונות** 16](#_Toc96207981)

[**Mediator (מתווך)** 16](#_Toc96207982)

[**תיאור במשפט קצר** 16](#_Toc96207983)

[**הבעיה התכנותית** 16](#_Toc96207984)

[**דוגמא** 16](#_Toc96207985)

[**הפתרון** 16](#_Toc96207986)

[**UML** 17](#_Toc96207987)

[**דוגמת קוד** 17](#_Toc96207988)

[**דוגמאות נוספות** 19](#_Toc96207989)

[**יתרונות** 19](#_Toc96207990)

[**חסרונות** 20](#_Toc96207991)

[**Memento (מזכרת)** 20](#_Toc96207992)

[**תיאור במשפט קצר** 20](#_Toc96207993)

[**הבעיה התכנותית** 20](#_Toc96207994)

[**דוגמא** 20](#_Toc96207995)

[**הפתרון** 20](#_Toc96207996)

[**UML** 21](#_Toc96207997)

[**דוגמת קוד** 21](#_Toc96207998)

[**דוגמאות נוספות** 24](#_Toc96207999)

[**יתרונות** 24](#_Toc96208000)

[**חסרונות** 24](#_Toc96208001)

[**Observer (משקיף)** 24](#_Toc96208002)

[**תיאור במשפט קצר** 24](#_Toc96208003)

[**הבעיה התכנותית** 25](#_Toc96208004)

[**דוגמא** 25](#_Toc96208005)

[**הפתרון** 25](#_Toc96208006)

[**UML** 25](#_Toc96208007)

[**דוגמת קוד** 25](#_Toc96208008)

[**דוגמאות נוספות** 28](#_Toc96208009)

[**יתרונות** 28](#_Toc96208010)

[**חסרונות** 28](#_Toc96208011)

[**State (מצב)** 28](#_Toc96208012)

[**תיאור במשפט קצר** 28](#_Toc96208013)

[**הבעיה התכנותית** 28](#_Toc96208014)

[**דוגמא** 28](#_Toc96208015)

[**הפתרון** 28](#_Toc96208016)

[**UML** 29](#_Toc96208017)

[**דוגמת קוד** 29](#_Toc96208018)

[**דוגמאות נוספות** 31](#_Toc96208019)

[**יתרונות** 31](#_Toc96208020)

[**חסרונות** 31](#_Toc96208021)

[**Strategy (אסטרטגיה, תכנית פעולה)** 31](#_Toc96208022)

[**תיאור במשפט קצר** 31](#_Toc96208023)

[**הבעיה התכנותית** 31](#_Toc96208024)

[**דוגמא** 31](#_Toc96208025)

[**הפתרון** 31](#_Toc96208026)

[**UML** 32](#_Toc96208027)

[**דוגמת קוד** 32](#_Toc96208028)

[**דוגמאות נוספות** 34](#_Toc96208029)

[**יתרונות** 34](#_Toc96208030)

[**חסרונות** 34](#_Toc96208031)

Behavioral Design Patterns

קבוצה זו עוסקת ב Design Patterns הקשורים באלגוריתמים, התקשורת ביניהם, וחלוקת האחריות בין הרכיבים השונים של האלגוריתם. תבניות עיצוב אלו מסייעות לפרק אלגוריתמים מורכבים לקוד שקל יותר לנהל, באמצעות חלוקת אחריות נכונה בין האובייקטים ותקשורת מוגדרת ביניהם.

ה Patterns שמופיעים בספר של GoF:

1. Chain of Responsibility
2. Command
3. Iterator
4. Mediator
5. Memento
6. Observer
7. State
8. Strategy
9. Template Method
10. Visitor

# **Chain of Responsibility (שרשרת סמכויות)**

## **תיאור במשפט קצר**

Design Pattern שמאפשר להעביר בקשה לאורך רצף של מטפלים. עם קבלת בקשה, כל מטפל מחליט אם לטפל בבקשה או להעביר אותה למטפל הבא ברצף.

## **הבעיה התכנותית**

ישנה משימה מסוימת לטיפול. הטיפול יכול להתבצע ע"י גורמים שונים (שיש קשר לוגי ביניהם), כאשר לא כולם מורשים או מסוגלים לטפל בכל משימה. פרמטרים מסוימים במשימה יקבעו מי יהיה הגורם המטפל.

## **דוגמא**

בקשה לאישור הוצאה תקציבית במפעל. קיימת הירארכיה של עובדים כך שבקשות עד לסכום של 100 ₪ מאושרות ע"י המזכירה, מ 100 עד 1000 – מאושרות ע"י המנהל, מ 10000 עד 50000 וכו'.

מעוניינים שהגשת הבקשות לאישור יעבור תמיד דרך המזכירה, מה שהיא יכולה לאשר – תאשר, ומה שלא – תעביר הלאה.

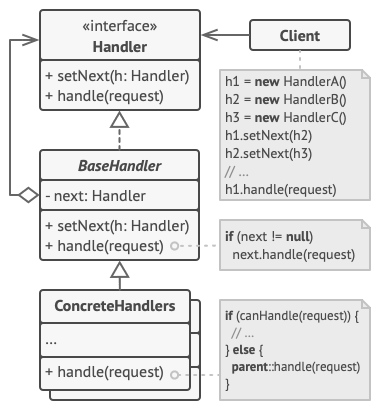
חשוב: המזכירה לא יודעת למי בדיוק יש סמכות לאשר איזה בקשה, היא רק יודעת מה היא יכולה לטפל ומטפלת כשהיא יכולה.

## **הפתרון**

מגדירים מחלקת בסיס ומחלקות יורשות מסוגי ה Handlers (המטפלים בבקשה) השונים. בכל handler יש הצבעה ל handler הבא בתור – אליו תועבר הבקשה במקרה ולא הסתיים הטיפול ב Handler הנוכחי. כל Handler יכול לסיים טיפול בבקשה, להעביר אותה הלאה לטיפול, לטפל בצורה מסוימת ולהעביר להמשך טיפול – אופי הטיפול תלוי במקרה הספציפי.

הגדרת ה Handlers השונים והרצף שלהם תתבצע בדרך כלל במקום מרכזי אחד בתוכנה, יכולה להתבצע בצורה סטטית או דינאמית.

## **UML**



## **דוגמת קוד**

**using** System;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.ChainOfResponsibility.Conceptual

{

// The Handler interface declares a method for building the chain of

// handlers. It also declares a method for executing a request.

**public** **interface** **IHandler**

{

IHandler SetNext(IHandler handler);

**object** Handle(**object** request);

}

// The default chaining behavior can be implemented inside a base handler

// class.

**abstract** **class** **AbstractHandler** : IHandler

{

**private** IHandler \_nextHandler;

**public** IHandler SetNext(IHandler handler)

{

**this**.\_nextHandler = handler;

// Returning a handler from here will let us link handlers in a

// convenient way like this:

// monkey.SetNext(squirrel).SetNext(dog);

**return** handler;

}

**public** **virtual** **object** Handle(**object** request)

{

**if** (**this**.\_nextHandler != **null**)

{

**return** **this**.\_nextHandler.Handle(request);

}

**else**

{

**return** **null**;

}

}

}

**class** **MonkeyHandler** : AbstractHandler

{

**public** **override** **object** Handle(**object** request)

{

**if** ((request **as** **string**) == "Banana")

{

**return** $"Monkey: I'll eat the {request.ToString()}.\n";

}

**else**

{

**return** **base**.Handle(request);

}

}

}

**class** **SquirrelHandler** : AbstractHandler

{

**public** **override** **object** Handle(**object** request)

{

**if** (request.ToString() == "Nut")

{

**return** $"Squirrel: I'll eat the {request.ToString()}.\n";

}

**else**

{

**return** **base**.Handle(request);

}

}

}

**class** **DogHandler** : AbstractHandler

{

**public** **override** **object** Handle(**object** request)

{

**if** (request.ToString() == "MeatBall")

{

**return** $"Dog: I'll eat the {request.ToString()}.\n";

}

**else**

{

**return** **base**.Handle(request);

}

}

}

**class** **Client**

{

// The client code is usually suited to work with a single handler. In

// most cases, it is not even aware that the handler is part of a chain.

**public** **static** **void** ClientCode(AbstractHandler handler)

{

**foreach** (**var** **food** **in** **new** List<**string**> { "Nut", "Banana", "Cup of coffee" })

{

Console.WriteLine($"Client: Who wants a {food}?");

**var** **result** = handler.Handle(food);

**if** (result != **null**)

{

Console.Write($" {result}");

}

**else**

{

Console.WriteLine($" {food} was left untouched.");

}

}

}

}

**class** **Program**

{

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

{

// The other part of the client code constructs the actual chain.

**var** **monkey** = **new** MonkeyHandler();

**var** **squirrel** = **new** SquirrelHandler();

**var** **dog** = **new** DogHandler();

monkey.SetNext(squirrel).SetNext(dog);

// The client should be able to send a request to any handler, not

// just the first one in the chain.

Console.WriteLine("Chain: Monkey > Squirrel > Dog\n");

Client.ClientCode(monkey);

Console.WriteLine();

Console.WriteLine("Subchain: Squirrel > Dog\n");

Client.ClientCode(squirrel);

}

}

}

פלט של הקוד לעיל

Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?

Squirrel: I'll eat the Nut.

Client: Who wants a Banana?

Monkey: I'll eat the Banana.

Client: Who wants a Cup of coffee?

Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?

Squirrel: I'll eat the Nut.

Client: Who wants a Banana?

Banana was left untouched.

Client: Who wants a Cup of coffee?

Cup of coffee was left untouched.

## **דוגמאות נוספות**

1. request שמגיעה לשרת. בדרך כלל יתבצע תהליך של authentication (בדיקת אבטחה), authorization (בדיקת הרשאות גישה לנתונים- מנהל, משתמש רגיל, עורך תוכן וכו'), ביצוע הבקשה, בדיקה אם כבר הייתה בקשה דומה לבקשה זו כך שלא צריך לעבד מחדש את התשובה, וכו'. כל שלב בעיבוד של הבקשה הוא חוליה בשרשרת של המטפלים בבקשה.

בדוגמא זו יתכן מצב שבו ישנם מספר מטפלים שיפעלו (גם בדיקת אבטחה, גם הרשאה וכו'), בשונה מהדוגמא של אישור הוצאת תקציבית.

1. מרכז תמיכה טכנית – שלב ראשון, במענה הקולי מתארים בעיות נפוצות ודרך התמודדות. אם הלקוח קיבל מענה, יוכל לסיים את השיחה. בהמשך, מועברים לנציג שירות, גם הוא ינסה לעזור במגוון בעיות, ושוב – אם הלקוח קיבל מענה, השיחה תסתיים לשביעות רצון הלקוח. רק אם נציג השירות לא יכול לעזור – השיחה תועבר לגורם מקצועי יותר.
2. try & catch

## **יתרונות**

1. ניתן לשלוט בסדר שבו מטפלים בבקשה.
2. Single Responsibility (אחריות אחת) – מפרידים בין האובייקט שמפעיל את הקוד (ה request) קוד לבין האובייקטים שמבצעים את הפעולה (handlers)
3. Open\Closed principle – ניתן להוסיף handlers חדשים בלי לפגוע בקיימים.

## **חסרונות**

1. יהיו בקשות שלא יטופלו

# **Command (פקודה)**

## **תיאור במשפט קצר**

הגדרת אובייקט שמייצג פעולה לביצוע. האובייקט כולל את הנתונים הדרושים לביצוע הפעולה, ואת הפעולה עצמה. הגדרת האובייקט מאפשרת הפרדה בין הגורם שמזמין את ביצוע הפעולה, לבין הגורם שמבצע אותה בפועל, וכן – העברת פעולה כפרמטר לפונקציה

## **הבעיה התכנותית**

בכל תוכנה שיש בה UI, ישנם כפתורים רבים. נניח שנרצה לייצר תפריט עם אופציות שונות לפעילות בתוכנה. כל הכפתורים בתפריט נראים אותו הדבר, אבל הפעולה שלהם שונה.

במחשבה ראשונה נייצר מחלקת בסיס של כפתור, וממנה יירשו כפתורים ספציפיים לפי הפעולה שלהם. זה אומר שיהיו הרבה מאד מחלקות – כל כפתור יהיה מחלקה בפני עצמה.

אבל מה יקרה כשנרצה שפעולה מסוימת – נניח שמירה - תתבצע דרך הכפתור בתפריט, או דרך קיצור במקלדת (Ctrl+S), או דרך עוד איזה אייקון? זה אומר שהקוד שכתבנו במקום אחד יצטרך להיות מוכפל בשאר המקומות.

## **דוגמא**

## **הפתרון**

במימוש התבנית ארבעה מרכיבים עיקריים:

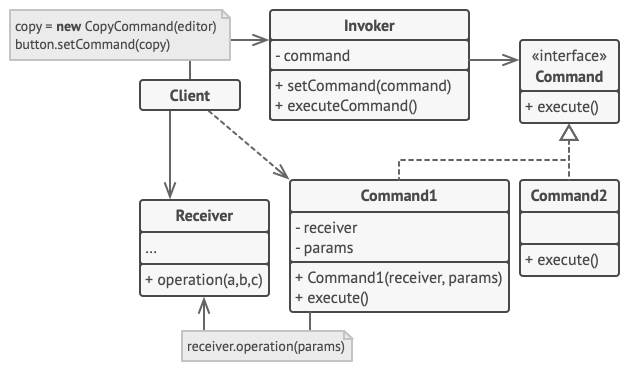
1. Command - ממשק המכיל בדרך כלל מתודה אחת , execute.
2. Concrete Command - מחלקה אשר מממשת את הממשק. בדרך כלל ניצור הרבה מחלקות כאלה - אחת לכל פקודה.
3. Invoker – תפקידה לבצע את הקריאה למתודה execute של ה Concrete Command הנבחר. לעיתים תכיל טבלה של כל המימושים הקיימים ל Command ותקבל רק מספר או שם שמיצגים את המימוש הרצוי (לצורך יעילות).
4. Receiver – המחלקה עליה בעצם מבוצעת הפקודה.

על פי העקרון separation of concerns – הפרדת אחריות, מבצעים הפרדה בין ה invoker – הגורם שמפעיל את הפעולה, לבין ה receiver – המבצע. ההפרדה מתבצעת באמצעות אובייקט ביניים שהוא ה command.

בדוגמא, נפריד את הפעולה שהכפתור צריך לבצע – שמירה, מהכפתור עצמו. הכפתור – אחראי על תצוגה למשתמש, הפעולה – שמירת הנתונים, באחריות הרכיב של לוגיקה עסקית.

בדוגמה של שמירה - הכפתור יקבל אובייקט command שמכיל את כל מה שנדרש לשמירה – ורק יצטרך להפעיל פונקציה, שתוגדר ב interfaceשל ה Command.בקוד של ה command עצמו תהיה פניה ללוגיקה העסקית, אבל הכפתור עצמו לא מעורב בתהליך זה.

## **UML**



צעדים לביצוע:

1. מגדירים interface של ה Command, עם פונקציה execute, ו member מסוג ה receiver – המבצע בפועל
2. מגדירים מחלקות שממשות את ה Interface, לפי פעולות נדרשות.
3. ב invoker – המפעיל, מגדירים reference למשהו מסוג ה interface.
4. בבנית ה Invoker מגדירים מה ה command הספציפי של ה Invoker (מייצרים מופע של אחת מהמחלקות שממשות את ה Interface)
5. כשצריך לבצע את הפעולה, ה invoker מזמן את הפונקציה execute.

## **דוגמת קוד**

**using** **System**;

*//the Command Interface*

**public** **interface** ICommand

{

**void** Execute();

}

*//The ConcreteCommands*

**public** **class** **DVDPlayCommand** : ICommand

{

**public** DVDPlayCommand(){}

**public** **void** Execute()

{

Console.WriteLine("DVD Started.");

}

}

**public** **class** **DVDStopCommand** : ICommand

{

**public** DVDStopCommand(){}

**public** **void** Execute()

{

Console.WriteLine("DVD Stopped.");

}

}

**public** **class** **VCRPlayCommand** : ICommand

{

**public** VCRPlayCommand(){}

**public** **void** Execute()

{

Console.WriteLine("VCR Started.");

}

}

**public** **class** **VCRStopCommand** : ICommand

{

**public** VCRStopCommand(){}

**public** **void** Execute()

{

Console.WriteLine("VCR Stopped.");

}

}

*//The Invoker*

**public** **class** **Remote**

{

**public** Remote(){}

**public** **void** Invoke(ICommand cmd )

{

Console.WriteLine("Invoking.......");

cmd.Execute();

}

}

*//The Client*

**public** **class** **Client**

{

**public** Client(){}

**public** **static** int Main(String[] args)

{

*//Instantiate the invoker object*

Remote remote = **new** Remote();

*//Instantiate DVD related commands and pass them to invoker object*

DVDPlayCommand dvdPlayCommand = **new** DVDPlayCommand();

remote.Invoke(dvdPlayCommand);

DVDStopCommand dvdStopCommand = **new** DVDStopCommand();

remote.Invoke(dvdStopCommand);

*//Instantiate VCR related commands and pass them to invoker object*

VCRPlayCommand vcrPlayCommand = **new** VCRPlayCommand();

remote.Invoke(vcrPlayCommand);

VCRStopCommand vcrStopCommand = **new** VCRStopCommand();

remote.Invoke(vcrStopCommand);

**return** 0;

}

}

## **דוגמאות נוספות**

1. שירות אל-תור בבנק – כל לקוח שמשתמש בשירות מציין מה הפעולה לביצוע, ומצרף את הנתונים הנדרשים – צ'ק, מזומן, וכו'. המעטפות נשמרות בתיבת האלתור (שהיא ה invoker במקרה זה), ומועברות לביצוע ה receiver – פקיד הבנק.
2. מסעדה – מלצר מקבל הזמנה עם כל פרטיה, ומעביר לטבח. במקרה זה, המלצר הוא ה Invoker, הטבח הוא ה receiver, וה command הוא ההזמנה עצמה על כל פרטיה.

שימושים נפוצים

1. אפשרות של ביטול פעולות – אם כל הפעולות נשמרות כאובייקטים, אפשר לשמור אותם במחסנית, ובמידת הצורך (לדוגמא: טרנזקציה ב DB) לבטל את כל הפעולות (במקרה כזה בדרך כלל ה command יאחסן גם מידע על מצב האובייקט לפני הפעולה שלו)
2. ניהול תור של פעולות – הפעולות מגיעות כאובייקט, נשמרות בתור, ומטופלות לפי הסדר.
3. GUI – הפקדים רק מפעילים commands, בלי להיכנס לעומק הלוגיקה העסקית המתבצעת.

## **יתרונות**

1. Principle Single Responsibility – מפרידים בין המחלקות שמפעילות פעולה לבין מחלקות שמבצעות את הפעולה (handlers)
2. Open\Closed Principle – ניתן להוסיף commands חדשים בלי לפגוע בקיימים.
3. מאפשר לבצע מנגנון של ביטול פעולה.
4. מאפשר לבצע מנגנון של deferred operation – דחית פעולה לביצוע בזמן אחר (במידה ומגיעים מספר commands בו זמנית, ניתן לשמור אותם בתור ולבצע אותם כשיסתיים command אחר)
5. אפשר להרכיב מספר command פשוטים ל command מורכב יותר.

## **חסרונות**

1. הקוד הופך להיות קצת מורכב בגלל שבעצם מוסיפים שכבה של קוד – ה command – בין המפעיל לביצוע.

# **Iterator (אובייקט למעבר על איברי קבוצה)**

## **תיאור במשפט קצר**

אובייקט שמשמש למעבר על איברי קבוצה ולגישה אליהם.

## **הבעיה התכנותית**

אוספים הם אחד מסוגי הנתונים הנפוצים ביותר בתכנות. אוסף למעשה הוא רק אובייקט שמכיל קבוצת אלמנטים. רוב האוספים מאחסנים את האלמנטים שלהם ברשימות פשוטות. עם זאת, חלקם מבוססים על ערימות, עצים, גרפים ומבני נתונים מורכבים אחרים.

בלי קשר למבנה האוסף, אוסף חייב לספק דרך כלשהי לגשת לאלמנטים שלו כדי שקוד אחר יוכל להשתמש באלמנטים האלה. צריכה להיות דרך לעבור על כל אלמנט באוסף מבלי לגשת לאותם אלמנטים שוב ושוב.

זה עשוי להישמע כמו עבודה קלה אם מדובר באוסף המבוסס על רשימה, פשוט מבצעים לולאה על כל האלמנטים. אבל איך עוברים ברצף על אלמנטים של מבנה נתונים מורכב, כמו עץ? לעיתים נרצה להשתמש ב BFS, לעיתים ב DFS, ולעיתים פשוט גישה אקראית לרכיבי העץ.

הוספת עוד ועוד אלגוריתמי מעבר לאוסף מטשטשת בהדרגה את האחריות העיקרית שלו, שהיא **אחסון** נתונים יעיל. בנוסף, חלק מהאלגוריתמים עשויים להיות מותאמים לאפליקציה ספציפית, כך שלכלול אותם במחלקת אוסף גנרית יהיה מוזר.

מצד שני, לקוד שאמור לעבוד עם אוספים שונים אולי אפילו לא אכפת איך הם מאחסנים את האלמנטים שלהם, אלא רק לגשת לאלמנטים בצורה הרצויה. עם זאת, מכיוון שהאוספים כולם מספקים דרכים שונות לגשת לאלמנטים שלהם, אין לך אפשרות מלבד לקשר את הקוד שלך למחלקות האוסף הספציפיות.

## **דוגמא**

נתון עץ. מעוניינים לבצע מעבר על אלמנטים של עץ בצורות שונות: Breadth-first Search (BFS) , Depth-first Search (DFS), גישה אקראית לאיברי האוסף.

## **הפתרון**

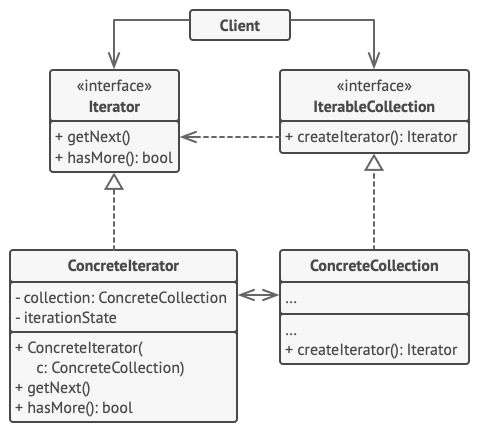
מפרידים בין האוסף – שאחראי לאחסון הנתונים, לבין הקוד שעובר על האלמנטים שלו.

בנוסף ליישום האלגוריתם עצמו, iterator מסתיר את כל פרטי האיטרציה, כגון המיקום הנוכחי וכמה אלמנטים נותרו עד הסוף. בזכות עובדה זו, מספר iterators יכולים לעבור על אותו אוסף בו זמנית, ללא תלות זה בזה.

בדרך כלל, Iteratorים מספקים פונקציה עיקרית אחת להבאת אלמנטים מהאוסף. קוד שמשתמש ב iterator יכול להמשיך להפעיל את הפונקציה הזו עד שהיא לא מחזירה כלום, מה שאומר שה- iterator עבר את כל האלמנטים.

כל ה- Iteratorים חייבים ליישם את אותו ממשק. זה הופך את קוד הלקוח לתואם לכל סוג אוסף או לכל אלגוריתם מעבר כל עוד יש iterator מתאים. אם צריך דרך מיוחדת לעבור על אוסף, פשוט יוצרים מחלקת iterator חדשה, מבלי לשנות את האוסף או את הלקוח.

## **UML**



## **דוגמת קוד**

ב C# השימוש ב iterator ממש מובנה בשפה, באמצעות IEnumerator ו IEnumerable.

**using** System;

**using** System.Collections;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.Iterator.Conceptual

{

**abstract** **class** **Iterator** : IEnumerator

{

**object** IEnumerator.Current => Current();

// Returns the key of the current element

**public** **abstract** **int** Key();

// Returns the current element

**public** **abstract** **object** Current();

// Move forward to next element

**public** **abstract** **bool** MoveNext();

// Rewinds the Iterator to the first element

**public** **abstract** **void** Reset();

}

**abstract** **class** **IteratorAggregate** : IEnumerable

{

// Returns an Iterator or another IteratorAggregate for the implementing

// object.

**public** **abstract** IEnumerator GetEnumerator();

}

// Concrete Iterators implement various traversal algorithms. These classes

// store the current traversal position at all times.

**class** **AlphabeticalOrderIterator** : Iterator

{

**private** WordsCollection \_collection;

// Stores the current traversal position. An iterator may have a lot of

// other fields for storing iteration state, especially when it is

// supposed to work with a particular kind of collection.

**private** **int** \_position = -1;

**private** **bool** \_reverse = **false**;

**public** AlphabeticalOrderIterator(WordsCollection collection, **bool** reverse = **false**)

{

**this**.\_collection = collection;

**this**.\_reverse = reverse;

**if** (reverse)

{

**this**.\_position = collection.getItems().Count;

}

}

**public** **override** **object** Current()

{

**return** **this**.\_collection.getItems()[\_position];

}

**public** **override** **int** Key()

{

**return** **this**.\_position;

}

**public** **override** **bool** MoveNext()

{

**int** updatedPosition = **this**.\_position + (**this**.\_reverse ? -1 : 1);

**if** (updatedPosition >= 0 && updatedPosition < **this**.\_collection.getItems().Count)

{

**this**.\_position = updatedPosition;

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**public** **override** **void** Reset()

{

**this**.\_position = **this**.\_reverse ? **this**.\_collection.getItems().Count - 1 : 0;

}

}

// Concrete Collections provide one or several methods for retrieving fresh

// iterator instances, compatible with the collection class.

**class** **WordsCollection** : IteratorAggregate

{

List<**string**> \_collection = **new** List<**string**>();

**bool** \_direction = **false**;

**public** **void** ReverseDirection()

{

\_direction = !\_direction;

}

**public** List<**string**> getItems()

{

**return** \_collection;

}

**public** **void** AddItem(**string** item)

{

**this**.\_collection.Add(item);

}

**public** **override** IEnumerator GetEnumerator()

{

**return** **new** AlphabeticalOrderIterator(**this**, \_direction);

}

}

**class** **Program**

{

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

{

// The client code may or may not know about the Concrete Iterator

// or Collection classes, depending on the level of indirection you

// want to keep in your program.

**var** **collection** = **new** WordsCollection();

collection.AddItem("First");

collection.AddItem("Second");

collection.AddItem("Third");

Console.WriteLine("Straight traversal:");

**foreach** (**var** **element** **in** collection)

{

Console.WriteLine(element);

}

Console.WriteLine("\nReverse traversal:");

collection.ReverseDirection();

**foreach** (**var** **element** **in** collection)

{

Console.WriteLine(element);

}

}

}

}

## **דוגמאות נוספות**

1. כל שם עצם קיבוצי שלמעשה מייצג רשימה של אלמנטים.
2. סיור במוזיאון – מעוניינים לעבור בין כל התצוגות, אבל כל אחד מעונין לבצע את הסיור בסדר אחר.

## **יתרונות**

1. Principle Single Responsibility – ה collection אחראי לאחסון הנתונים, ה iterator אחראי למעבר על הנתונים.
2. Open\Closed Principle – ניתן להוסיף collections חדשים ו-iterators נוספים בלי לפגוע בקוד קיים.
3. אפשר לעבור על אוסף אחד בכמה צורות במקביל.
4. אפשר לעצור איטרציה מסוימת ולהמשיך אותה מאותה נקודה בזמן אחר, מבלי לפגוע באיטרציות אחרות או מבלי לחסום פעילות אחרת על האוסף.

## **חסרונות**

1. כשמדובר באוספים פשוטים זה מוגזם ומיותר להשתמש ב iterator
2. יתכן שבאוספים מסוימים יותר יעיל לעבור על האלמנטים שלהם ישירות במקום להשתמש באובייקט נוסף שמבצע את זה.

# **Mediator (מתווך)**

## **תיאור במשפט קצר**

מצמצם תלויות בין אובייקטים כך ששיתוף פעולה בין אובייקטים מתבצע רק דרך גורם מתווך.

## **הבעיה התכנותית**

לעיתים ישנם מספר אובייקטים שמעורבים בפעילות תכנותית כלשהי. הפעילות של אובייקטים מסוימים תלויה באובייקטים אחרים, התלות הזו גורמת שהקוד לא ניתן לשימוש חוזר בפעילויות תכנותיות אחרות.

## **דוגמא**

קיימים פקדים שונים בטופס: checkbox לסימון אם מעוניינים לקבל עדכונים, אם מסמנים אותו, נפתחת תיבת טקסט להזנת אימייל, כפתור שמירה צריך לבדוק תקינות הנתונים בפקדים האחרים וכו'. ישנה תקשורת רבה בין הפקדים, ואם נרצה לבודד את הפקדים ולהשתמש בהם בטפסים נוספים תהיה בעיה, כי כפתור השמירה למשל לא עצמאי – הוא בודק פקדים אחרים בטופס, או ה checkbox כדוגמא – גם הוא תלוי בפקד של האימייל, וקובע שבסימון שלו הפקד הנוסף יוצג.

## **הפתרון**

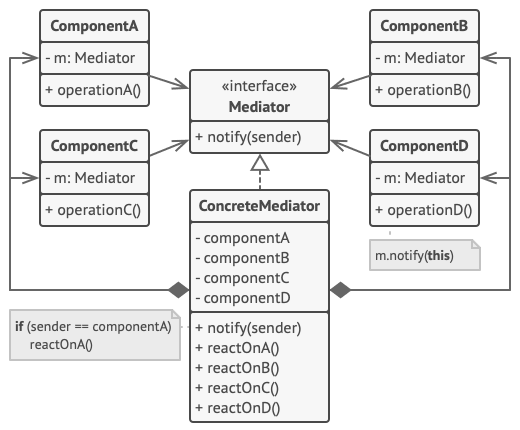
מגדירים גורם אחד אחראי, וכל התקשורת מתבצעת דרכו. זהו ה Mediator.

בדוגמא של הטופס, הטופס עצמו אחראי לניהול ההתרחשויות בטופס: אם לחצו על שמירה, כפתור שמירה מעדכן את הטופס שלחצו עליו. הטופס יקבע מה קורה לפקדים אחרים בזמן שמירה.

אם סימנו את ה checkbox של העדכונים – הוא מודיע לטופס. הטופס ידאג שהפקד של האימייל יוצג.

הטופס הוא בעצם ה mediator – מתווך.

## **UML**



צעדים לביצוע:

1. זיהוי קבוצה של אובייקטים שתלויים אחד בשני, והפרדה יכולה להביא לניהול פשוט יותר או שימוש חוזר באובייקטים.
2. הגדרת המתווך ואופן התקשורת בינו לבין האובייקטים. בדרך כלל תהיה פונקציה notify או משהו דומה שתזומן ע"י האובייקטים. notify תקבל בדרך כלל את האובייקט שגרם להפעלה שלה.

מקובל להגדיר אותו כ interface, ולממש באופן ספציפי היכן שרוצים.

1. הגדרת המחלקות השונות לאובייקטים שקשורים למתווך. בכל מחלקה כזו יהיה ייחוס למתווך.
2. כשנוצר צורך לעדכן גורם אחר, מעדכנים את המתווך באמצעות הפונקציה notify.

## **דוגמת קוד**

**using** System;

**namespace** **RefactoringGuru**.DesignPatterns.Mediator.Conceptual

{

// The Mediator interface declares a method used by components to notify the

// mediator about various events. The Mediator may react to these events and

// pass the execution to other components.

**public** **interface** **IMediator**

{

**void** Notify(**object** sender, **string** ev);

}

// Concrete Mediators implement cooperative behavior by coordinating several

// components.

**class** **ConcreteMediator** : IMediator

{

**private** Component1 \_component1;

**private** Component2 \_component2;

**public** ConcreteMediator(Component1 component1, Component2 component2)

{

**this**.\_component1 = component1;

**this**.\_component1.SetMediator(**this**);

**this**.\_component2 = component2;

**this**.\_component2.SetMediator(**this**);

}

**public** **void** Notify(**object** sender, **string** ev)

{

**if** (ev == "A")

{

Console.WriteLine("Mediator reacts on A and triggers folowing operations:");

**this**.\_component2.DoC();

}

**if** (ev == "D")

{

Console.WriteLine("Mediator reacts on D and triggers following operations:");

**this**.\_component1.DoB();

**this**.\_component2.DoC();

}

}

}

// The Base Component provides the basic functionality of storing a

// mediator's instance inside component objects.

**class** **BaseComponent**

{

**protected** IMediator \_mediator;

**public** BaseComponent(IMediator mediator = **null**)

{

**this**.\_mediator = mediator;

}

**public** **void** SetMediator(IMediator mediator)

{

**this**.\_mediator = mediator;

}

}

// Concrete Components implement various functionality. They don't depend on

// other components. They also don't depend on any concrete mediator

// classes.

**class** **Component1** : BaseComponent

{

**public** **void** DoA()

{

Console.WriteLine("Component 1 does A.");

**this**.\_mediator.Notify(**this**, "A");

}

**public** **void** DoB()

{

Console.WriteLine("Component 1 does B.");

**this**.\_mediator.Notify(**this**, "B");

}

}

**class** **Component2** : BaseComponent

{

**public** **void** DoC()

{

Console.WriteLine("Component 2 does C.");

**this**.\_mediator.Notify(**this**, "C");

}

**public** **void** DoD()

{

Console.WriteLine("Component 2 does D.");

**this**.\_mediator.Notify(**this**, "D");

}

}

**class** **Program**

{

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

{

// The client code.

Component1 component1 = **new** Component1();

Component2 component2 = **new** Component2();

**new** ConcreteMediator(component1, component2);

Console.WriteLine("Client triggets operation A.");

component1.DoA();

Console.WriteLine();

Console.WriteLine("Client triggers operation D.");

component2.DoD();

}

}

}

פלט של הקוד:

Client triggers operation A.

Component 1 does A.

Mediator reacts on A and triggers following operations:

Component 2 does C.

Client triggers operation D.

Component 2 does D.

Mediator reacts on D and triggers following operations:

Component 1 does B.

Component 2 does C.

## **דוגמאות נוספות**

1. ארגון אירוע – במקום שכל המשתתפים בארגון האירוע יתקשרו ביניהם, הבמאית תתאם עם האחראי על התלבושות ועם האחראי על התאורה ועם השחקניות, והאחראי על התאורה יתאם עם בעל המפתחות לאולם וכו', יש בדרך כלל גורם אחד אחראי שמתווך בין כולם, כל מי שצריך משהו פונה אליו – והוא מעביר את המידע למקום המתאים.
2. מטוסים שרוצים לנחות – לא מתאמים מידע ביניהם אלא מגדל הפיקוח מנהל את הנחיתות של כולם.

## **יתרונות**

1. Principle Single Responsibility – מחלצים את התקשורת בין הרכיבים מהרכיבים עצמם, ומרכזים אותה במקום אחד, כך הרבה יותר קל לנהל אותה.
2. Open\Closed Principle – ניתן להוסיף mediators חדשים בלי לפגוע בקיימים.
3. צמצום התלות בין רכיבים שונים בתוכנית.
4. אפשרות לשימוש חוזר בתוכניות אחרות.

## **חסרונות**

1. במהלך הזמן mediator עלול להפוך ל God object – אובייקט שמנהל יותר מדי דברים, מכיל כל מיני פונקציות לא קשורות ביניהן וכו'.

# **Memento (מזכרת)**

## **תיאור במשפט קצר**

תבנית המאפשרת לשמור ולשחזר את המצב הקודם של אובייקט (לפני שבוצעה פעולה מסוימת, שינויים וכדו') מבלי לחשוף את פרטי היישום שלו.

## **הבעיה התכנותית**

מעונינים לבצע פעולות על אובייקט, אך להיות מסוגלים לשחזר את מצבו באם הפעולה לא תצליח.

כדי לשחזר את מצב האובייקט צריכים, לכאורה, שנתוניו יהיו חשופים – מה שנוגד את עקרון הכימוס (encapsulation) של OOP.

מצד שני – אם יהיו נתונים שהם Private, איך ניתן יהיה לשחזר אותם?

נשים לב: למרות שזה נשמע קצת דומה ל prototype, צריך לזכור ש Prototype קשור רק ליצירת אובייקטים, ולא ניהול של הנתונים שלהם לאורך מחזור החיים של האובייקט.

## **דוגמא**

עורך טקסט שמאפשר עיצוב של הטקסט, הוספת תמונות , ועוד. בכל מצב צריכים לאפשר ביטול של הפעולה הקודמת שבוצעה. למעשה צריכים לשמור את הנתונים על כל האובייקטים שיש כרגע בטקסט – הטקסט עצמו, תמונות מצורפות, גודל שלהן, מיקום, וכו'.

## **הפתרון**

כדי לשמור על הכימוס – האובייקט עצמו אחראי ליצירת עותק של עצמו, ולשחזור של עצמו למצב קודם במידת הצורך. תזמון השחזור נעשה מחוץ לאובייקט.

Memento מכיל מצב של אובייקט שיש לשחזר. המקור יוצר ומאחסן מצבים באובייקטים של Memento ואובייקט Caretaker אחראי לשחזר מצב אובייקט מממנטו. יצרנו כיתות Memento, Originator ו-CareTaker.

בתבנית זו יש שלושה גורמים:

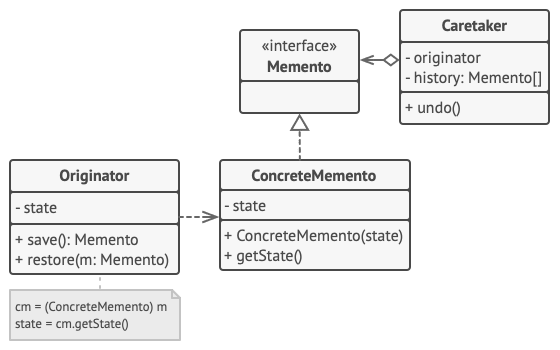
memento – מכיל מצב של אובייקט שיתכן ויצטרכו לשחזר למצב זה.

originator – אובייקט במצב מסוים (מצב הכוונה נתונים מסוימים שמאוחסנים באובייקט, ויכולים להשתנות – כל שינוי כביכול נחשב כ"מצב" אחר)

caretaker – אובייקט שמבצע שינוי כלשהו על ה originator, אך מעונין באפשרות לבצע ביטול של הפעולה. הוא זה שיבקש את ה memento.

המטפל (caretaker) מעונין לבצע פעולה על האובייקט המקורי (originator). לפני ביצוע הפעולה, הוא מבקש ממנו "מזכרת" – memento, בדרך כלל זו פונקציה של createMemento או משהו דומה, ושומר אותה אצלו. אם נדרש שחזור של האובייקט, המטפל יזמן פונקציה של setMemento על האובייקט המקורי, ויעביר בה את ה memento שהוא קיבל ממנו בתחילת התהליך.

## **UML**



## **דוגמת קוד**

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Threading;

**namespace** **RefactoringGuru**.DesignPatterns.Memento.Conceptual

{

// The Originator holds some important state that may change over time. It

// also defines a method for saving the state inside a memento and another

// method for restoring the state from it.

**class** **Originator**

{

// For the sake of simplicity, the originator's state is stored inside a

// single variable.

**private** **string** \_state;

**public** Originator(**string** state)

{

**this**.\_state = state;

Console.WriteLine("Originator: My initial state is: " + state);

}

// The Originator's business logic may affect its internal state.

// Therefore, the client should backup the state before launching

// methods of the business logic via the save() method.

**public** **void** DoSomething()

{

Console.WriteLine("Originator: I'm doing something important.");

**this**.\_state = **this**.GenerateRandomString(30);

Console.WriteLine($"Originator: and my state has changed to: {\_state}");

}

**private** **string** GenerateRandomString(**int** length = 10)

{

**string** allowedSymbols = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

**string** result = **string**.Empty;

**while** (length > 0)

{

result += allowedSymbols[**new** Random().Next(0, allowedSymbols.Length)];

Thread.Sleep(12);

length--;

}

**return** result;

}

// Saves the current state inside a memento.

**public** IMemento Save()

{

**return** **new** ConcreteMemento(**this**.\_state);

}

// Restores the Originator's state from a memento object.

**public** **void** Restore(IMemento memento)

{

**if** (!(memento **is** ConcreteMemento))

{

**throw** **new** Exception("Unknown memento class " + memento.ToString());

}

**this**.\_state = memento.GetState();

Console.Write($"Originator: My state has changed to: {\_state}");

}

}

// The Memento interface provides a way to retrieve the memento's metadata,

// such as creation date or name. However, it doesn't expose the

// Originator's state.

**public** **interface** **IMemento**

{

**string** GetName();

**string** GetState();

**DateTime** GetDate();

}

// The Concrete Memento contains the infrastructure for storing the

// Originator's state.

**class** **ConcreteMemento** : IMemento

{

**private** **string** \_state;

**private** **DateTime** \_date;

**public** ConcreteMemento(**string** state)

{

**this**.\_state = state;

**this**.\_date = **DateTime**.Now;

}

// The Originator uses this method when restoring its state.

**public** **string** GetState()

{

**return** **this**.\_state;

}

// The rest of the methods are used by the Caretaker to display

// metadata.

**public** **string** GetName()

{

**return** $"{this.\_date} / ({this.\_state.Substring(0, 9)})...";

}

**public** **DateTime** GetDate()

{

**return** **this**.\_date;

}

}

// The Caretaker doesn't depend on the Concrete Memento class. Therefore, it

// doesn't have access to the originator's state, stored inside the memento.

// It works with all mementos via the base Memento interface.

**class** **Caretaker**

{

**private** List<IMemento> \_mementos = **new** List<IMemento>();

**private** Originator \_originator = **null**;

**public** Caretaker(Originator originator)

{

**this**.\_originator = originator;

}

**public** **void** Backup()

{

Console.WriteLine("\nCaretaker: Saving Originator's state...");

**this**.\_mementos.Add(**this**.\_originator.Save());

}

**public** **void** Undo()

{

**if** (**this**.\_mementos.Count == 0)

{

**return**;

}

**var** **memento** = **this**.\_mementos.Last();

**this**.\_mementos.Remove(memento);

Console.WriteLine("Caretaker: Restoring state to: " + memento.GetName());

**try**

{

**this**.\_originator.Restore(memento);

}

**catch** (Exception)

{

**this**.Undo();

}

}

**public** **void** ShowHistory()

{

Console.WriteLine("Caretaker: Here's the list of mementos:");

**foreach** (**var** **memento** **in** **this**.\_mementos)

{

Console.WriteLine(memento.GetName());

}

}

}

**class** **Program**

{

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

{

// Client code.

Originator originator = **new** Originator("Super-duper-super-puper-super.");

Caretaker caretaker = **new** Caretaker(originator);

caretaker.Backup();

originator.DoSomething();

caretaker.Backup();

originator.DoSomething();

caretaker.Backup();

originator.DoSomething();

Console.WriteLine();

caretaker.ShowHistory();

Console.WriteLine("\nClient: Now, let's rollback!\n");

caretaker.Undo();

Console.WriteLine("\n\nClient: Once more!\n");

caretaker.Undo();

Console.WriteLine();

}

}

}

## **דוגמאות נוספות**

1. טרנזקציה של פעולות מול ה DB – אם הטרנזקציה נכשלת, מבטלים את הפעולות ומחזירים את האובייקט למצבו הקודם.

## **יתרונות**

1. אפשר לייצר תמונת מצב של אובייקט בלי להפר את הכלל של כימוס.
2. האובייקט עצמו לא צריך לנהל את ההיסטוריה שלו, זה התפקיד של ה caretaker.

## **חסרונות**

1. הקוד של התוכנה עלול לדרוש הרבה זיכרון אם מיצריםmementos לעיתים קרובות.
2. ה"מטפל", caretaker, עוקב אחרי ה originator כדי להחליט מתי אפשר למחוק mementos
3. יש שפות שלא יכולות לאכוף את זה שלא נוגעים ב mementos.

# **Observer (משקיף)**

## **תיאור במשפט קצר**

תבנית שבונה מנגנון להרשמה לקבלת התראות לגבי אירועים שונים שמתרחשים על אובייקט עליו "משקיפים".

## **הבעיה התכנותית**

ידוע שקיים אובייקט X, ואירוע מסוים עתיד להתרחש בו. אובייקטים Y,Z וכו' ממתינים להתרחשות האירוע כדי לבצע פונקציונליות מסוימת. כיצד הם יתעדכנו על האירוע? ישנה אפשרות שהם יבדקו בפעימות מוגדרות מראש האם האירוע התרחש. כמובן שזה יגרום לביצוע קוד מיותר. כל מה שנדרש זה שמישהו יעדכן אותם כאשר האירוע מתרחש.

## **דוגמא**

לקוחות ממתינים להגעת מוצר כלשהו לחנות. אפשר לבוא לבקר בחנות מדי יום כדי לבדוק אם המוצר מהדגם הרצוי הגיע. לחילופין, ישנה אפשרות שכשהמוצר יגיע, החנות תודיע לכל לקוחותיה על הגעת המוצר, אבל סביר להניח שלא כל הלקוחות מתעניינים במוצר הזה ואז יפריעו להם העדכונים התכופים על המוצרים שהגיעו לחנות. יהיה יעיל יותר אם המוכרת תרשום את פרטי הלקוחות המתעניינים במוצר מסוים, ותעדכן רק אותם על הגעת המוצר.

## **הפתרון**

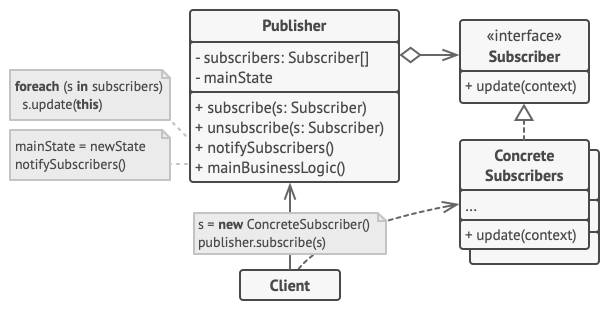
מגדירים publisher(מפרסם) – האובייקט שעליו יש התעניינות לגבי אירוע מסוים שלו (בדוגמא – החנות).

מגדירים subscriber\s – אחד או יותר, אלו ירשמו לקבלת עדכון כשהאירוע מתרחש (בדוגמא – הלקוחות).

ב publisher יוגדרו פונקציות לקבלת עדכון, להסרה מקבלת עדכונים, ופונקציה של העדכון על האירוע.

כל subscriber יגדיר פונקציה שצריכה להתבצע כאשר האירוע מתרחש.

## **UML**



## **דוגמת קוד**

**using** System;

**using** System.Collections.Generic;

**using** System.Threading;

**namespace** **RefactoringGuru**.DesignPatterns.Observer.Conceptual

{

**public** **interface** **IObserver**

{

// Receive update from subject

**void** Update(ISubject subject);

}

**public** **interface** **ISubject**

{

// Attach an observer to the subject.

**void** Attach(IObserver observer);

// Detach an observer from the subject.

**void** Detach(IObserver observer);

// Notify all observers about an event.

**void** Notify();

}

// The Subject owns some important state and notifies observers when the

// state changes.

**public** **class** **Subject** : ISubject

{

// For the sake of simplicity, the Subject's state, essential to all

// subscribers, is stored in this variable.

**public** **int** State { **get**; **set**; } = -0;

// List of subscribers. In real life, the list of subscribers can be

// stored more comprehensively (categorized by event type, etc.).

**private** List<IObserver> \_observers = **new** List<IObserver>();

// The subscription management methods.

**public** **void** Attach(IObserver observer)

{

Console.WriteLine("Subject: Attached an observer.");

**this**.\_observers.Add(observer);

}

**public** **void** Detach(IObserver observer)

{

**this**.\_observers.Remove(observer);

Console.WriteLine("Subject: Detached an observer.");

}

// Trigger an update in each subscriber.

**public** **void** Notify()

{

Console.WriteLine("Subject: Notifying observers...");

**foreach** (**var** **observer** **in** \_observers)

{

observer.Update(**this**);

}

}

// Usually, the subscription logic is only a fraction of what a Subject

// can really do. Subjects commonly hold some important business logic,

// that triggers a notification method whenever something important is

// about to happen (or after it).

**public** **void** SomeBusinessLogic()

{

Console.WriteLine("\nSubject: I'm doing something important.");

**this**.State = **new** Random().Next(0, 10);

Thread.Sleep(15);

Console.WriteLine("Subject: My state has just changed to: " + **this**.State);

**this**.Notify();

}

}

// Concrete Observers react to the updates issued by the Subject they had

// been attached to.

**class** **ConcreteObserverA** : IObserver

{

**public** **void** Update(ISubject subject)

{

**if** ((subject **as** Subject).State < 3)

{

Console.WriteLine("ConcreteObserverA: Reacted to the event.");

}

}

}

**class** **ConcreteObserverB** : IObserver

{

**public** **void** Update(ISubject subject)

{

**if** ((subject **as** Subject).State == 0 || (subject **as** Subject).State >= 2)

{

Console.WriteLine("ConcreteObserverB: Reacted to the event.");

}

}

}

**class** **Program**

{

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

{

// The client code.

**var** **subject** = **new** Subject();

**var** **observerA** = **new** ConcreteObserverA();

subject.Attach(observerA);

**var** **observerB** = **new** ConcreteObserverB();

subject.Attach(observerB);

subject.SomeBusinessLogic();

subject.SomeBusinessLogic();

subject.Detach(observerB);

subject.SomeBusinessLogic();

}

}

}

פלט:

Subject: Attached an observer.

Subject: Attached an observer.

Subject: I'm doing something important.

Subject: My state has just changed to: 2

Subject: Notifying observers...

ConcreteObserverA: Reacted to the event.

ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.

Subject: My state has just changed to: 1

Subject: Notifying observers...

ConcreteObserverA: Reacted to the event.

Subject: Detached an observer.

Subject: I'm doing something important.

Subject: My state has just changed to: 5

Subject: Notifying observers...

## **דוגמאות נוספות**

1. פקדים ב UI – נרשמים לאירוע של הפקד, נניח לחיצה, מעבר עכבר וכו'. כאשר מתרחש האירוע – מזומנת פונקציה מסוימת רצויה.
2. עדכונים לתוכנה – במקום לבדוק כל יום אם יש עדכונים, החברה מעדכנת כשיש עדכון לתוכנה.

## **יתרונות**

1. Open\Closed Principle – ניתן להוסיף subscribers חדשים בלי לפגוע בקוד של ה publisher.
2. אפשר להגדיר קשר בין אובייקטים בזמן ריצה.

## **חסרונות**

1. אין אחריות לסדר שבו מיידעים את ה subscribers על התרחשות האירוע.

# **State (מצב)**

## **תיאור במשפט קצר**

תבנית המאפשרת לאובייקט לשנות את התנהגותו כאשר מצבו הפנימי משתנה. חיצונית זה נראה כאילו האובייקט שינה את המחלקה שלו.

## **הבעיה התכנותית**

המושג state מתייחס לנתון או מספר נתונים של אובייקט שייחד מייצגים מצב מסוים של האובייקט.

מעוניינים שהתנהגות האובייקט תשתנה כאשר ה state שלו משתנה.

כל התנהגות שאמורה להתרחש במצב מסוים, לא אמורה להשפיע על התנהגויות אחרות. אם יש התנהגות חדשה, האחרות לא מושפעות ממנה.

הטמעת התנהגות ספציפית למצב ישירות בתוך מחלקה אינה גמישה מכיוון שהיא מחייבת את המחלקה להתנהגות מסוימת ואי אפשר להוסיף מצב חדש או לשנות את ההתנהגות של מצב קיים מאוחר יותר, ללא תלות במחלקה, מבלי לשנות את המחלקה.

## **דוגמא**

קיים מושג שנקרא מכונת מצבים (State Machine). מכונת מצבים מתארת תכנית שיכולה להימצא בכל רגע נתון באחד מ X שלבים. התוכנית יכולה לעבור ממצב X למצב אחד או יותר, בדרך כלל על פי תנאי מסוים.

פעמים רבות בתחילת הדרך מספר המצבים האפשריים קטן וכן התנאים למעבר בין המצבים, אך בהמשך הפיתוח, מספר המצבים גדל, אפשרויות המעבר בין מצבים מתרחבות וכן גם התנאים למעבר ממצב למצב הולכים ומסתעפים.

ביטוי נוסף לבעיה הוא כל קוד פשוט עם switch ותנאים נוספים בתוך caseים מסוימים ב switch. קוד כזה יכול להסתעף בקלות רבה לקוד מלא ifים ו-caseים, והוא עלול להיות בלתי קריא, קשה לניהול, וחשוף לשגיאות באגים (מישהו אחד משנה את ה if מ or ל && כי זה מתאים לקוד שהוא צריך לפתח, והוא לא יודע שבעצם זה קריטי למצב אחר).

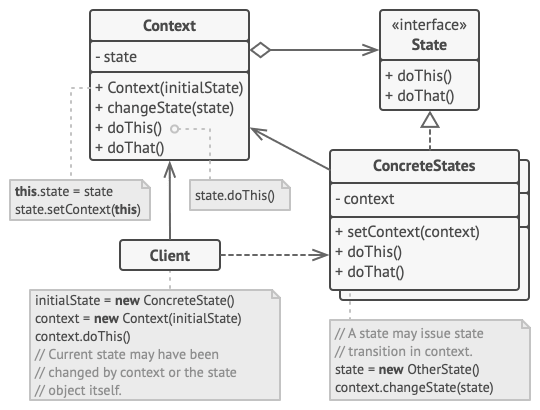
## **הפתרון**

לכל מצב אפשרי של האובייקט, מגדירים מחלקה. מחלקות המצבים כולן ממשות interface אחד.

האובייקט (נקרא גם context) מחזיק ייחוס ל state הנוכחי שלו. בתוכו תהיה גם הפעולה המתאימה ל state הנוכחי. בכך מפרידים את הקוד של ניהול המצבים מהאובייקט עצמו. להעברת האובייקט למצב חדש, משנים את הייחוס למופע מתאים למצב החדש. ההעברה תתבצע בדרך כלל בתוך המחלקות של ה state, שכל אחת מהן מחזקה בעצמה גם ייחוס ל context.

בדוגמא של התכנית, היא עצמה ה context, והמצבים הם ה states. למעבר בין המצבים ה state מחליף את הייחוס ל state של התכנית, לפי תנאים מסוימים.

## **UML**



## **דוגמת קוד**

**using** System;

**namespace** **RefactoringGuru**.DesignPatterns.State.Conceptual

{

// The Context defines the interface of interest to clients. It also

// maintains a reference to an instance of a State subclass, which

// represents the current state of the Context.

**class** **Context**

{

// A reference to the current state of the Context.

**private** State \_state = **null**;

**public** Context(State state)

{

**this**.TransitionTo(state);

}

// The Context allows changing the State object at runtime.

**public** **void** TransitionTo(State state)

{

Console.WriteLine($"Context: Transition to {state.GetType().Name}.");

**this**.\_state = state;

**this**.\_state.SetContext(**this**);

}

// The Context delegates part of its behavior to the current State

// object.

**public** **void** Request1()

{

**this**.\_state.Handle1();

}

**public** **void** Request2()

{

**this**.\_state.Handle2();

}

}

// The base State class declares methods that all Concrete State should

// implement and also provides a backreference to the Context object,

// associated with the State. This backreference can be used by States to

// transition the Context to another State.

**abstract** **class** **State**

{

**protected** Context \_context;

**public** **void** SetContext(Context context)

{

**this**.\_context = context;

}

**public** **abstract** **void** Handle1();

**public** **abstract** **void** Handle2();

}

// Concrete States implement various behaviors, associated with a state of

// the Context.

**class** **ConcreteStateA** : State

{

**public** **override** **void** Handle1()

{

Console.WriteLine("ConcreteStateA handles request1.");

Console.WriteLine("ConcreteStateA wants to change the state of the context.");

**this**.\_context.TransitionTo(**new** ConcreteStateB());

}

**public** **override** **void** Handle2()

{

Console.WriteLine("ConcreteStateA handles request2.");

}

}

**class** **ConcreteStateB** : State

{

**public** **override** **void** Handle1()

{

Console.Write("ConcreteStateB handles request1.");

}

**public** **override** **void** Handle2()

{

Console.WriteLine("ConcreteStateB handles request2.");

Console.WriteLine("ConcreteStateB wants to change the state of the context.");

**this**.\_context.TransitionTo(**new** ConcreteStateA());

}

}

**class** **Program**

{

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

{

// The client code.

**var** **context** = **new** Context(**new** ConcreteStateA());

context.Request1();

context.Request2();

}

}

}

## **דוגמאות נוספות**

1. קובץ יכול להיות במצב טיוטה, מועמד לפרסום, מפורסם. יכול לעבור מטיוטה למועמד לפרסום, ולחזור לטיוטה, אם נניח המנהל עבר על הקובץ ולא אישר את פרסומו בצורה נוכחית.
2. כפתורים של הטלפון פועלים בצורות- שונות בהתאם למצב שלו –

אם משוחרר – כל כפתור מבצע פונקציונליות שהוגדרה לו.

אם הוא נעול – כל לחיצה מציגה א מסך שחרור מנעילה.

אם הסוללה חלשה – מוצגת הודעה שיש לטעון את הטלפון.

## **יתרונות**

1. מפשטת את הקוד מאד – במקום תנאים ו switchים מסורבלים שקשה לעקוב אחרי ביצועם, חלוקה מסודרת וברורה של כל מצב והפעילות המתאימה לו.
2. Single Responsibility principle – כל state אחראי לבצע את הקוד שקשור למצב שלו, ולא כל הקוד זרוק במקום אחד.
3. Open\Closed principle – מאפשר הוספת מצבים חדשים ללא שינוי הקוד של המצבים הקיימים.

## **חסרונות**

1. אם אין הרבה מצבים – החלוקה מחלקות מסרבלת ומעמיסה על התוכנית.

# **Strategy (אסטרטגיה, תכנית פעולה)**

## **תיאור במשפט קצר**

מגדירה משפחה של אלגוריתמים להשגת יעד מסוים. במקום ליישם אלגוריתם בודד, הקוד מקבל החלטה בזמן ריצה באיזה משפחה של אלגוריתמים להשתמש.

## **הבעיה התכנותית**

מעוניינים לבצע אלגוריתם מסוים בתוכנית, שניתן לבצע אותו בדרכים שונות. האלגוריתמים כולם יגיעו לאותה תוצאה, אך הדרך להגיע לתוצאת החישוב יכולה להשתנות. הסיבות לבחירת דרך (אסטרטגיה) אחת על פני דרך אחרת יכולות להיות מגוונות: משך זמן, עלות, יעילות, וכו'.

## **דוגמא**

אפליקציית מפות גוגל לניווט – המטרה היא הגעה ליעד. יש אפשרות לנווט בהליכה רגלית, ברכב פרטי, תחבורה ציבורית וכו'. לפי בחירת המשתמש – יופעל האלגוריתם המתאים.

## **הפתרון**

יצירת interface שמכיל פונקציה שמבצעת את האלגוריתם.

יצירת מחלקות שמממשות את הפונקציה כל אחת בדרכה היא. כל מחלקה היא בעצם strategy.

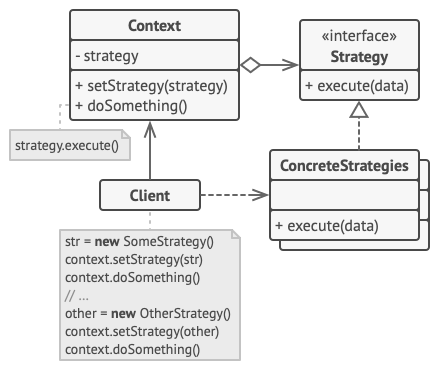
יצירת מחלקה נוספת שנקראת context. מחלקה זו תגדיר חבר מחלקה מסוג ה strategy interface, ותקבל את ה strategy ב ctor. בנוסף, היא גם תגדיר פונקציה או setter ל strategy.

במחלקת ה context תוגדר פונקציה נוספת שתפעיל את האלגוריתם של ה strategy.

בדוגמה של הניווט, יוגדר Route Interface עם פונקציה buildRoute.

כל ניווט הוא בעצם Strategy אחר, שמממש את ה interface. בנוסף, תוגדר מחלקה עם member מסוג ה interface, ופונקציה של navigate. הפונקציה תפעיל את buildRoute של ה strategy שהוכנס אליה ב ctor או ב setter.

## **UML**



## **דוגמת קוד**

**public** **class** **StrategyPatternWiki**

{

**public** **static** **void** Main(String[] args)

{

*// Prepare strategies*

var normalStrategy = **new** NormalStrategy();

var happyHourStrategy = **new** HappyHourStrategy();

var firstCustomer = **new** CustomerBill(normalStrategy);

*// Normal billing*

firstCustomer.Add(1.0, 1);

*// Start Happy Hour*

firstCustomer.Strategy = happyHourStrategy;

firstCustomer.Add(1.0, 2);

*// New Customer*

var secondCustomer = **new** CustomerBill(happyHourStrategy);

secondCustomer.Add(0.8, 1);

*// The Customer pays*

firstCustomer.Print();

*// End Happy Hour*

secondCustomer.Strategy = normalStrategy;

secondCustomer.Add(1.3, 2);

secondCustomer.Add(2.5, 1);

secondCustomer.Print();

}

}

*// CustomerBill as class name since it narrowly pertains to a customer's bill*

**class** **CustomerBill**

{

**private** IList<double> drinks;

*// Get/Set Strategy*

**public** IBillingStrategy Strategy { **get**; **set**; }

**public** CustomerBill(IBillingStrategy strategy)

{

**this**.drinks = **new** List<double>();

**this**.Strategy = strategy;

}

**public** **void** Add(double price, int quantity)

{

**this**.drinks.Add(**this**.Strategy.GetActPrice(price \* quantity));

}

*// Payment of bill*

**public** **void** Print()

{

double sum = 0;

**foreach** (var drinkCost **in** **this**.drinks)

{

sum += drinkCost;

}

Console.WriteLine($"Total due: {sum}.");

**this**.drinks.Clear();

}

}

**interface** IBillingStrategy

{

double GetActPrice(double rawPrice);

}

*// Normal billing strategy (unchanged price)*

**class** **NormalStrategy** : IBillingStrategy

{

**public** double GetActPrice(double rawPrice) => rawPrice;

}

*// Strategy for Happy hour (50% discount)*

**class** **HappyHourStrategy** : IBillingStrategy

{

**public** double GetActPrice(double rawPrice) => rawPrice \* 0.5;

}

## **דוגמאות נוספות**

1. חישובי מחיר סופי של הזמנה – תלוי במועד הרכישה, בסוג הלקוח וכו'.
2. מחלקה המבצעת אימות על נתונים נכנסים עשויה להשתמש ב Strategy לבחירת אלגוריתם אימות בהתאם לסוג הנתונים, מקור הנתונים, בחירת המשתמש או פרמטרים שונים נוספים. גורמים אלה אינם ידועים עד לזמן הריצה ועשויים לדרוש אימות שונה. אלגוריתמי האימות (אסטרטגיות), המובלעים בנפרד מהאובייקט המאמת, עשויים לשמש אובייקטים מאמתים אחרים באזורים שונים של המערכת (או אפילו מערכות שונות) ללא שכפול קוד.

## **יתרונות**

1. אפשר להחליף את האלגוריתם בזמן ריצה.
2. מפרידים את פרטי המימוש מהקוד של האלגוריתם לבין הקוד שמשתמש בו.
3. מחליפים הורשה בהרכבה (composition over inheritance).
4. Open\Closed Principle – ניתן להוסיף strategies חדשים בלי לפגוע בקיימים.

## **חסרונות**

1. אם יש מספר מצומצם של אלגוריתמים שמתחלפים לעיתים רחוקות, מימוש באמצעות strategy סתם יסרבל את הקוד.
2. ה client חייב לדעת את ההבדלים בין ה strategies כדי לדעת במה לבחור.
3. יש שפות תוכנה שמאפשרות להשתמש בפונקציות כפרמטר, ואז אפשר להחליף את הקוד באמצעות פונקציות שונות ולא אובייקטים. זה קצר יותר ומסורבל פחות.

# **Template Method (תבנית)**

## **תיאור במשפט קצר**

מגדירה את השלד של אלגוריתם במחלקת בסיס, מאפשרת לתת-מחלקות לעקוף שלבים ספציפיים של האלגוריתם מבלי לשנות את המבנה שלו.

## **הבעיה התכנותית**

ישנו אלגוריתם שהמבנה שלו די דומה במקרים רבים, אך חלקים ממנו משתנים בין המקרים. מעוניינים לאחד את הקוד לכולם – אבל לאפשר הבדל בין הסוגים השונים במידת הצורך.

## **דוגמא**

תוכנה לכריית מידע מקבצים מסוגים שונים. קבצי word, קבצי excel, קבצי PDF וכו'.בכל המקרים צריך לפתוח את הקובץ, לשלוף ממנו את המידע, להמיר אותו לצורה אחידה של המידע, ולנתח אותו. רק ההמרה של הנתונים לאיזשהו מבנה אחיד של הנתונים יהיה שונה בין הסוגים השונים של הקבצים.

## **הפתרון**

התבנית מציעה לפרק אלגוריתם לסדרה של שלבים, להפוך את השלבים הללו לפונקציות, ולהכניס סדרה של קריאות לפונקציות הללו בתוך פונקציית-תבנית (template method) אחת. השלבים עשויים להיות מופשטים או שיש להם מימוש ברירת מחדל. כדי להשתמש באלגוריתם, שימוש ספציפי אמור לספק תת-מחלקה משלו, ליישם את כל השלבים המופשטים ולעקוף כמה מהשלבים האופציונליים במידת הצורך (אך לא את פונקציית התבנית עצמה).

## **UML**



## **דוגמת קוד (Java)**

abstract class OrderProcessTemplate

{

    public boolean isGift;

    public abstract void doSelect();

    public abstract void doPayment();

    public final void giftWrap()

    {

        try

        {

            System.out.println("Gift wrap successful");

        }

        catch (Exception e)

        {

            System.out.println("Gift wrap unsuccessful");

        }

    }

    public abstract void doDelivery();

    public final void processOrder(boolean isGift)

    {

        doSelect();

        doPayment();

        if (isGift) {

            giftWrap();

        }

        doDelivery();

    }

}

class NetOrder extends OrderProcessTemplate

{

    @Override

    public void doSelect()

    {

        System.out.println("Item added to online shopping cart");

        System.out.println("Get gift wrap preference");

        System.out.println("Get delivery address.");

    }

    @Override

    public void doPayment()

    {

        System.out.println

                   ("Online Payment through Netbanking, card or Paytm");

    }

    @Override

    public void doDelivery()

    {

        System.out.println

                    ("Ship the item through post to delivery address");

    }

}

class StoreOrder extends OrderProcessTemplate

{

    @Override

    public void doSelect()

    {

        System.out.println("Customer chooses the item from shelf.");

    }

    @Override

    public void doPayment()

    {

        System.out.println("Pays at counter through cash/POS");

    }

    @Override

    public void doDelivery()

    {

        System.out.println("Item delivered to in delivery counter.");

    }

}

class TemplateMethodPatternClient

{

    public static void main(String[] args)

    {

        OrderProcessTemplate netOrder = new NetOrder();

        netOrder.processOrder(true);

        System.out.println();

        OrderProcessTemplate storeOrder = new StoreOrder();

        storeOrder.processOrder(true);

    }

}

פלט:

Item added to online shopping cart

Get gift wrap preference

Get delivery address.

Online Payment through Netbanking, card or Paytm

Gift wrap successful

Ship the item through post to delivery address

Customer chooses the item from shelf.

Pays at counter through cash/POS

Gift wrap successful

Item delivered to in delivery counter.

## **דוגמאות נוספות**

1. מופיעה בקוד לעיל – רכישת מוצר אונליין או בחנות פיזית – יש הבדל בתהליך אבל למעשה בשני הסוגים מתבצעת בחירה, תשלום, ומשלוח.
2. קוד של משחק – בכל משחק יש את שלב ההתחלה, מצב הלוח, שחקן או שחקנים, תזוזות שחקן, צבירת נקודות, זיהוי פסילה וכדו'. המטרות בכל משחק הן כמובן שונות ואופי השגתן שונה. ניתן להגדיר את השלבים של כל משחק ולממש בהתאם רק את מה שמשתנה בין המשחקים.

## **יתרונות**

1. מאפשרים למממשים לדרוס רק חלקים מסוימים מהאלגוריתם, והם נעשים פחות מושפעים מחלקים אחרים באלגוריתם.
2. ניתן לשים את הקוד שחוזר על עצמו במחלקה אחת במקום להכפיל אותו.

## **חסרונות**

1. מימושים מסוימים יצטרכו להגביל את עצמם לפי התבנית, שלא תמיד תתאים להכל.
2. ככל שיש יותר צעדים באלגוריתם, נעשה קשה יותר לנהל את ה template method.
3. עלולים להפר את Liskov Substitution Principle בזה שמממשים קוד ברירת מחדל לצעד מסוים של האלגוריתם.

# **Visitor (מבקר)**

## **תיאור במשפט קצר**

תבנית המאפשרת להפריד אלגוריתם מהאובייקטים עליהם הוא פועל.

## **הבעיה התכנותית**

מעוניינים להוסיף פונקציונליות למחלקה, בלי לשנות אותה.

## **דוגמא**

קוד שמציג נתונים גיאוגרפיים של שטח מסוים. ישנם נתונים שונים שהוא מציג: עיר, ארגון, אתר תיירות, וכו'. כולם מיוצגים באמצעות מחלקת Node בסיסית ומממשים את מה שמתאים להם באופן פרטני. יום אחד עולה דרישה לייצא את הנתונים הללו לקובץ Xml או Json. לכאורה הפשוט ביותר – להוסיף לכל מחלקה את הפונקציה ExoprtToXML ולממש לכל אחד בהתאם, באופן רקורסיבי או לא – לפי הסוג שלו. הבעיה – לא מעונינים לפתוח את הקוד של המחלקות בכל פעם שמוסיפים סוג ייצוא חדש.

ובנוסף – מבחינה לוגית, קוד של ייצוא של המחלקה לא קשור למהות של המחלקה Node? ייצוא זה נושא נפרד ממהות מחלקת Node.

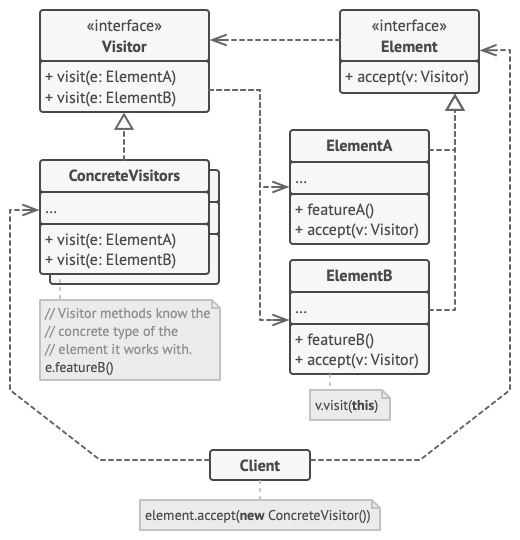
## **הפתרון**

מגדירים אובייקט נפרד שמיישם את הפונקציונליות החדשה של המחלקה. זהו ה Visitor. במחלקת ה visitor מיישמים פונקציה visit שמקבלת אובייקט, פונקציה אחת לכל סוג של אובייקט מעץ ההורשה שאותו מרחיבים.

במחלקות שאותן מרחיבים מגדירים פונקציה accept שמקבלת אובייקט מסוג ה Visitor, ושולחים לה את ה this.

אם יתכן שנרצה לממש מספר Visitors – אחד לייצוא ל XML, אחד לייצוא ל JSON וכו', אפשר לייצר מחלקת בסיס ל Visitor, ממנה יירשו כל Visitors שהם.

## **UML**



## **דוגמת קוד**

**namespace** **Wikipedia**;

**public** **class** **ExpressionPrintingVisitor**

{

**public** **void** PrintLiteral(Literal literal)

{

Console.WriteLine(literal.Value);

}

**public** **void** PrintAddition(Addition addition)

{

double leftValue = addition.Left.GetValue();

double rightValue = addition.Right.GetValue();

var sum = addition.GetValue();

Console.WriteLine("{0} + {1} = {2}", leftValue, rightValue, sum);

}

}

**public** **abstract** **class** **Expression**

{

**public** **abstract** **void** Accept(ExpressionPrintingVisitor v);

**public** **abstract** double GetValue();

}

**public** **class** **Literal** : Expression

{

**public** double Value { **get**; **set**; }

**public** Literal(double **value**)

{

**this**.Value = **value**;

}

**public** **override** **void** Accept(ExpressionPrintingVisitor v)

{

v.PrintLiteral(**this**);

}

**public** **override** double GetValue()

{

**return** Value;

}

}

**public** **class** **Addition** : Expression

{

**public** Expression Left { **get**; **set**; }

**public** Expression Right { **get**; **set**; }

**public** Addition(Expression left, Expression right)

{

Left = left;

Right = right;

}

**public** **override** **void** Accept(ExpressionPrintingVisitor v)

{

Left.Accept(v);

Right.Accept(v);

v.PrintAddition(**this**);

}

**public** **override** double GetValue()

{

**return** Left.GetValue() + Right.GetValue();

}

}

**public** **static** **class** **Program**

{

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

{

*// Emulate 1 + 2 + 3*

var e = **new** Addition(

**new** Addition(

**new** Literal(1),

**new** Literal(2)

),

**new** Literal(3)

);

var printingVisitor = **new** ExpressionPrintingVisitor();

e.Accept(printingVisitor);

}

}

## **דוגמאות נוספות**

## **יתרונות**

1. Open\Closed Principle – אפשר להוסיף פונקציונליות למחלקה בלי להצטרך לשנות את הקוד שלה.
2. Single Responsibility Principle – כל Visitor מרכז את האחריות לפעולה מסוימת בתוכו.
3. פעולות שמשותפות לעבודה של ה visitor על האובייקטים השונים באותה מחלקה, יכולות להיות משותפות לכולם באותו Visitor.
4. ל visitor אין גישה למאפיינים Private של המחלקה.

## **חסרונות**

1. בכל פעם שמוסיפים מחלקה או מורידים אותה מעץ המחלקות, צריך לעדכן את ה visitors.
2. ל visitor אין גישה למאפיינים Private של המחלקה.