

# The Java Design Patterns

## (Udemy Course- Jason Fedin)

DP הם פתרונות הנדסיים נפוצים לבניות הנדרסיות נפוצות. פתרונות הכל טוביים לבניה, שנבדקו הרבה פעמים. לא צריך לחשב על פתרון חדש לבניה זה בזבוז זמן. בתכנות זה לא רק לכתוב קוד וזהו, צריך לבדוק שהוא יעבוד תמיד, שאתה לא חוזר על עצמו, זה נראה טוב וקל להבנה, והתחזוקה שלו טובה (אם הצורך לשנות משהו זאת לא תהיה בעיה). אלו דברים ששימושם ב DP יכול לבוא ולעזר בהם. DP מוכרים ולכן אפשר לתקשר עם השותף שלך ובמוקם להסביר לו פתרון לבניה אתה פשוט אומר לו את שם הפתרן. שימוש ב DP יהפוך אותך למתכנת יותר טוב.

פתרונות מתחלקיים לשלווש סוגים:

1. Creational - נוגע לתהליכי ייצור האובייקט.
2. Structural - להתמודד עם קומפוזיציה של מחלקות ואובייקטים.
3. Behavioral - לאפיין את הדרכים שבהם מחלקות ואובייקטים מתקשרים ומעבירים אחריות אחד לשני.

לכל סוג יש גם תת סוג: פתרן לאובייקטים (קומפוזיציה). הקשרים נוצרים בזמן ריצה) או למחלקות (הורשה. הקשרים הם בזמן קומפונציה).

בעולם התוכנה בדרך כלל כתובים ישר את המחלקות השדות והmethodות ורוק אז חשבים על הקשר ביניהם. גישה יותר טובה תהיה להתבסס על DP ולהבין לפני שכותבים את הכל גם את הקישורויות שאנו רוצים ואת האחריות של כל קומפוננטה. DP יכול לעזור לא רק לבניה הנדרסית מסויימת אלא לכל עיצוב המערכת.

לא צריך לחפש בנחישות ובכח איפה למשר DP אלא פשוט לזכור שם נתקלים לבניה מסויימת יש לה פתרון.

עיצוב תכנוני גרוע יהפוך את תחזוקת המערכת לקשה: יהיה קשה למצאו באגים ולהוסיף פיצרים. מאוד חשוב לחשב על הקישורויות בין האובייקטים ומחלקות עוד לפני מימושם.

העקרונות של DP יכולים לעזור גם לבניות אחרות, ולהנגיש יותר לדפוסי חישיבה של מתכנת טוב. DP מתקשר באופן ישיר להמון מושגים מעולם ה OOP כמו: אינקפסולציה, הורשה, אבסטרקציה, שיכוף קוד, גמישות לשינויים, Cohesion, - ו- Coupling.

Coupling- מידת תלות הדדיות בין מחלקות ומתחודות (עד כמה הם מחוברים ותלויים אחד בשני). ככל שהזה פחות מעיד על פחתת תלות זה טוב לנו.

Cohesion- מתייחס למה מחלוקת מתחודה יכולים לעשות (האחריות שלה). ככל שהזה יותר זה אומר שהדבר יותר מפוקס זה יתיר טוב לנו.

אסטרטגיות נפוצות:

שימוש באבסטרקציה.

עדיפות לאgregation במקום הורשה (אגרגציה- הקשר בין מטוס לשדה תעופה; מטוס יכול להתקיים בלבד ואין קשר בין המטוסים. קומפוזיציה- הקשר בין גלגל למטוס; גלגל לא יכול להתקיים בלבד מטוס).

שימוש אינקפסולציה לדברים שימושיים הרבה. מיעוט השימוש של הורשה היררכית.

"ריח (מסריך) של עיצוב"- מחלוקת שמצוינות על הפרה של עקרונות עיצוב; שהוא שלא מרגיש טוב מבחינה עיצובית.

דוגמאות:

הרבבה מחלוקת עם אותו שכפול קוד. עדיף להשתמש בהורשה.  
יותר מדי קומפוננטות ציבוריות. נוגד את עיקנון האינקפסולציה.  
מחלקה ענקית עם יותר מדי אחריות.  
DP גורם לכל הרichות האלה להיעלם.

## עקרונות מרכזים ב- OOD:

### Programming to an interface, not an implementation . 1

השימוש ב Interface או Abstract Class תורם לי במספר דברים: הפחיתה Coupling, התעלמות מהדברים הקטנים, קוד גנרי, ופולימורפיזם בזמן ריצה.

#### השווי בין Interface לבין Abstract Class :

משתנים ב Interface הם public static final. ב Abstract Class אין הגבלות.  
מתוודות ב Interface הם public or public static. ב Abstract Class אין הגבלות.  
למתוודות ב Interface אין. ב Abstract Class אין הגבלות.  
Abstract Class יכול לספק מימוש של Interface אבל ההפר לא נכון (כי חיברים מימוש).  
דרך Interface אפשר לחבר בין מחלקות שאין בהם שום קשר (דוגמה: יש המון מימוש  
ל comparable).

דברים משותפים:

אפשר לספק התנהלות של מחלקות מבלית לדעת מי מ ממש אותה.  
אפשר להנות מהפירוט של הורשה מרובה (דרך עקיפה כי אין בגאואה).  
בשניהם אי אפשר ליצור אובייקט כלל שליהם.  
אין יותר טוב או פחות טוב. זה תלוי סיטואציה.

## Composition over Inheritance .2

קומפוזיציה- הקישוריות בין מחלקה ש"יש לה" אובייקט בתוכה. למחלקה יש שליטה מוחלטת על האובייקט. לדוגמה חדר לא יכול להתקיים בלי בניין או בית. נעשה בכך זה שימוש בהמון DP.

אגרגציה- ההבדל בין לבן קומפוזיציה הוא שכן אובייקט יכול להתקיים מבלי המחלקה. דוגמא לכך תהיה סטודנט וכיתה, סטודנט יכול להתקיים בלי כיתה.

איןקספולציה- בעצם מתעסק בהסתדרת אובייקטים שיכולים לפגוע בלקוח פוטנציאלי, בכונה או לא.

### מה העדיפיות:

זה גורם לכך שכל מחלקה תקיים את עקרון האינקספולציה ושתתמקד בדבר אחד. המחלקות לא יגדלו ויישארו קטנות ככל שאפשר.

הורשה שוברת איןקספולציה כי המחלקה הירושת תלויה במחלקה הראשית. הורשה היא מאוד coupled וקומפוזיציה פחותה. סביר להניח שמחלקה ירושת תישבר את נשנה משחו במחלקה האב.

אין בג'אווה הורשה מרובה אבל קומפוזיציה יכולה לעקוף את המנגנון זהה. רוב ה DP משתמשות בקומפוזיציה.

בקומפוזיציה יותר קל לנצל טסטים. ביצירת mock בהורשה תctrיך לעשות לשני המחלקות במקום אחד מהם. יותר קל לעשות unit-tests ולעבד ב TDD בקומפוזיציה. מבחינת "שימוש חוזר" הרבה יותר קל להשתמש בקומפוזיציה (יותר גמיש וניתן להרחבה). למחרות שנייהם תורמים לשימוש חוזר. אל תחשוף ישר הורשה כשתרצה להשתמש בעקרון זהה. בקומפוזיציה אפשר לעשות שימוש חוזר במחלקות final כי הם לא ניתנות להורשה.

## Delegation .3

הكونספט הוא בעצם שמחלקה מעבירה חלק מהאחריות למחלקה אחרת ומשתמשת במתודות שלה בלי לדעת מה הם עושים בדיק אבל עם ההנחה שהכל בסדר.

בעצם יחס בין אובייקטים שאובייקט אחד מעביר מתודות לאובייקט אחר. זהה דוגמא חזקה לקומפוזיציה ששוב מוכיחה שאפשר להחליף הורשה בקומפוזיציה. לרוב זה יותר טוב מההורשה, אתה מעביר מתודות ספציפיות ולא את כלם, גמישות זמן ריצה (היכולת לשנות את התוכנית בזמן ריצה באופן מהיר דרך שינוי קוד או Data).

دلגציה טובה רק אם היא מפשטת יותר מאשר שהיא מסבכת.

دلגציה זה שימוש חוזר במתודות שהוא בעצם קומפוזיציה. מה שבעצם שונה הוא שבקומפוזיציה חלק מהמתודות של האובייקט

יכולים להיות private או לא מיועדים בדיק לשימוש חוזר מדויק (סוג של Override).

בקומפוזיציה האובייקט לא יכול להתקיים בלי המעטפת ובdlgציה לא כך הדבר.

דוגמא פשוטה:

```
class Printer {
    // the "delegator"
    RealPrinter p = new RealPrinter();

    // create the delegate
    void print()
    {
        p.print(); // delegation
    }
}

public class Tester {

    // To the outside world it looks like Printer actually prints.
    public static void main(String[] args)
    {
        Printer printer = new Printer();
        printer.print();
    }
}
```

## Single Responsibility .4

הweeney של מחלקה יהיה רק חישבות\עבודה אחת ו声称 המתודות והשדות שלה נועדו לאותה מטרה. רק אם יהיה צורך לשנות משהו בעבודה יהיה צורך לשנות רק את המחלקה האחת הזאת. מתייחס לעקרון של cohesion - Coupling.

דוגמא פשוטה: במחלקה של Employee יש מתודה שמחשבת אם מגע לעבוד העלהה במשכורת. זה לא תפקידו של העובד אלא של ה HR. לכן נמחק את המתודה מהמחלקה ונשים אותה במחלקה חדשה שתיקרא למשל HRPromotion והמתודה מקבל עובד כפרמטר.

## The Open Closed .5

עיקרונו שבו מחלקות ומתודות צרכות להיות פתוחות להרחבה מבחינה פונקציונאלית וסגורות עבור שינויים. מחלקה אמורה להיות ניתנת להרחבה מבלי לשנות את המחלקה עצמה. תורם loose coupling.

דוגמא פשוטה:

```
class Rectangle {
    public double length;
    public double width;
}

class AreaCalculator {
    public double calculateRectangleArea(Rectangle rectangle) {
        return rectangle.length * rectangle.width;
    }

    public double calculateCircleArea(Circle circle) {
        return (22/7) * circle.radius * circle.radius;
    }
}

class Circle {
    public double radius;
}
```

יש לנו מחלקה לחישוב שטחים של צורות. הבעה היא שם נוסף למערכת עוד צורות נצטרך כל פעם לשנות ואו להוסיף מתודות לאותה מחלקה.

פתרון לבעה: יצירת Interface Shape שתהייה בה את המתודות של חישוב שטח. ואז כל מחלקה של צורה תמשח אותו אינטראפיס ואותה המתודה. ואז יהיה אפשר לשנות את המחלקה שלעיל כדלקמן:

```
class AreaCalculator {
    public double calculateShapeArea(Shape shape) {
        return shape.calculateArea();
    }
}
```

בכך השגנו שכל צורה חדשה שתתוסף למערכת לא נצטרך לגעת במחלקה הזאת. כמו כן, מדוברפה על דוגמיה כי העברינו אחריות של חישוב השטח של צורה למחלקה אחרת. למעשה כאן המחלקה הופכת למיותרת כי יש לי את יכולת של חישוב השטח גם בנסיבות עצם אבל זה רק דוגמא שמנסה להמחיש את העיקרונו.

## Liskov Substitution .6

הweeneyון הוא שאובייקט של מחלקה אב יכול להתחלף עם אובייקט של מחלקה יורשת,igli ששם דבר ישר. מציר שמתודות אצל שני האובייקטים יעבדו כהכלא באוטה מידה. בעצם זה פשוט העקרונות של הורשה ופולימורפיזם. מתודות override במחלקה היורשת חייבות לקבל את אותן פתרומים כמו המקור. ערך החזר של אותן מתודות יפעלו על אותן חוקים בשני המקורים, אלא אם כן צריך לעשות משהו ספציפי למחלקה יורשת ספציפית. העיקנון מאד קשור לעקרונות **Interface Segregation** - **Single Responsibility** והוא בעצם הרחבה של **The Open Closed**.

דוגמא פשוטה: מכונית יורשת מורכב.

## Interface Segregation .7

לקוחות לא חיבים להיות תלויים באינטראפיזם שהם לא משתמשים בהם. לקוחות לא צריכים למש אינטראפיזם אם הם לא מתכוונים להשתמש במתודות אלו. בעצם מדובר על Cohesion באינטראפיזם. זה קורה כאשר באינטראפיזם יש יותר ממתודה אחת יותר אחריות אחת אז הוא לא צריך את כל המתודות (זה לא בהכרח אומר שלכל אינטראפיזם אמרו להיות מוגדרות אחת). המטרה כאן היא היכולת לעמוד בשינויים דרך פיצול אחריות לכמה מחלקות.

דוגמא פשוטה: Moveable Interface שבתוכו המתודה של לטלוק. אם נרצה ליצור מחלקה של קורקינט שמשממת את אותו אינטראפיזם תהיה לנו בעיה כי קורקינט לא מטלוקים.

## Dependency Inversion .8

הweeneyון הוא שתלות בין אובייקטים תהיה אבסטרקטית ולא רגילה דרך מחלקות רגילות (עדיף כמה שפחות מזו). מחלקה "האב" לא צריכה להיות במחלקה "היורשת". המחלקה "היורשת" ניתנת לגישה במחלקה "האב".

מציר את עקרון ה- **Programming to an interface, not an implementation** פשוט כאן באמירה יותר חזקה לגבי האבסטרקטיה.

```
class PasswordReminder {
    private DBConnectionInterface dbConnection;
    public PasswordReminder(MySqlConnection dbConnection)
    {
        this.dbConnection = dbConnection;
    }
}
```

דוגמא פשוטה:

הfonקציה הזאת תליה ממש במימוש הספציפי של המסד נתונים שלו. הוא משוחה בה השתנה אני אctrיך לשנות גם פה.

מה שעדיף לעשות זה אינטראפיזם של תקשורת עם מסד נתונים עם מימוש שיש בתמונה. כך אין תלות בין המחלקה שלנו למימוש ספציפי של מסד נתונים ויש תלות אבסטרקטית בין במימוש הספציפי לאינטראפיזם. מציר את עקרון ה- **The Open Closed**.

## Dependency Injection .9

שיטה שבה אובייקט מספק תלות של אובייקט אחר, כך שהיה אפשר לשנות את התלות (אובייקט אחר) בלי לשנות את הקוד.  
הולך יד ביד עם העקרון Dependency Inversion. יש שלוש סוגי של "הזרקה": דרך הבנאי, דרך setter, או דרך מימוש אינטראפיס עם פונקציית setter.  
המטרה כאן להעלים תלויות חזקות בין אובייקטים (שימוש בnew).

```
public class Client {  
    // Internal reference to the service  
    private ExampleService service;  
  
    Client(){  
        // specify a specific implementation  
        service = new ExampleService();  
    }  
}
```

זו דוגמא של תלות חזקה בין אובייקטים.  
החסרון הוא שאם השם service משתנה אז גם הclient משתנה. הקוד לא גמיש לשינויים ולא תחזוקתי. יש בעיה גם מבחינת טסטים כי אין צורך לבדוק את שתי המחלקות.

## UML- unified modeling language

שיטה לייצוג דיאגרמות שיפכו את המרכיבת של התוכנה להרבה יותר קלים להבנה. יש לה קשור ישר לOOD. אנחנו רק נלמד את הבסיס בשביל שיהיה לנו יותר קל להבין את ה-OP.

נתעסוק בעיקר ב class diagram. מאוד קל לשימוש אפילו גם לאנשים שלא מתכונים.

בהתחלתו צריך לזהות את הישויות המערכת ו怎 על הקשרים ביניהם. ורק אחרי זה להתחיל עם הדיאגרמות.

### :class diagram

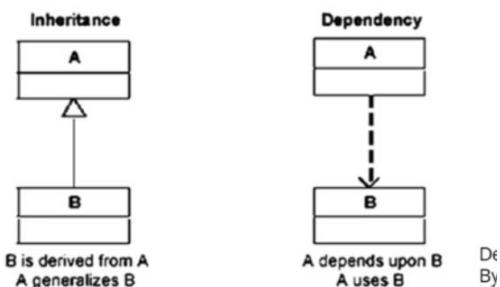
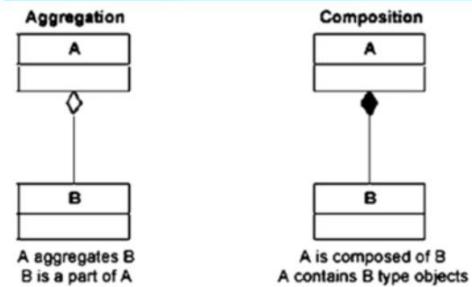
הדיagramה צריכה לתאר את המתוודות והשיטות של המחלקה, את הקשר בין מחלקות אחרות. השם של הדיאגרמות צריך להיות מובן וקשר למערכת כולה. בדיאגרמה נשים רק שדות שבאמת נחוצים. עדיף להוסיף הסבר לכל דיאגרמה.

מחלקה תהיה בצורה של מלבן. אם נרצה רק לתאר את הדברים מלמעלה אז נשים רק את שם המחלקה זהה. אם נרצה להוסיף קצת יותר נסיף מתוודות עם סוג הנגישות שלהם (#=public=+, protected=-, private=-). אם נרצה להוסיף עוד מידע נוסף את השדות, נגישותם, וטיפוס ערך ההחזר שלהם. רמה מעלה שלא חייבים לעשות זה להוסיף הסבר קצר על המחלקה.

אפשר להוסיף לקשרים מספרים שמתארים קשר מסוימי, כמו: אחד לרבים, רבים לאחד, רבים לרבים, או סתם מספרים ספציפיים.

דוגמאות יהיו לכל פטרן בנפרד.

## Class Relationships



## Creational Design Patterns

תוכניות לא אמורה להיות תלויות בארכ אובייקטים נוצרים ומארגנים.  
מחלקות רגילות גורמות לקוד להיות שבריר ופחות coupling.  
הפטרים בקטgorיה זו מספקים דרך לייצר אובייקטים (להמעיט את השימוש ב new) ומנסים להפוך את הדברים ליותר אבסטרקטיים.

### 1. Factory Method

התפקיד של המחלקה הוא לייצר את האובייקט שברצוננו לייצר.

מתי משתמש בפתרן?

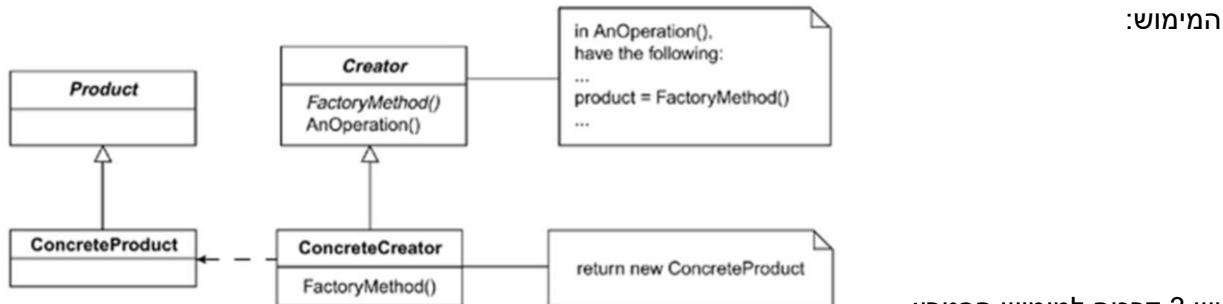
כשרצה להחזיק מופע של תת מחלקה אחרת אבל אנחנו לא יודעים איזה.

כשנרצה לייצר אובייקט מבלי לחושף ללקוח את הלוגיקה של הייצור.

כשנרצה להפעיל את עקרון האינקפסולציה על מופע רגל.

כשנרצה להפריד בין הלוגיקה של המחלקה לייצור שלה, ההפרדה לא רק בתוך המחלקה עצמה אלא גם בלוגיקה חיצונית.

הופך את הקוד ליותר חסין, פחותים coupled, וקל להרחבה. אבל הופך את קוד לפחות קרייא.



מחלקה שאחראית על הייצור תהיה אבסטרקטית. תהיה פונקציית ייצור ומחלקה "יצור לכל מימוש של מחלקה" "ירשת" (שזה באסה).

דוגמה קוד: נגיד שיש לנו אינטראפיס של Shape עם מתודה של draw () ולמחלקה המתוועדים: Circle, Square, Rectangle. כתת ניצור את המחלקה כמו בתמונה.

```

public abstract class AbstractShapeFactory {
    protected abstract Shape factoryMethod();
    public Shape getShape() { return factoryMethod(); }
}
  
```

מחלקה זו תהיה שלוש מימושים לכל צורה. וכל אחת תחזיר במתודה של הייצור אובייקט חדש ספציפי של המחלקה (ممוש new).

```

Shape shape1 = new CircleFactory().getShape();
shape1.draw();
  
```

ויר הדברים נראה דרכ הלקוח:

- **מימוש Concrete Creator**

כאן יש רק מחלקה יוצר אחת והיא לא תהיה אבסטרקטית.  
ニイズルアトアオビイキツハナクノンレヒツルヘルメトリムシイタヌロボンキツハ(マホロツトオヌエヌ  
שמייצג את שם המחלקה הספציפית) ואז נחזיר אובייקט חדש של המחלקה.

```
ShapeFactory shapeFactory = new ShapeFactory();
Shape shape1 = shapeFactory.getShape(shapeType: "CIRCLE");
shape1.draw();
```

וכך זה נראה בצד לקוח:

- **מימוש Static Method Creator**

כמו בConcrete Creator רק שהמתודה של הייצור הינה static. ובכך אנחנו חוסכים את הייצור של מחלקה הייצור עצמה. החיסרונו הוא שאנו יכול לעשות לה הרחבה ואי אפשר לשנות את התנהלות הפונקציה.

## Abstract Factory Method .2

ספק ממשק לייצור, אבסטרקטי, משפחות של אובייקטים שקשורים או תלויים אחד בשני, מבלי לדעת את המחלקה הספציפית. סוג של **factory of factories**.  
,Factory Methods המתוודות של ה Abstract Factory ממומנשות על ידי **Factory Method** "ירוש" מפטן זה.  
הו מספק אינקפסולציה על כל היצרנים.

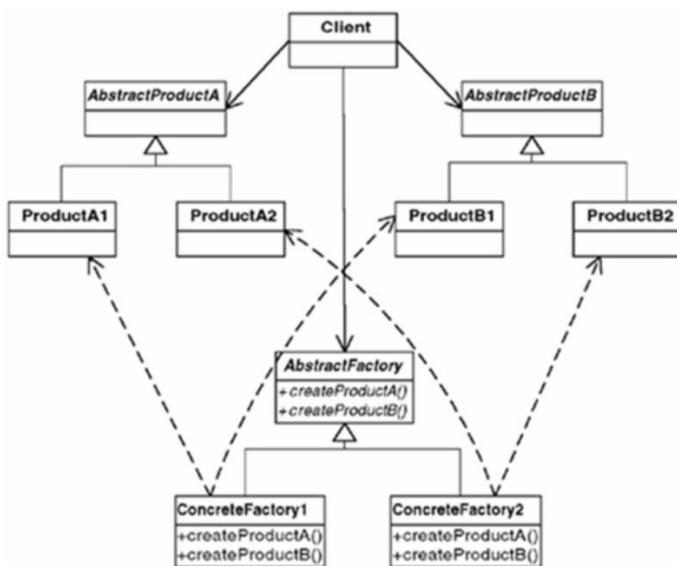
מתי נשתמש בפטן?  
הרבה מהמקרים נרצה להשתמש בו באופן סיבוט כמו ה **Factory Method**, רק שכן יש לנו משפחה של מוצרים.  
כשמערכת צריכה להיות תלויה באיך מוצרים נוצרים, מאוכלים, ומוצגים.  
כשרה להתמודד עם כמה יצרנים במערכת.  
כשיש לנו משפחות של מוצרים והשימוש בכל משפחה תלוי בסביבות שונות.  
הבעיה שકצת קשה להוסיף משפחות של מוצרים חדשים, צריך ממש לכת ולהרחיב את המחלקה.

הקשר בין לבין **Factory Method** שני הפטנים עושים אינקפסולציה לייצור אובייקטים. מפחיתים את ה coupling במערכת ואת התלות בימוש.  
Abstract Factory עשו דלגציה, מעביר את האחוריות שלו אל **factory** ספציפי לייצור האובייקט, והוא עשו זאת דרך קומפוזיציה.

בדרך כלל במערכת לכל משפחא צריך להיות **factory** אחד. מימוש טוב יהיה באמצעות **Singleton**.

בגלל שקשה להוסיף מוצרים חדשים מימוש יותר גמיש יהיה הוספה פרמטרים למתחודות הייצרה כמו שעשינו ב **Factory Method**.

אפשרה לייצר מחלקה **FactoryProducer** שאחראית על איזה **factory** לייצור באמצעות **static**.



דוגמת קוד: נניח שיש לנו שתי אינטראפיזים של Color ו- Shape, ולהם מימושים שונים.

```
public abstract class AbstractFactory {  
    abstract Color getColor(String color);  
    abstract Shape getShape(String shape);}
```

יש לנו את המחלקה factories שבתמונה ושתי שיטות המשמשים אותה, כל אחת עבור אינטראפי אחד. המחלקות כמעט זהות לממה. Factory Method שעשינו

**שימוש לב:** דוגמא זו נוגדת את עקרון ה Segregation Interface כדי לדוגמא למודול אחד. מחייב למש את המethode של getColor למגוון שאינו בזיהו.

```
public class FactoryProducer {  
    public static AbstractFactory getFactory(String choice) {  
        if(choice.equalsIgnoreCase("SHAPE")){  
            return new ShapeFactory();  
        }else if(choice.equalsIgnoreCase("COLOR")){  
            return new ColorFactory();  
        }  
        return null;}}
```

מומלץ לייצר מחלקה עם מתודת סטטית שמחייבת איזה factory ליצור.

```
AbstractFactory shapeFactory = FactoryProducer.  
    getFactory(choice: "SHAPE");  
Shape shape1 = shapeFactory.getShape("CIRCLE");  
shape1.draw();
```

כך הלקוח רואה את הדברים:

## Singleton .3

mbitich shel machlaka ha-madobera yihya mofe' yachid u-mospek gisha globlit al-ya.

Dogmato: manhal ha-kbutzim, logim. Zrik mofe' yachid ba-me'uretka.

Yesh machlakot shaiyib shiyya la-hem mofe' achad ci am la-2 mofe'im yekolim lagut ba-otno shetach zikron u-lagrom l-kashel lo-ya.

Itrono: Singletion uvesha ain-kafsaltsia ul mofe'u ma shmekna lo shelita bludiyat ul mati u-ayr yastamsho bo.

Singletion gamish le-kmota ha-mofe'im she-o yekol la-harushot shiyakim. Am la-dogma nrecha yoter mofe' achad zat la-tahia b'ya.

Chosronot: Singletion yekol lagrom libu'ot mabchinat bennit tsutim. Kasha le-ukob achor ha-mofe' cshmevrim otu bein ha-matodot. Shymosh yatraha b'mofe'.

Mati nrecha la-hastmesh b'Dependency Injection b'mekom Singletion:

Cashchob lo-nu shatotona tahya tachat unit test. Pchot coupling.

Cshnrecha le-himnu mahmila static. Grom le-kod la-hiot kasha yotter la-bdika.

Cshis lo-nu talot la-yibba um grom chizoni camo mad natonim udif lo-nu "lezrik" at ha-talioth ha-nchuzot.

Mati nrecha la-hastmesh b'Singletion b'mekom Dependency Injection:

Cshbme'ret ish talioth shel ha-monu machlakot, u-shkbotot shel machlakot. Ba sotion natzetr.

Le-havir at ha-mofe' la-oruk kl ha-me'ret u-lken Singletion yihya ha-raba yotter pesot.

Dogma matzint la-za ha-ava logger ci cmut kl machlaka zricha otu u-la nrecha kl ha-zman le-zezik at ha-mofe' la-kil machlaka.

Sh 5 drachim lemash at ha-petran:

Ma shmoshotf l-kolm ha-ya sh:

Zrik CDI l-mano' malitzor mofe' chadsh mabchuz.

hamofe' zrik le-hiot private static.

Ha-matoda shme'zira at ha-mofe' tahya public static.

### Lazy evaluation .1

Gisha zo ha-ya la-thread safe. Yekol la-hiot ba race condition (caasher shni traddim roa'im

ba-otno zman shahmofe null u-achd miyizer pum rashaona u-mazkir at ha-avbikat u-hetrad ha-shni

miyizer shob u-mazkir at ha-avbikat). Gisha la-momlaza.

```
public class Singleton {  
    private static Singleton uniqueInstance = null;  
    private int data = 0;  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
}
```

## Synchronized .2

גישה זו היא thread safe . אבל לא אופטימלית. لكن משתמשים בגישה זו כ舍יביצועים זה פחות חשוב.

השינוי היחידי במימוש היה הפיכת המתודה getInstance ל synchronized .  
זה למה הביצועים כאן הם פחות טובים: יכולה להיות בעיה חמורה של צוואר בקבוק של טרדים, ככלומר כמה טרדים מחכים לטרד אחר (בגלל זה סינכרון צריך להיות כמו שיטור מינימלי).  
בנוסף, השימוש היחידי בסינכרון הוא רק בקריאה הראשונה למתודה. אחרי זה אנחנו לא באמת צריכים את הסינכרון. הוא מיותר והופך את הקוד לאיטי יותר.

## Double-checked locking principle .3

גישה זו היא thread safe . יותר טובה בBITS של גישה 2

השינויים:

נוסיף למופיע את מילת המפתח volatile אשר מבטיחה שכיש מספר טרדים הם ישתמשו במופיע באופן בטוח (מסנכרן את כל העותקים במתมอง של המשתנים בזיכרון הראשי).

בנוסף, המתודה של המופיע תשתנה לפחות בתמונה. וכך אנחנו בעצם פתרנו את כל בעיות BITS של גישה 2.  
הסינכרון הוא רק על מה שצורך וחוץ מזה קורה רק בקריאה הראשונה.

```

public static Singleton getInstance() {
    if (uniqueInstance == null) {
        synchronized (Singleton.class) {
            if (uniqueInstance == null) {
                uniqueInstance = new Singleton();
            }
        }
    }
    return uniqueInstance;
}

```

## Eager evaluation .4

גישה זו היא thread safe (ה JVM מבטיח לנו שהמופיע יבוצע לפני כל טרד שייגש אליו). אם המערכת תמיד יוצרת ומשתמשת בסינגלטן נעדיף את הגישה הזאת. אם המחלקה לא מצורכנית הרבה משאבים זו הגישה שונצחה. החיסרונו היחידי שאפלו אם אף אחד לא משתמש בו הוא יבצע. נוצר בטיענות המחלקה. הגישה הכי קלה.

```

public class Singleton {
    // create an instance of singleton in a static initializer, code is guaranteed to be thread safe
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        // we already got an instance so just return it
        return uniqueInstance;
    }
}

```

גישה זו היא **thread safe** . ביצועים גבויים. מבטיח שהמופיע נוצר רק אם משתמשים בו (שיעור גישה 4). הסינגלטון נוצר בעזרת מחלוקת סטטית פנימית. הגישה הסטנדרטיבית.

```
public class Singleton {  
    private int data = 0;  
    private Singleton() {}  
    private static class SingletonHelper {  
        // Nested class is referenced after getInstance() is called  
        private static final Singleton uniqueInstance = new Singleton();  
    }  
    public static Singleton getInstance() { return SingletonHelper.uniqueInstance; }  
}
```

בונוס:

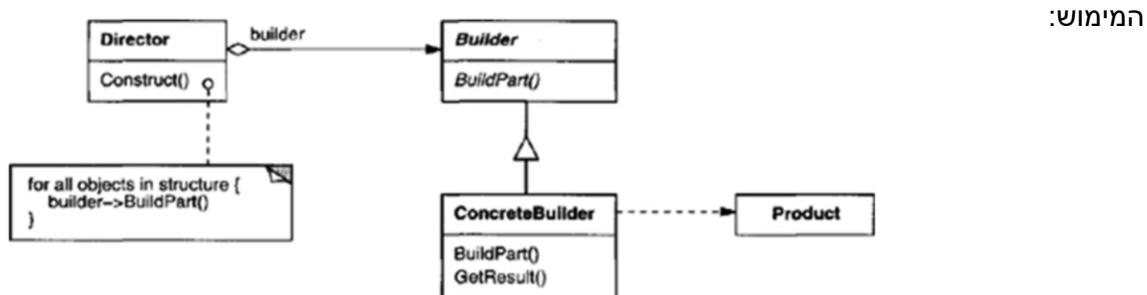
בדרכ כל נרצה שאך מחלוקת לא תוכל לעשות הורשה מסינגלטון ולכן הסינגלטון יהיה **.final class**. כמו כן, ריבוי סינגלטוניים במערכת עלול לייצר באגים מהסוג הגרוע ביותר. סינגלטון זה לא **driven-reality** לפיכך במערכת, כל מערכת, נגדיר סינגלטון יחיד.

## Builder .4

הבדיל בין בניית אובייקט מורכב לייצוג שלו. טוב כאשר יש סדר לבנייה עצמה ולא כל הרכיבים חיברים להתקיים. טוב כאשר אין תלות בין האובייקט לרכיבים שלו.

דוגמא: ייצור מחשב. יש סדר בהרכבתו וכן כל מחשב אפשר ליצור עם מעבד שונה. עוד דוגמא: בניית תוכנית לחופשה בדיסני ווארד. יש הרבה רכיבים לבנייה (בחירת מלון, סוג כרטיס כניסה, מקום לאוכל ועוד). לכן צריך עיצוב גמיש, כי לקוח אחד לא צריך בכלל מלון לדוגמא. יש המונ אילוצים.

לא נשתמש בפטון זה אם אותו אובייקט נועד להשתנות הרבה במערכת.



הIMPLEMENTATION:

דוגמא קוד:

```

public class Vehicle { //Product
    private LinkedList<String> parts;
    public Vehicle() {parts = new LinkedList<String>();}
    public void add(String part) { parts.addLast(part); }
}
  
```

ראשית יהיה לנו  
מחלקה רכב  
שהיא המוצר  
שנרצה ליצור.

לאחר מכן נרצה  
לייצר אינטראפ"יס  
של "בונה"  
הרכיבים" ונעשה  
לו שימושים לכל  
רכיב במערכת.

אפשר ליצור בונה  
של אופנוע והוא  
יהו שונה במבנה  
 מאשר מכונית.

```

interface BuilderInterface {
    void buildBody();
    void insertWheels();
    void addHeadlights();
    Vehicle getVehicle();
}

class CarBuilder implements BuilderInterface {
    private Vehicle product = new Vehicle();
    @Override public void buildBody() { product.add("This is a body of a car"); }
    @Override public void insertWheels() { product.add("4 wheels are added"); }
    @Override public void addHeadlights() { product.add("2 headlights are added"); }
    @Override public Vehicle getVehicle() {return product;}
}
  
```

\*\* הרכיבים בכל רכב מיוצגים כרשימה מקושרת לשם פשטוות ממובן.

```
public class Director {  
    BuilderInterface myBuilder;  
    public void construct(BuilderInterface builder) {  
        myBuilder = builder;  
        myBuilder.buildBody();  
        myBuilder.insertWheels();  
        myBuilder.addHeadlights();  
    }  
}
```

נשאר לבנות דירקטורי שאחרראי על ניהול הבנייה מבחינת סדר הרכיבים.

וכך בעצם הלוקו מייצר מכונית:

```
Director director = new Director();  
BuilderInterface carBuilder = new CarBuilder();  
director.construct(carBuilder);  
Vehicle car = carBuilder.getVehicle();
```

## Prototype .5

מתייחס ליצירת שכפול אובייקט תוך שמרה על ביצועים. מצין את סוג האובייקטים ליצירה באמצעות מופע אב טיפוס, ויציר אובייקטים חדשים על ידי העתקת אב טיפוס זה. נעשה בו שימוש כאשר יוצרת אובייקט היא יקרת ערך מבחינות זמן ומשאבים, וכשיש לי כבר אובייקט דומה מוקן. הפתרן עושים שימוש ב cloning אם רוצים העתקה רדודה (לא מעתק אובייקטים מורכבים בתוך המחלקה) או serialization כנדרש העתקה عمוקה. מה שמיוחד גם בפתרן זהה שהשלקוח לא חייב לדעת מה המחלקה הספציפית שירצה ליצור. שוב חזרנו לאבסטרקציה. השתמש בפתרן ככלא יהיה לנו אכפת איך לייצר את אותו אובייקט. כשהאובייקט אחראי על היררכיות של מחלקות אחרות וקשה לייצר.

יתרונות:

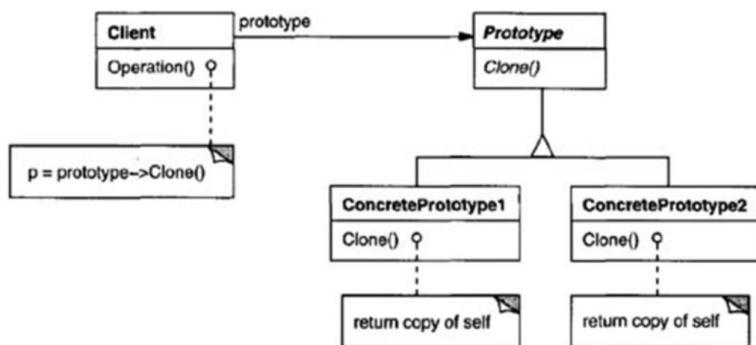
החבאת המורכבות של הלוגיקה של הייצור מחלקות. אינקסולציה.  
 "יצור אובייקט ספציפי מבלי לדעת מה הוא.  
 לפעמים העתקת אובייקט יותר קל מהיצור שלו מחדש.

חסרונות:

כל שימוש או מחלקה יורשת חייבת למש את עקרון העתקה.  
 לעיתים מימוש העתקה יכול להיות קשות. בעיקר אם במקרים מסוימים אסור להעתיק או אם יש קשר מעגלי בין האובייקטים.  
 באינטראפיס של cloneable ב java יש בעיות (לא חייבים להשתמש בו).

השימוש:

צריך למש את האינטראפיס `cloneable` ואז למש את המתודה `clone`.



```

abstract class Shape implements Cloneable {
    private String id;
    protected String type;
    abstract void draw();
    public String getType() { return type; }
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}

```

```

class Circle extends Shape {
    public Circle() { type = "Circle"; }
    @Override public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

```

```

public class ShapeCache {
    private static Hashtable<String, Shape> shapeMap = new Hashtable<?>();
    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }
    public static void loadCache() {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId(), circle);
    }
}

```

```

ShapeCache.loadCache();
Shape clonedShape = (Shape) ShapeCache.getShape(shapeld: "1");
System.out.println("Shape : " + clonedShape.getType());
//Shape: Circle           //this is a new object. a copy

```

#### דוגמת קוד:

תחליה נוצר מחלקה אבסטרקטית של צורה. נשים לב שהיא משתמשת אינטראפיס של cloneable למרות שיש בו בעיות.

את המתודה clone נמסח במחלקה האבסטרקטית כמו בתמונה ו록 אם צריך גם אצל המימושים הספציפיים של המחלקה.

הינה דוגמא פשוטה של שימוש ספציפי. אין () clone כי המחלקה פשוטה.

לשם פשטות הדוגמא נוצר מטען של צורות ספציפיות. בעצם זה מפה של id של הצורה והצורה עצמה.

ובתמונה الأخيرة אפשר לראות את צד הלקות.

אפשר לראות שהלקות לא יודע איזה צורה הוא בכלל מעתיק.

```

class Person implements Cloneable { // step 1
    private String name;
    private City city; // deep copy

    // no @override, means we are not overriding clone
    public Person clone() throws CloneNotSupportedException { // Step 2
        Person clonedObj = (Person) super.clone(); // step 3
        clonedObj.city = this.city.clone(); // Making deep copy of city
        return clonedObj;
    }
}

```

הינה עוד דוגמא:

הבעיות ב `:interface Cloneable`

אין בכלל את המתודה `clone` זה בעצם מפרק אינטראפייס. אנחנו בכל זאת צריכים למשמש אותו רק כדי להגיד ל JVM שהחלוקת שלנו יש לה את המתודה `clone`. אם אנחנו ממשים את המתודה לחלוקת יורשת נהיה ח'יבים ממש אותה גם לחלוקת האב כי אז השרשרת של `super.clone()`

`(Object.clone()` בעצם מבצע העתקה רדודה. צריך למשם את המתודה בכלחלוקת שМОוחזקת בתחום האובייקט שלנו בשביל העתקה عمוקה.

שדות `final` לא ניתנים לשינוי רק דרך הבנאי. אי אפשר לעשות פולימורפיזם למתודה: אם יהיה לי מערך של `Cloneable` זה נדמה אולי אפשר לróż על הכל ולעשות `clone` לכל תא בתורו אבל זה לא עובד כך. גם לאינטראפייס וגם ל-`Object` אין המתודה `clone`.

אחרי העתקה אם נשנה אובייקט אחד גם השני ישנה, אם מדובר בהעתקה רדודה כשייש באובייקט שדה מטיפוס לא פרימיטיבי.

אפשר במקום כל זה לעשות פשוט `copy` קונסטרוקטור או `serialization` (לכתוב ולקראן מקובץ). לעיתים יש להם יתרונות מאוד טובים. אבל לרוב עדין עדיף למשם את האינטראפייס שלויל.

## Structural Design Patterns

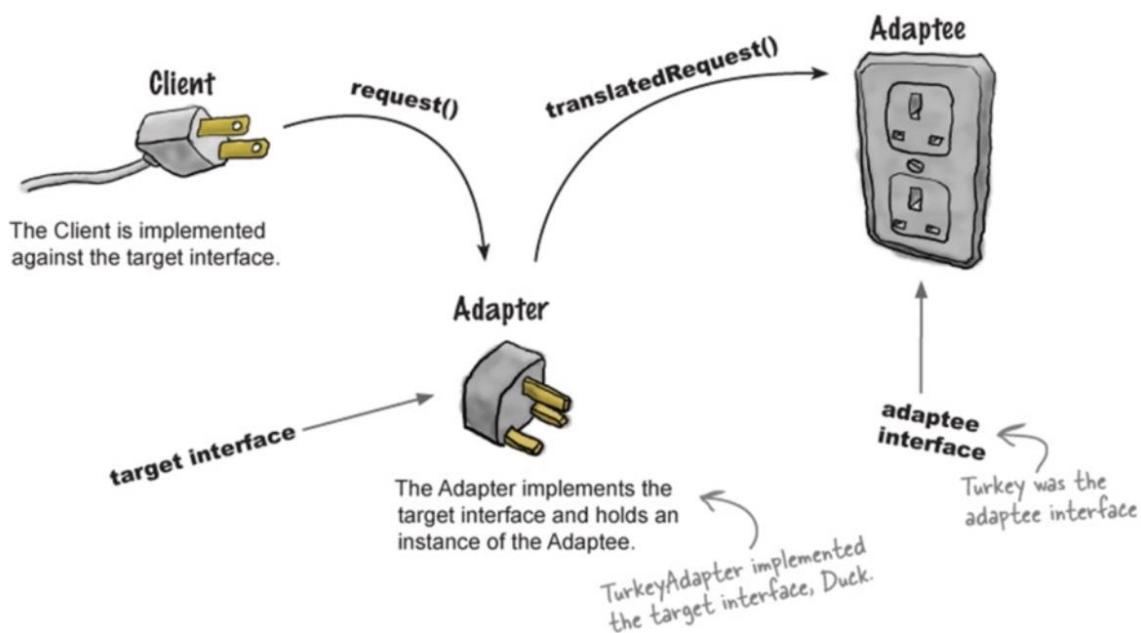
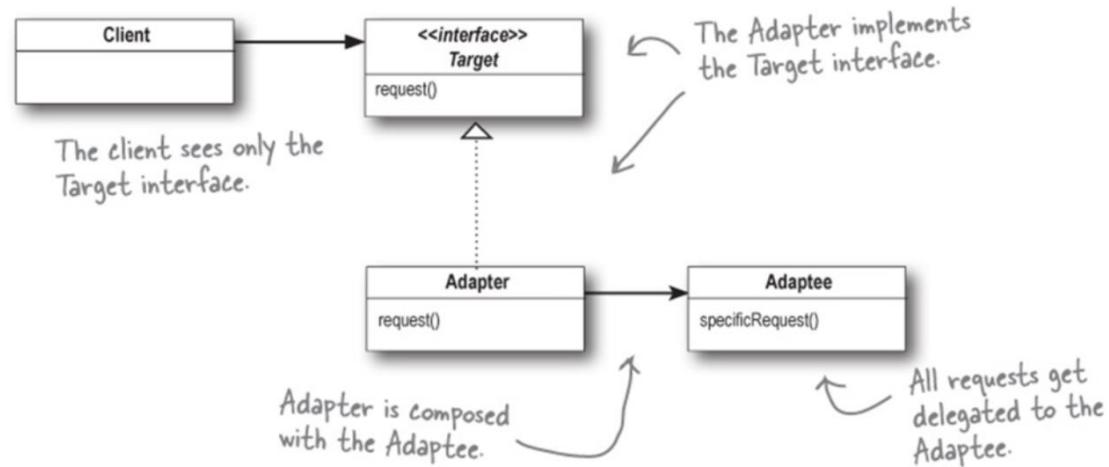
מתאר כיצד ניתן לשלב מחלקות אוbjectים לצירוף מבנים גדולים יותר. משתמש בירושה כדי ליצור ממשקים או יישומים. נעזרים בהורשה וקומפוזיציה.

### Adapter (Wrapper) .6

ממיר אינטראפיז של מחלקה אחת לאינטראפיז אחר שהליך מצפה אליו. מעין גשר בין שני אינטראפיזים שאין קשר ביניהם. הפטן נותן את האפשרות לשני המחלקות לעבוד ביחד למטרות שם לא יכולם. הלקוח לא צריך לשנות את הקוד אם ירצה לשנות אינטראפיז.

דוגמאות: מתאם מטענים, מילונית, Collection classes, `InputStreamReader`, `Object Adapter`.

#### Object Adapter



דוגמת קוד 1:

תחליה נממש אינטראפיס של ברוז עם מימוש ספציפי.

השלב הבא יהיה ליצור עוד אינטראפיס של תרגול הודה עם מימוש ספציפי.

המטרה שלנו כאן בעצם להמיר את ההודו לברוז.

למרות שאין קשר בין שני המחלקות.

מי שארחאי בעצם על ההמרה הוא Adapter-הוודו. שמש את הברוז ומקבל

```
interface Duck { //Target
    public void quack();
    public void fly();
}

class MallardDuck implements Duck {
    public void quack() { System.out.println("Quack"); }
    public void fly() { System.out.println("I am flying"); }
}
```

```
interface Turkey { //Adaptee
    public void gobble();
    public void fly();
}

class WildTurkey implements Turkey {
    public void gobble() { System.out.println("Gobble gobble"); }
    public void fly() { System.out.println("I am flying a short distance"); }
}
```

```
public class TurkeyAdapter implements Duck{
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey) { this.turkey = turkey; }
    public void quack() { turkey.gobble(); }
    public void fly() {
        for(int i = 0; i < 5; i++)
            turkey.fly();
    }
}
```

וכך זה נראה מצד הלוקוט:

```
MallardDuck duck = new MallardDuck();
WildTurkey turkey = new WildTurkey();
Duck turkeyAdapter = new TurkeyAdapter(turkey);
```

## דוגמת קוד 2:

```
class Rectangle {  
    public double length;  
    public double width;  
}  
  
class Triangle {  
    public double base;  
    public double height;  
    public Triangle(double b, double h)  
    {  
        base = b;  
        height = h;  
    }  
}
```

נתחיל עם ייצרת שתי צורות: מלבן ומשולש.

נניח שיש לנו אינטראפיס עם מתודה  
שמחשבת שטח רק של מלבן. כמו בתמונה  
השנייה.

```
public interface CalculatorInterface {  
    // target interface  
    public double getArea(Rectangle r);  
}  
  
class Calculator implements CalculatorInterface //Adaptee  
{  
    Rectangle rectangle;  
    public double getArea(Rectangle r) {  
        rectangle = r;  
        return rectangle.length * rectangle.width;  
    }  
}
```

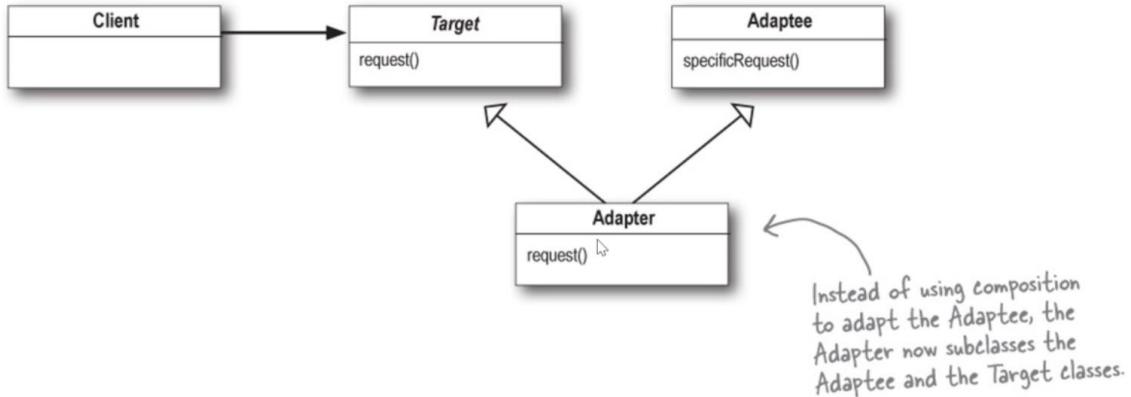
```
public class CalculatorAdapter implements CalculatorInterface {  
    Calculator calculator;  
    Triangle t;  
  
    public CalculatorAdapter(Triangle myTriangle) { t = myTriangle; }  
    public double getArea(Rectangle r) {  
        calculator = new Calculator();  
        Rectangle rectangle = new Rectangle();  
        rectangle.length = t.base;  
        rectangle.width = 0.5 * t.height;  
        return calculator.getArea(rectangle);  
    }  
}
```

הבעיה פה היא  
שאנחנו רוצים גם  
לחשב את השטח  
של משולש. לכן  
نبנה מתאם  
שיהפוך משולש  
למלבן עם  
התאמות הנחוצות  
וכך נחשב את שטח  
המשולש מבלי  
להרוס את  
האינטראפיס.

כך זה נראה מצג הלקוח:

```
Triangle t = new Triangle(b: 20, h: 10);  
CalculatorInterface calculatorAdapter = new CalculatorAdapter(t);  
System.out.println("Area of Triangle is: " +  
    calculatorAdapter.getArea(r: null));
```

## Class Adapter



הכעיה היחידה פה היא שבג'אווה אין ריבוי הורשה. ולכן נשאר עם האפשרות הראשונה שלמדנו. אבל אפשר להתחמק מזה דרך אינטראפיזם.

נראה דוגמא עם הורשה אחת:

```
interface IntegerValueInterface {  
    public int getInteger();  
}  
  
class IntegerValue implements IntegerValueInterface {  
    @Override  
    public int getInteger() {  
        return 5;  
    }  
}  
  
class ClassAdapter extends IntegerValue {  
    @Override  
    public int getInteger() {  
        return 2 + super.getInteger();  
    }  
}
```

הרעיון הוא לנתק את הקשר בין האבסטרקציה למימוש עצמה כדי שניהם יהיו יכולים להשתנות באופן עצמאי, על ידי סיפוק מבנה גישורי שייהי ביניהם.

מתי משתמש בפתרן?

כשנרצה להימנע מחיבור קבוע בין אבסטרקציה למימושה.

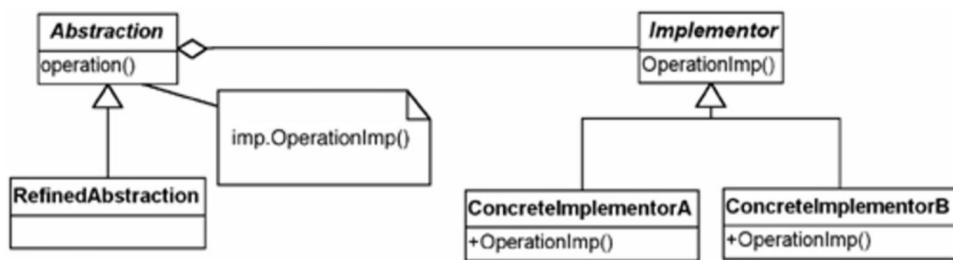
כשהשימוש הספציפי חייב להיבחר או להשתנות בזמן ריצה.

כשגם השימוש וגם האבסטרקציה אמורים להיות ניתנות להרחבה.

כשנרצה לשינויים במימוש ספציפי לא יפגעו בקוד של הלוקו. שהוא לא יצטרך לקלפל מחדש.

כשנרצה להסתיר מימוש אבסטרקט מלהlkoth.

כשיש לנו יותר מדי מימושים למחלקה אבסטרקטית אחת.



המימוש:

לפתרן יש שני

חלקים:

האבסטרקט

והמימוש. שניהם

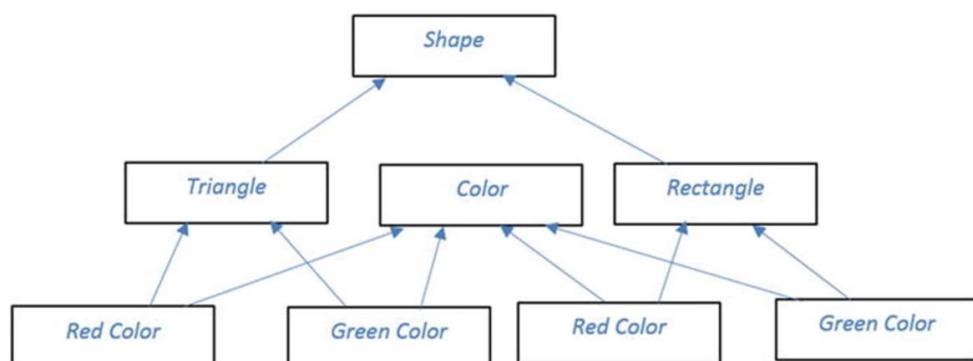
יכולים להיות

מחלקה

או אבסטרקטית או

אינטראפייס. הפtran נותן את האפשרות לעבוד על שניהם באופן עצמאי. הלוקו יכול לגשת רק לחולק האבסטרקט. האבסטרקציה מחזיקה בתוכה את ה `Implementer`.

גם כאן יש העדפה של קומפוזיציה\אגרגציה לעומת הורשה.



דוגמא מהתרגיל:

הדיagramma נראה מסובכת מדי.

בעזרת הפtran

ניתן להפריד בין

הצורות לצבעים

ולפוך את

הדברים ליותר

נוחים.

ניצור אינטראפייס של `Color` עם המетодה `fillWithColor(int border)`. ולה בעצם שת'

מימושים: `Red`, `Green`.

```
public abstract class Shape {  
    public abstract void drawShapes(int border);  
    public abstract void modifyBorder(int border);  
    protected Color color;  
    protected Shape(Color color)  
    {this.color = color;}  
}
```

```
public class Rectangle extends Shape {  
    protected Rectangle(Color color) {  
        super(color);  
    }  
    @Override  
    public void drawShapes(int border) {  
        System.out.print("This is Rectangle colored: ");  
        color.fillWithColor(border);  
    }  
    @Override  
    public void modifyBorder(int border) {  
        System.out.println("Changing the border: ");  
        drawShapes(border);  
    }  
}
```

از ניצר מחלוקת  
אבסטרקטית של צורה  
עם שתי המетодות הניל'ן  
עם שתי מימושים:  
.Triangle, Rectangle

נשים לב שתוך הצורה  
מוחזק שדה של צבע.

לשם ההמחשה:

ואז הלקוח:

```
Shape shape1 = new Triangle(new GreenColor());  
shape1.drawShapes(border: 20);  
shape1.modifyBorder(60);
```

וכך בעצם השגנו אי תלות בין צבעים וצורות.

השוני בין הפטון זהה ל Adapter הוא שה Adapter מתמקד יותר בפתרון אי התאמה בין שני אינטראפיסים קיימים. בדרך כלל משתמש בו במהלך הפיתוח שגילינו את אותן אינטראפיסים.

ה Bridge מגשר בין חלק אבסטרקטי למימושים הפוטנציאליים שלו. והשימוש בו הוא בתחלת פיתוח המערכת

## Composite .8

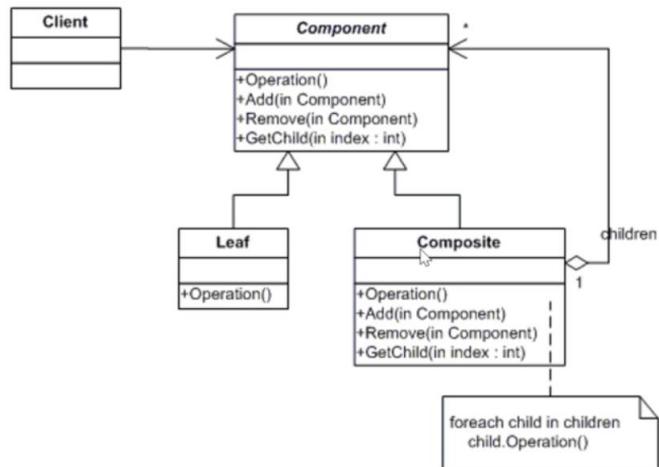
נועד בשביל להרכיב ולחבר אובייקטים שונים לבנייה עצ'י כדי להראות את המבנה ההיררכי בהם. מחזיקים את האובייקטים בקומפוזיציה ומתנהגים לכלם באותו אוף. במבנה העצ'י אנחנו מחזיקים אובייקטים בעליים וקבוצות בצמתים. בעזרת אינטראפייס דומה לשניהם נוכל להשתמש בהם באותו אוף.

דוגמא: למדל עובדים בחברה לפי היררכיות של המחלקות שבהם הם עובדים. GUI של תפריטים ותת תפריטים למערכת.

מתי משתמש בפתרן?

כהשלוקה צריך להתעלם בין קומפוזיציה של אובייקטים לאובייקטים אינדיבידואליים, וכשנרצה לטפל בהם באותו אוף. כשאנו מודאגים משימוש יתרה בזיכרון. כשנרצה ליעל את תליך יצירת האובייקטים באמצעות שיטופם. כישיש לנו אובייקטים בצורה היררכית (הורה וילד).

קל להוסיף אובייקטים חדשים להיררכיה, והלקות לא באמת יודע מזה.



המימוש:

יש את אינטראפייס Component שמספק התנהגוויות לכל האובייקטים שיופיעו בעץ. המימושים שלו זה עליים שהם הפקדים הקונקרטיים ויש את "אובייקטים הקומפוזיציות" שמכילים בתוכם עליים.

קל לראות מהディגרמה שהפתרן נוגד את העקרונות: Interface Segregation Principle כי העלה חיבר ו-Single Responsibility למשתודות שלא קשורות אליו (תלו依 או מסתכלים על זה). בכל מקרה אפשר להפריד את מתודות האלו לאינטראפייס אחר מה שיפגע ביכולות השימוש בשני הסוגים באותו אוף. Trade off

דוגמת קוד:

```
interface Employee { //Component
    public void showEmployeeDetails();
}
```

ニיצר אינטראפייס של עובד עם מתודה של להציג מי הוא. אחר כך, נוסיף שני מימושים למחלקה שלהם: מתכנת ומנהל.

```
class Directory implements Employee{
    private List<Employee> employeeList = new ArrayList<>();
    @Override
    public void showEmployeeDetails() {
        for (Employee emp:employeeList) {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(Employee emp) { employeeList.add(emp); }
    public void removeEmployee(Employee emp) { employeeList.remove(emp); }
}
```

השלב השני היא לייצר מחלקה שמייצגת קבוצה של עובדים שמשמשת את האינטראפייס של עובד. למחלקה יש את היכולות להוסיף ולהוריד עובדים.

```
Employee dev1 = new Developer(empld: 100, name: "Jason Fedin", position: "Pro Developer");
Employee dev2 = new Developer(empld: 101, name: "Myra Fedin", position: "Entry level Developer");
Directory engDirectory = new Directory();
engDirectory.addEmployee(dev1);
engDirectory.addEmployee(dev2);
Employee man1 = new Manager(empld: 200, name: "Jennifer Fedin", position: "SEO Manager");
Employee man2 = new Manager(empld: 201, name: "Ian Fedin", position: "Myra's Manager");
Directory accDirectory = new Directory();
accDirectory.addEmployee(man1);
accDirectory.addEmployee(man2);
Directory companyDirectory = new Directory();
companyDirectory.addEmployee(engDirectory);
companyDirectory.addEmployee(accDirectory);
companyDirectory.showEmployeeDetails();
```

כך נראה צד  
הלקוח:

\*\* באותו מידה יכולנו לאחד בין המחלקות Manager - .Directory

## Decorator .9

שימוש בפטן זה נותן לנו את האפשרות לצרף אחריות ופונקציונאלית חדשה לאובייקט קיים בצורה דינמית (בזמן ריצה) מבלי לשנות את המחלקה עצמה. מופעים אחרים של המחלקה לא ישפעו מהשימוש בפטן למופע אחר.

למה לא הורשה?

כאן אנחנו רוצים להוסיף אחריות למופע יחיד ולא למחלקה שלמה.

בהורשה צריך ליצור מחלקות חדשות. מה שיכל לגרום למורכבות במערכת.

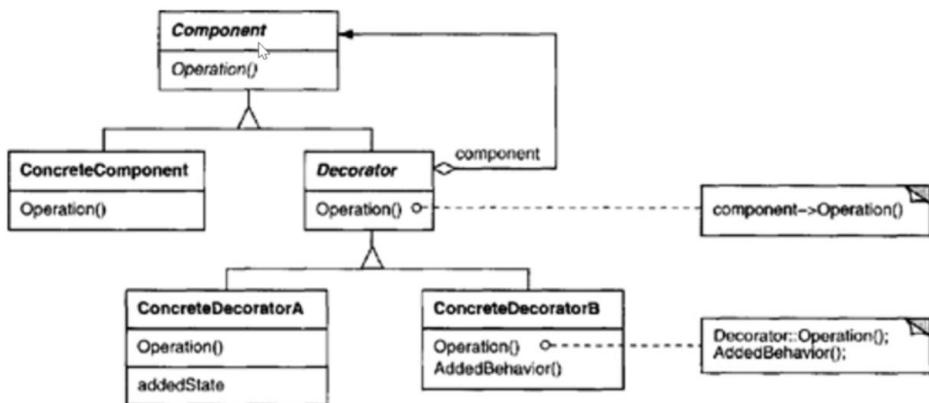
בהורשה יורשים שדה מסוים לכל המחלקות היורשות למרות שרצית אותו רק למופע יחיד. כאן אפשר לצרף גם לנתק חלק מהפונקציונליות מהמופעים השונים.

דוגמא: יש לי בית ואני רוצה להוסיף עוד קומה חדשה מבפנים לפגוע בכל הארכיטקטורה הקיימת כרגע.

דוגמא נוספת: יש לי textField שאון לו scroller ואני רוצה רזהה. אך אני בונה ScrollDecorator.

דרך פטן זה קל להוסיף פעמיים את אותו תcona. אנחנו יכולים לבנות אובייקטים בהדרגה בעזרת הוספת תכונות בהמשך. חישרון אחד בפטן הוא שלפעמיים במערכת יש המונן מחלקות ואובייקטים קטנים.

הימוש:



ה- **Component** הוא האינטראפיס של האובייקט המבוקש. אפשר להוסיף לאחריות באופן דינامي.

ה- **ConcreteComponent** הוא המחלקה שאני רוצה להוסיף לה דקורטורו.

ה- **Decorator** הוא מחלקה אבסטרקטית שמממשת את ה **Component** ושמחזקיה בתוכה אובייקט שלה.

בתוך ה- **ConcreteDecorator** יש את יכולת להוסיף אחריות לאובייקט המבוקש.

```

abstract class Component {
    public abstract void doJob();
}

class ConcreteComponent extends Component {
    @Override
    public void doJob() {
        System.out.println(
            "I am from a Concrete Component" +
            " - I am closed for modification");
    }
}

```

### דוגמא (כללית) 1:

נגיד שיש לנו קומפוננטה עם המתודה הנ"ל.

ניצור מחלקה אבסטרקטית של דקורטור  
שבתוכה שדה מסווג קומפוננטה עם מתודה  
כמו שלעיל.

```

abstract class AbstractDecorator extends Component {
    protected Component com;
    public void SetTheComponent(Component c) { com = c; }
    public void doJob() {
        if (com != null) {
            com.doJob();
        }
    }
}

```

```

class ConcreteDecorator1 extends AbstractDecorator {
    public void doJob() {
        super.doJob();
        // add additional responsibilities
        System.out.println("I am explicitly from Dec1");
    }
}

class ConcreteDecorator2 extends AbstractDecorator {
    public void doJob() {
        System.out.println("***START Dec2***");
        super.doJob();
        System.out.println("***END. Dec2***");
    }
}

```

אפשר לבנות דקורטורים ספציפיים כדי להוסיף עוד פונקציונליות לאותה מתודה.

```

ConcreteComponent cc = new ConcreteComponent();
ConcreteDecorator1 cd1 = new ConcreteDecorator1();
cd1.SetTheComponent(cc);
cd1.doJob();
//Decorator on Decorator
ConcreteDecorator2 cd2= new ConcreteDecorator2();
cd2.SetTheComponent(cd1);
cd2.doJob();

```

צד הלקוח:

דוגמא 2: היכולת לקרוא מקובץ ולהמיר את התוכן תוך כדי לאותיות קטנות.

```

public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) { super(in); }
    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }
}

```

## Facade .10

מספק אינטראפיביוס מואחד לקבוצה של אינטראפיביוסים קיימים במערכת. מסתיר את הסיבוכיות באינטראפיביוסים האחרים. האינטראפיביוס החדש לא מוסיף פונקציונליות חדשה. ממש כמו במצבות שרוואים בניין, רואים רק את הפנים הידידותיות שלו ולא את המורכבות של הבניה עצמה.

דוגמה: רוצים לתכנן יומם הולדת בלי הבלגן של התיכנון. שוכרים אחד שיעשה את כל העבודה הקשה וניתן לו את המידע המיניימי. דוגמא נוספת: כשמחשב מתחילה לפעול זה מערב את המעבד, הזיכרון, והדיסק הקשיח. אז נאחד את הכל לפעולה אחת לשם פשוטה.

האינטראפיביוס החדש לא מסתיר את האינטראפיביוסים האחרים מhalbוקו. האינטראפיביוסים במערכת לא אמורים להיות קשורים ותלוים באינטראפיביוס החדש (לא יודעים בכלל שהוא קיים).

יתרונות:  
 מפחית את מספר האובייקטים שהלkop מהתמודד איתם. מפחית את ה coupling מהALKOHOL.  
 לכל האינטראפיביוסים. הופך את הדברים ליוצרים קלים לשימוש.  
 בנוסף, יכול לחסול תלויות מסוימות או מעגליות בין אובייקטים.

דוגמת קוד:

```
public class HomeTheaterFacade {  
    Amplifier amp;  
    Tuner tuner;  
    DvdPlayer dvd;  
    CdPlayer cd;  
    Projector projector;  
    TheaterLights lights;  
    Screen screen;  
    PopcornPopper popper;  
    //Contractor below  
    public void watchMovie(String movie) {  
        System.out.println(  
            "Get ready to watch a movie...");  
        popper.on();  
        popper.pop();  
        lights.dim( level: 10 );  
        screen.down();  
        projector.on();  
        projector.wideScreenMode();  
        amp.on();  
        amp.setDvd(dvd);  
        amp.setSurroundSound();  
        amp.setVolume(5);  
        dvd.on();  
        dvd.play(movie);  
    }  
}
```

בוא נדמיין שאנו רוצים לראות סרט, אז פועלה מאוד מורכבת שמצריכה הכנות ומעורבות עם מגוון מכשירים כמו: להפעיל את גנג הדיסק, לכבות את האורות, להדליק את המסר, ועוד.

דרך האינטראפיביוס הנ"ל יהיה ניתן דרך פעולה פשוטה לדאוג לכל התהליך המסובך ולהחסוך זאת מהALKOHOL.

\*\* אפשר לשים לב שהקונסטרוקטור כאן צריך לקבל המונ פורמטרים ולכן לנו להשתמש ב Builder.

## Flyweight .11

הפתרן משתמש בשיתוף כדי לתמוך במספר רב של אובייקטיםudenim ביעילות. המטרה היא להפחית את כמות האובייקטים, מה שמקל את השימוש בזיכרון. מה שעוד תורם לזה הוא שיתוף נתונים כמה שאפשר. בנוסף משפר ביצועים.

הפתרן מנסה להשתמש באובייקטים דומים למזה שניץ צריך באמצעות אחסוןם. אנחנו ניצור אובייקט חדש רק כאשר אין אף אובייקט שמתאים לדרישות. אוטם אובייקטים מאוחסנים לא ניתנים לשינוי.

דוגמא: שני אנשים מחפשים דירה עם נתונים ספציפיים דומים. לא מרצוים עם הדירות שיש קרוגע. עד שמצאו דירה הם החליטו לגור שם ביחד.  
דוגמא נוספת: String pool in Java.

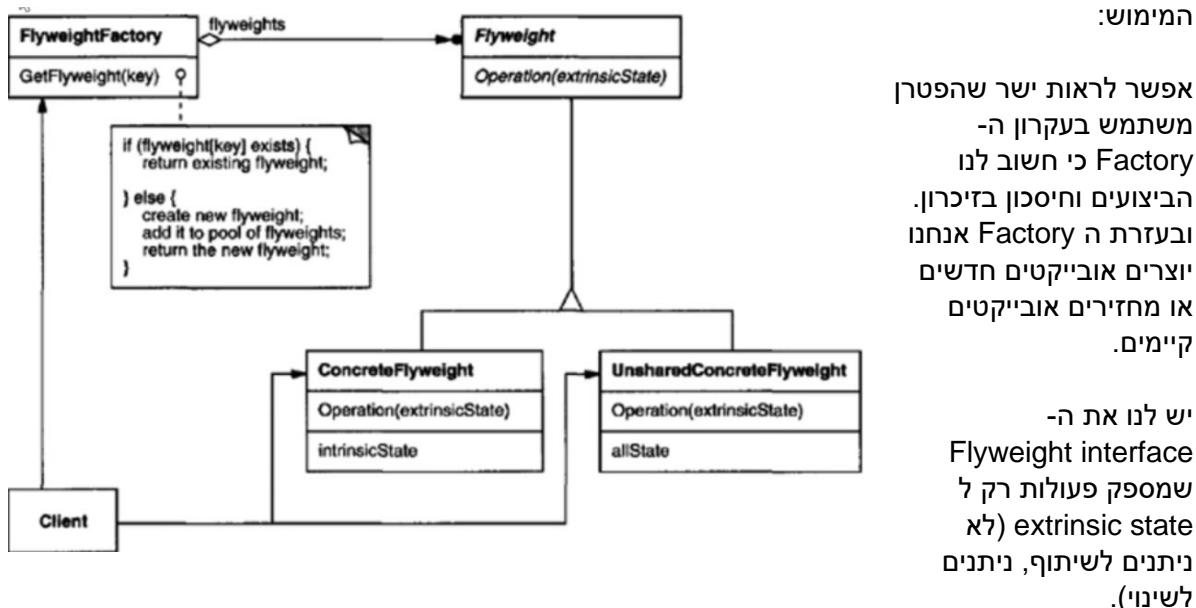
### Intrinsic vs. extrinsic state/properties

Intrinsic state/properties - יכול להיות מאוחסן ונitin לשיתוף.  
extrinsic state/properties - תלוי בתוכן האחסון ולא ניתן לשיתוף.

דוגמא: נניח שיש לנו אפליקציה של text editor ואת המחלקה Character שמייצגתתו ובתוכה השדות: שם, גודל, וגוון. אם הלקוח כתוב את האות 'B' ואז שוב פעם, אז אין שום הבדל בין שני המופעים. נאמר כי כל המאפיינים של המחלקה שלתו הם intrinsic. אם נוציא למחלקה את השדות: מס' שורה ומס' עמודה אז בין כל מופע שלתו יש לפחות שתי הבדלים. لكن הם לא ניתנים לשיתוף. נאמר-cut שמוופעים אלו הם extrinsic.

### מתי משתמש בפתרן?

כשיש במערכת המון אובייקטים ובעיקר שהרבה מהם זהים.  
כאשר עלויות האחסון גבוהות בגלל הכמות העצומה של האובייקטים.



ה- ConcreteFlyweight אחראי על האובייקטים השימושיים.

```

public interface IRobot { //Flyweight
    void print();
    // extrinsic data is passed as arguments
    void setColor(String colorOfRobot);
}

class Robot implements IRobot {
    String robotType;
    public String colorOfRobot;
    public Robot(String robotType) {
        this.robotType=robotType;
    }
    public void setColor(String colorOfRobot) {
        this.colorOfRobot=colorOfRobot;
    }
    @Override
    public void print() {
        System.out.println(" This is a " +robotType+
            " type robot with "+colorOfRobot+ "color");
    }
}

```

דוגמת קוד:

תחילת ניצר אינטראפיו IRobot ומיימוש ספציפי Robot עם סוג צבע.

השלב הבא הוא לייצר את מפעלי הרובוטים. מפה של כל הרובוטים שקיים. ברגע שיבקשו רובוט (לפי סוג) שכבר נוצר נביא להם את אותו אחד. אחרת, ניצור אחד חדש ונוסיף אותו למפה.

```

class RobotFactory {
    Map<String, IRobot> robots = new HashMap<~>();
    public int totalObjectsCreated() { return robots.size(); }
    public IRobot getRobotFromFactory(String robotType) throws Exception {
        IRobot myRobot = null;
        if (robots.containsKey(robotType)) {
            myRobot = robots.get(robotType);
        } else {
            switch (robotType) {
                case "King":
                    System.out.println("We don't have a King Robot." +
                        " So we are creating a King Robot now");
                    myRobot = new Robot( robotType: "King");
                    robots.put("King", myRobot);
                    break;
                case "Queen":
                    System.out.println("We don't have Queen Robot." +
                        " So we are creating a Queen Robot now .");
                    myRobot = new Robot( robotType: "Queen");
                    robots.put("Queen", myRobot);
                    break;
                default:
                    throw new Exception(" Robot Factory can create only" +
                        " King and Queen Robots");
            }
        }
        return myRobot;
    }
}

```

```

Robot robot = null;
for (int i = 0; i < 3; i++) {
    robot =(Robot) myfactory.getRobotFromFactory( robotType: "King");
    robot.setColor(getRandomColor());
    robot.print();
}
for (int i = 0; i < 3; i++) {
    robot =(Robot) myfactory.getRobotFromFactory( robotType: "Queen");
    robot.setColor(getRandomColor());
    robot.print();
}
int NumOfDistinctRobots = myfactory.totalObjectsCreated();

```

צד הלקוח יראה כך:  
 מה שקרה כאן הוא  
 שאנו יוצרים רובוט  
 אחד מסוג King  
 מקבלים אותו כל פעם  
 ומשנים לו את הצבע.  
 הוא שיתופי בగל  
 מאפיין דומה - הסוג.  
 כך גם עבור הרובוט  
 מסוג Queen.

לכן מספר הרובוטים שייהו בmphga בסוף הוא: 2.

\*\* פטן זה מזכיר לפעמים את פטן הסינגלטן. בדוגמה שלנו לאvr כרך הדבר מכיוון שאנו  
 משתפים את הדמיון בין האובייקטים ולא בהכרח רצים ליצור רק אובייקט אחד של מחלוקת  
 אחת. והואו אובייקט שיתופי גם ניתן לשינוי.

אם לא היה לנו את מאפיין הצבע ברובוט ובנוסף במקום מאפיין סוג הרובוט היה לנו פשוט  
 שתי מחלוקות שונות אז כאן זה היה מאד מזכיר את הסינגלטן כי יש לנו אובייקט אחד מכל  
 מחלוקת שיתופי שלא משנים אותו.

## Proxy. 12

מספק תחליף או מציין מקום לאובייקט אחר כדי לשלוט בגישה אליו. משתמשים בו כאשר אינם רוצים לספק גישה מבוסקת על פונקציונליות מסוימת. הגדרה פורמלית: אדם המוסמך לפעול עבור אדם אחר. סוכן או תחליף. הסכמה לפעול למען אחר. יש מקרים שבהםwkoch לא יכול לגשת שירות לאובייקט מסוים ובכל זאת רוצה. אנחנו באים לפתר את הבעיה הזאת. עוד שימוש לפטרן הוא לשמש כמעטפת לביצועים יותר טובים.

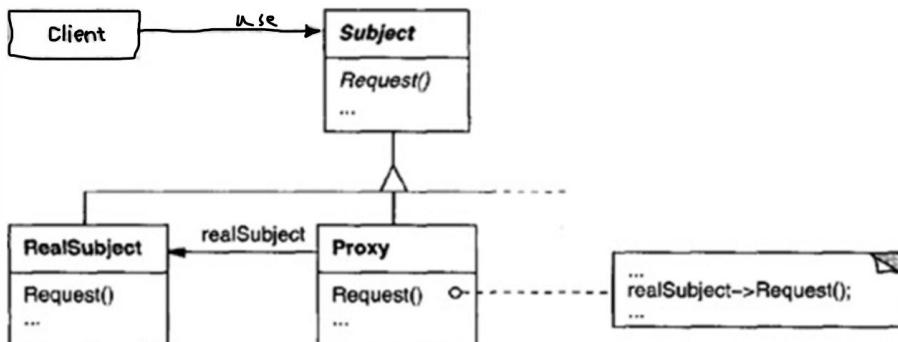
דוגמה: כרטיס אשראי הוא פרויקט לכסף בחשבון הבנק. הוא מספק גישה בטוחה ומבוקרת לחשבון.

סיבה אחת לשלוט בגישה לאובייקט היא כדי ליצור אותו רק בפעם הראשונה שמשתמשים בו. מזכיר את Bill Pugh בסינגלטון. דוגמא לכך תהיה לפתיחת אתר שיש בו המון תמונות. אנחנו לא רוצים לשלוח את כל התמונות ללקוח בהתחלה, רק את אלו שהוא יכול לראותן כרגע.

### סוגי פרויקטים:

- Remote Proxy - מנהל את האינטראקציה בין הלקוח לאובייקט המרוחק. מספק reference לאובייקט שנמצא למרחב זיכרון של אותו מכונה או אחרת.  
 - Virtual Proxy - שולט בגישה לאובייקט שיקר להחזיק מופע שלו. האובייקט יוצר רק כשנចטרך אותו.  
 - Copy-On-Write Proxy - דוחה העתקה (clone) של אובייקט עד שיידרש על ידי לקוח.  
 - Protection (Action) Proxy - מספק ללקוחות שונים רמות שונות של גישה לאובייקט.  
 - Cache Proxy - מספק אחסון זמן עבר פועליה יקרה כך שמספר לקוחות יכולים לשאוף בינם לבין עצמם תוצאה.  
 - Firewall Proxy - מגן על אובייקטים מלקוחות זדוניים. או ההפר.  
 - Synchronization Proxy - מספק גישה מרובה לאובייקט.  
 - Smart Reference Proxy - מספק פועלות לאובייקטים שהוזכרו, כמו לספרור את כמות הפעמים שאוזכר.

### הIMPLEMENTATION:



ה- Subject הוא אינטראקטיבי וה- RealSubject זה בעצם האובייקט שאנו רוצים לגשת אליו.

```

public interface Image { //Subject
    void display();
}

class RealImage implements Image {
    private String fileName;
    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(this.fileName);
    }
    @Override
    public void display()
    {System.out.println("Displaying " + fileName);}
    private void loadFromDisk(String fileName)
    {System.out.println("Loading " + fileName);}
}

```

דוגמת קוד:

יש לנו אינטראפיס של תמונה  
ומימוש ספציפי. זהו בעצם  
האובייקט שאנו רצים  
להשתמש בו.

```

class ProxyImage implements Image {
    private RealImage realImage;
    private String fileName;
    public ProxyImage(String fileName) {
        this.fileName = fileName;}
    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}

```

אפשר ליצור פרויקט כדי  
LAGSHET AL HA-OVIEKUT BE-ZORAH  
MA-OBUTCHOT OM-BOKERAT.

NESSIM LB SH-BTUR HAPROJEKT  
ISH LENO AT HA-OVIEKUT HAYUD  
BKOMPOZIYA\AGRAGTSIA.

```

Image image  = new ProxyImage( fileName: "image.jpg");
// image will be loaded from disk
image.display(); //Here the real image created
image.display(); //Just display again

```

צד הלקוח:

## Behavioral Design Patterns

פטרנים אלו ידגו לתקשרות בין אובייקטים. הם מספקים פתרונות לאיר להפוך אובייקטים לתלויים או לבלי תלוימם. Class פטרנים מסווג זה משתמשים בהורשה כדי לתאר אלגוריתם או זרימה של המערכת. Object פטרנים מסווג זה משתמשים בקומפוזיציה כדי לתאר איך קבוצה של אובייקטים משתפים פעולה כדי לבצע משימה שאף אחד מהם לא יכול לבצע לבד.

### Chain of Responsibility .13

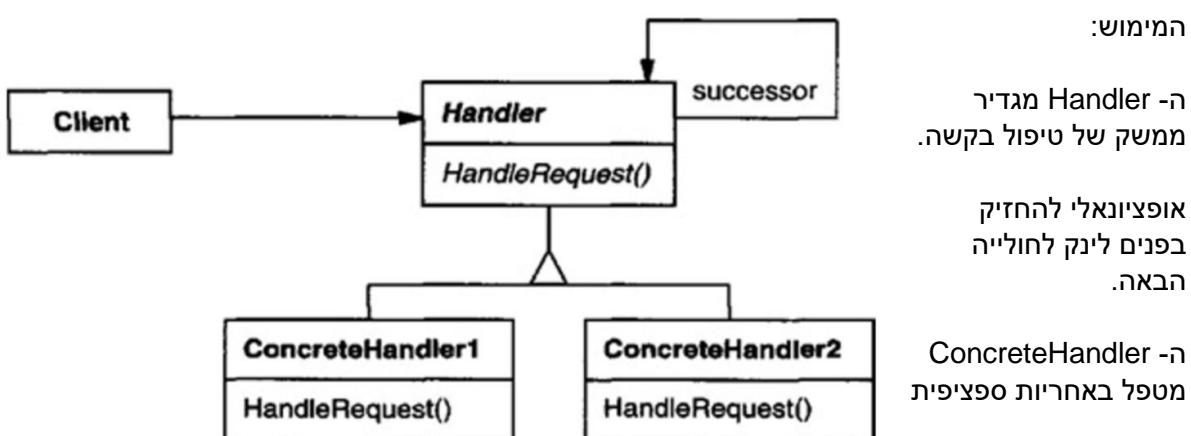
מטרתו היא להימנע מ-Coupling אצל שולח הבקשה למקבל על ידי מתן הזדמנויות ליותר מאובייקט אחד לטפל בבקשתה. מעבירים את האחריות כמו שרשות עד שימושה דואג לטפל בעיה. אנחנו יכולים להוסיף עוד חוליות לסוף שרשת בזמן ריצה.

דוגמאות:  
밀וי טופס פידבק מצד לקוחות בחברה. נשלח את הטופס רק למחלקות הקשורות לפידבק.  
העברת אחריות של אירועים כמו לחיצה על העכבר.  
בלוק של try and catch.

מתי נשתמש בפתרן?  
כשנרצה להפחית את התלות בבקשתה בין שולח למקבל.  
כמספר אובייקטים שנקבעו בזמן ריצה לטפל בבקשתה מסוימת.  
כאנו נרצה לא רוצים לציין מי מקבל של הבקשה.

יתרונות נוספים:  
החוליות בשרשרת לא חיברים להבין את המבנה שלה ולא להחזיק מופעים של כל החוליות, אלא רק של הבא בתור אחריהם.  
הפתרן מאפשר להוסיף או לבטל אחריות מהאובייקטים בזמן ריצה, ובנוסף לזה את האופציה לשנות את הסדר בין האובייקטים.

חסרונות:  
אין הבטחה שימושה יתפao ויקח על עצמו את האחריות למשימה מה שייגרום לשרשרת ליפול. ממש כמו בExceptions שאף אחד לא תופס את השגיאה.  
דבר נוסף הוא שבקשתה לדבאג את הקוד.



שלו. אם הוא יכול לטפל בא הוא יעשה זאת. אחרת, יעביר את האחריות הלאה.

```
//Represents the link in the chain
interface DispenseChain {
    void setNextChain(DispenseChain nextChain);
    //method to handle the request
    //Currency is just 'int amount'
    void dispense(Currency cur);
}
```

```
class Dollar50Dispenser implements DispenseChain {
    private DispenseChain chain;
    @Override
    public void setNextChain(DispenseChain nextChain)
    {chain = nextChain;}
    @Override
    public void dispense(Currency cur) {
        if (cur.getAmount() >= 50) {
            int num = cur.getAmount() / 50;
            int remainder = cur.getAmount() % 50;
            System.out.println("Dispensing " + num + " 50$ note");
            if (remainder != 0)
                this.chain.dispense(new Currency(remainder));
        }
        else {
            this.chain.dispense(cur);
        }
    }
}
```

#### דוגמת קוד:

אנו ננסה לבנות סופומט שמקבל סכומים מהלך שרך מתחלקים ב-10 ליל"ש ומחלקים אותם לסכומים האפשריים: 10, 20, 50.

בננה תחילת אינטראיט שמייצג חוליה בשרשרת עם מתודה של גישה לחוליה הבהה וגם חילוק הסכום לסכומים האפשריים (זריקת אחריות).

הימוש הראשון לחלוקת תהיה זאת שבתמונה השנייה. הבדיקה הראשונית על הסכום היא עם החוליה הזאת שבודקת אם הסכום גדול מ- 50 אז מטפלת בו. כך או כך מעבירה את האחריות להבא בתור שכירך.

כמו כן, ניצור את המימושים - `Dollar20Dispenser` ו- `Dollar10Dispenser`.

```

public class Client {
    private DispenseChain c1;
    public Client() {
        this.c1 = new Dollar50Dispensor();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();
        c1.setNextChain(c2);
        c2.setNextChain(c3);
    }
}

```

cutet נטמיך בצד  
הלקוח:

המחלקה הזאת נעודה  
ליצור את השרשרת  
ולדאוג שכל אחד  
יעביר את האחריות  
להבא בתורו שלא  
יוכל להתמודד איתה  
לגמרי.

\*\* חשוב לשים לב שהסדר של החוליות במקורה זהה הוא מאד חשוב.

```

Client atmDispenser = new Client();
while(true)
{
    int amount = 0;
    System.out.println("Enter amount to dispense");
    Scanner input = new Scanner(System.in);
    amount = input.nextInt();
    if (amount % 10 != 0) {
        System.out.println("Amount should be in multiple of 10s.");
        return;
    }
    atmDispenser.c1.dispense(new Currency(amount));
}

```

כאן אנחנו משתמשים בכוספומט. מה שיקרא אם נכניס את הסכום 130 לדוגמא הפלט יראה  
כך:

```

130
Dispensing 2 50$ note
Dispensing 1 20$ note
Dispensing 1 10$ note

```

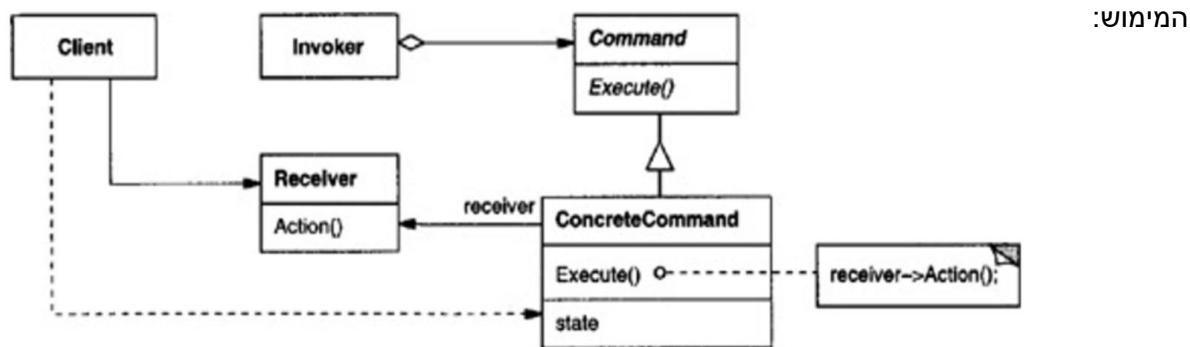
## Command .14

הפתרן מחזיק בתוכו (אינטראקטיה) בקשה אובייקט (פקודה). אפשר לנו להגדיר פרמטרים ללקוחות עם בקשות שונים. את הבקשות אפשר לסדר, לששות Undo עליהם, redo, ואפילו execute. אובייקט command מחזיק בתוכו את הבקשה על ידי חיבור קבוצה של פעולות על receiver ספציפי, וזה נעשה על ידי חישוף מתודת אחת שעשוה invoke לחלק מהפעולות אצל receiver.

קל להוסיף פקודות חדשות מבלי לפגוע בקוד בזכות האי תלות בין כל המשתפים בפתרן.

דוגמא: ביצוע הזמן לקנייה ומירה של מנויות. יש לנו אינטראפיס של הזמן (הפקודה) ומימושים ספציפיים שהוא חלקה שאחראית על קנייה ואחת על מירה. יהיה לנו חלקה שמייצגת מניה (הבקשה). בנוסף, יהיה חלקה של ברוקר (Invoker) שאחראי על ליקחת ולמוך הזמן.

דוגמא ל- undo: למחוק שירות של עירפן. דוגמא ל- redo: לתוכנן מחדש תוכנה.



יש לנו אינטראפיס של command. עושים invoked דרך המתודה שבפנים. ישאל את ה- receiver כדי לבצע פעולה.  
ה- ConcreteCommand מחזיק בתוכו (אגרגציה) את ה- receiver.

אובייקט ה- command מסוגל לקרוא למетодות ספציפיות אצל ה- receiver.  
ה- Invoker מכיר רק את command interface ולא את הפקודה הספציפית.  
הלקוח מחזיק את ה- command וואז מחייב איזה פקודה צריכה להתבצע ומתי.

```

public interface FileSystemReceiver {
    void openFile();
    void writeFile();
    void closeFile();
}

class WindowsFileSystemReceiver implements FileSystemReceiver {
    @Override
    public void openFile()
    {System.out.println("Opening file in Windows OS");}
    @Override
    public void writeFile()
    {System.out.println("Writing file in Windows OS");}
    @Override
    public void closeFile()
    {System.out.println("Closing file in Windows OS");}
}

```

דוגמת קוד:

מערכת קבצים  
שמאפשרת  
לפתח, לסגור,  
ולכתוב לקבצים.

תחילה נוצר  
אינטראפיס של  
receiver שהוא  
בעצם אחראי על  
כל הפעולות על  
הקבצים. ממש לו  
מיושם ספציפיים  
כרצוננו.

```

interface Command {
    void execute();
    //could add an undo/redo commands
}

class OpenFileCommand implements Command {
    private FileSystemReceiver fileSystem;
    //store previous stat for undo, String someState
    public OpenFileCommand(FileSystemReceiver fs)
    {fileSystem = fs;}
    @Override
    public void execute() {
        //save previous state, in case undo called
        this.fileSystem.openFile();
    }
}

```

הצעד הבא הוא  
לبنנות אינטראפיס  
של command  
עם מיושים  
אשר מייצגים את  
הפעולות  
שהגדרנו לפני.  
3 פעולה<-> 3  
מיושים.  
צירפת מימוש  
ספציפי, כל  
המיושים זהים  
מלבד הקראיה  
לפונקציה  
הספציפית של ה-  
receiver execute().

אחר מכן, נוצר  
את ה invoker  
שבסר הכל  
מחזיק  
command  
ועושה דlgציה  
על המתודה שלו.

```

class FileInvoker {
    public Command command;
    public FileInvoker(Command c) { command = c; }
    public void execute() { command.execute(); }
}

```

```
public class FileSystemReceiverUtil {  
    public static FileSystemReceiver getUnderlyingFileSystem(){  
        String osName = System.getProperty("os.name");  
        System.out.println("Underlying OS is:"+osName);  
        if(osName.contains("Windows")){  
            return new WindowsFileSystemReceiver();  
        }else{  
            return new UnixFileSystemReceiver();  
        }  
    }  
}
```

לשם הנוחות  
נבנה את  
המחלקה  
הנ"ל רק כדי  
לקבוע איזה  
receiver  
ספציפי ליצור  
בשימוש.

באוטו מידה  
ישולנו לבנות  
.factory

צד הלקוח  
יראה כך:

```
FileSystemReceiver fs =  
FileSystemReceiverUtil.getUnderlyingFileSystem();  
Command openFileCommand = new OpenFileCommand(fs);  
FileInvoker file = new FileInvoker(openFileCommand);  
file.execute();
```

אפשר לראות שתחילה אנחנו מחליטים איזה receiver ליצור, לאחר מכן מגדירים את הפעולה שנרצה לעשות, בונים invoker דרכו נוכל לבצע את הפעולה עצמה.

השагנו גמישות בקוד: נוכל להוסיף בקלות פקודות חדשות, רק צריך להוסיף את מתודה מתאימה ב- receiver. כמו כן, אנחנו יכולים בקלות להוסיף receiver חדש.

## Interpreter .15

מספק דרך להעריך את דקדוק או ביטוי של שפה. המפרש או המתורגם מקבל שפה ובעזרת הדקדוק שלו הוא מפרש משפט לשפה אחרת.

דוגמאות: גוגל טרנסלט, הקומפיילר של java: ממיר קבצי java ל byte code.

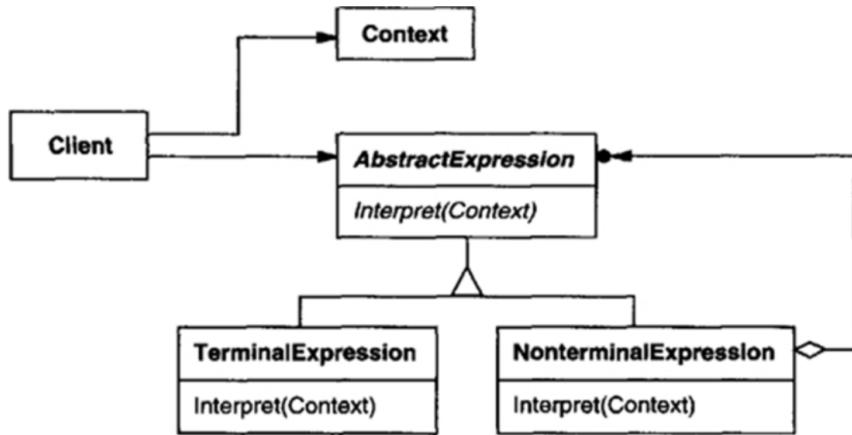
המימוש:

יש לנו את AbstractExpression שהוא האבסטראקציה של המפרש עצמו.

אחד המימושים שלו הוא TerminalExpression שמתעסק בערכים hardcoded שזה ערכיהם שאי אפשר לפרוק כמו מילה או מספר.

בניגוד לו - nonterminal שאפשר לפרוק, כמו הביטוי: 5+9.

ה- Context מכיל מידע גלובלי על המפרש. התרגום נמצא כאן. המתודה interpret תקבל אותו כפרמטר ונקרא למתחודה הספציפית בהתאם ל- TerminalExpression המתאים.



דוגמת קוד:

תחילה ניצר את ה context שיש לו שתי מתחודות ששמירות מספר לבינארי או הקסיה.

השלב הבא ליצור את ה expressions (השתיים terminal). בדוגמה הם context.

כל אחד יקרה למתחודה שמתאימה לו אצל ה context.

```

class InterpreterContext {
    public String getBinaryFormat(int i)
    {return Integer.toBinaryString(i);}
    public String getHexadecimalFormat(int i)
    {return Integer.toHexString(i);}
}
  
```

```

public interface Expression {
    String interpret(InterpreterContext ic);
}

class IntToBinaryExpression implements Expression {
    private int i;
    public IntToBinaryExpression(int c) { i = c; }
    @Override
    public String interpret(InterpreterContext ic)
    {return ic.getBinaryFormat(i);}
}
  
```

```

public class Client {
    public InterpreterContext ic;
    public Client(InterpreterContext ic)
    {this.ic = ic;}
    public String interpret(String str, String base) {
        Expression exp = null;
        switch (base){
            case "Hexadecimal":
                exp = new IntToHexExpression(
                    Integer.parseInt(str));
                break;
            case "Binary":
                exp = new IntToBinaryExpression(
                    Integer.parseInt(str));
                break;
            default:
                return str;
        }
        return exp.interpret(ic);
    }
}

```

כעת אנחנו בונים את צד הלקוח  
שאחראי לבנות את ה-  
expression הספציפי שצריך.  
ניתן לראות כי הוא בעצם מבצע  
دلגציה למתודה interpret().

השימוש:

```

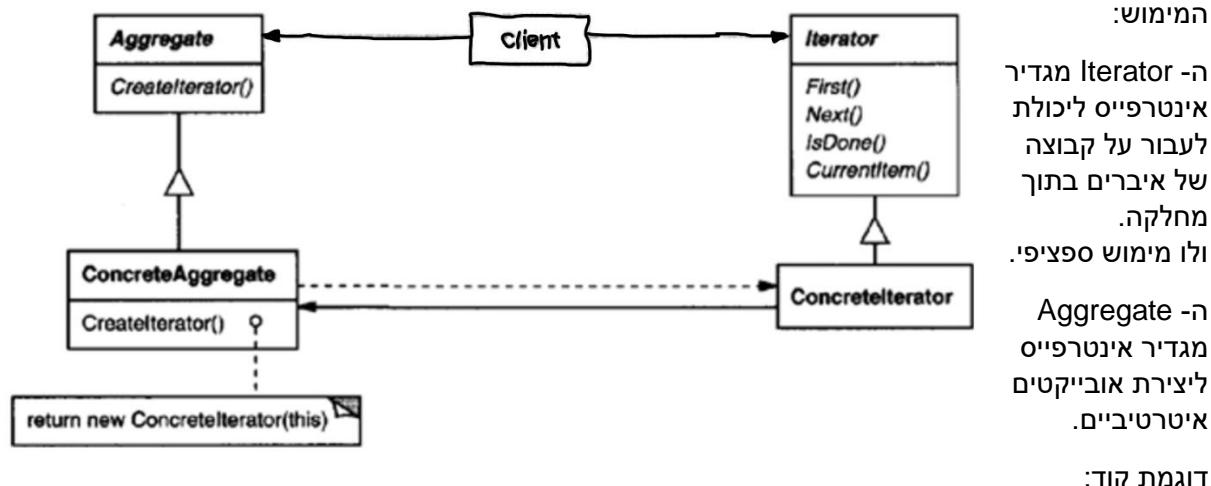
Client ec = new Client(new InterpreterContext());
System.out.println(
"28 in Binary= " + ec.interpret(str: "28", base: "Binary"));
System.out.println(
"28 in Hexadecimal"+ec.interpret(str: "28", base: "Hexadecimal"));

```

## Iterator .16

מספק דרך לגשת לאלמנטים בתוך (אגראגציה) אובייקט באופן מסודר מוביל לחשוף את ייצוגם. משתמשים בפטרן למחלקות שמחזיקות קבוצה של אובייקטים אחרים. אנחנו מסתירים את השימוש של "הטיול" ונונטנים ללוקוח רק את המתודה `.iterator`.

דוגמא: collections in java



תחילה ניצר `Iterator` עם המתודות: `.boolean hasNext()`, `Object next()` עם המתודה: `.Iterator createIterator()`

```

class NotificationCollection implements Collection {
    static final int MAX_ITEMS = 6;
    int numberofItems = 0;
    Notification [] notificationList;
    public NotificationCollection() {
        notificationList = new Notification[MAX_ITEMS];
        addItem(str: "Notification 1");
        addItem(str: "Notification 2");
        addItem(str: "Notification 3");
    }
    public void addItem(String str) {...}
    public Iterator createIterator() {
        return new NotificationIterator(notificationList);
    }
}
  
```

השלב הבא הוא ליצור מחלקה `NotificationCollection` של `Collection` שמייצרת (בסר הכל מהrozenת) שימושת את המתודה `createIterator()`. נשים לב שבתוכה יושב מערך של `Notification`ים.

```

class NotificationIterator implements Iterator {
    Notification[] notificationList;
    //curr pos
    int pos = 0;
    public NotificationIterator(Notification[] notificationList)
    {this.notificationList = notificationList;}
    public Object next() {
        Notification notification = notificationList[pos];
        pos += 1;
        return notification;
    }
    public boolean hasNext() {
        return pos < notificationList.length
            && notificationList[pos] != null;
    }
}

```

כעת נתבונן ביצירת iterator – הדבר הוא מקבל דרך הבנאי את אותה קבוצה של נוטיפיקציות ובכך הכל ממש את המתוודת הרצויות כדי לעבור בביטחון על הקבוצה.

```

public class NotificationBar {
    NotificationCollection notifications;
    public NotificationBar(NotificationCollection notifications) {
        this.notifications = notifications;
    }
    public void printNotifications() {
        Iterator iterator = notifications.createIterator();
        System.out.println("-----NOTIFICATION BAR-----");
        while (iterator.hasNext()) {
            Notification n = (Notification)iterator.next();
            System.out.println(n.getNotification());
        }
    }
}

```

השימוש:

צד הלקוח יכול להראות כך:

יצירת מחלקה בשם נוחות בלבד שטרתו להחזיק collection ולו רץ עלי.

פונקציית ה main יכולה להראות כך:

```

NotificationCollection nc = new NotificationCollection();
NotificationBar nb = new NotificationBar(nc);
nb.printNotifications();

```

## Mediator .17

פטרן המתוור, מגדר את האינטראקציה של קבוצה של אובייקטים שנמצאים בתוך אובייקט גדול.

תורם להפחחתה coupling על ידי גירמת חוסר התייחסות והפחחתה היחסים בין אובייקטים באופן מפורש.

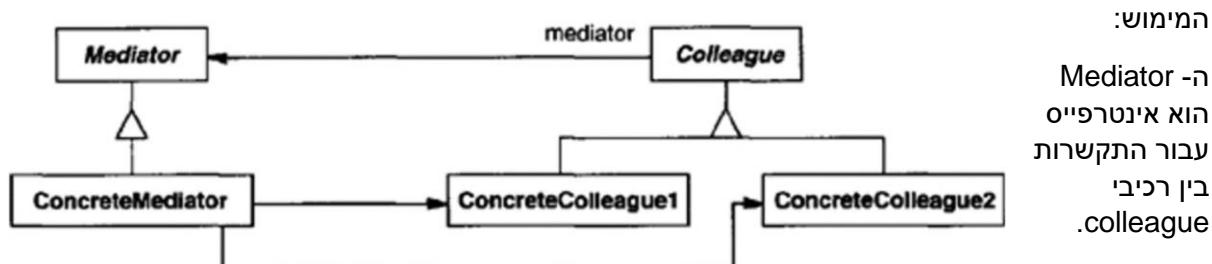
המתוור אחראי על התקשרות בין אובייקטים. לשים לב אם מספר המתוורים במערכת גדול יהיה יותר קשה לתחזק את המערכת.

דוגמא: מרכז בקרה אויר עובד כמתוור בכך שהוא גורם לתקשרות בין מטוסים שונים.

מתי נשימוש בפטרן?

כמספר אובייקטים מתחברים אחד עם השני בצורה מרוכבת.

כשימוש חוזר באובייקט הוא מאד קשה כי הוא מתייחס ומתקשר עם מספר אובייקטים.



ה- ConcreteMediator מכיר ומתוחזק את כל ה colleagues. הוא בעצם מימוש את התקשרות.

כל מחלוקת של colleague מכירה את המתוור שלה. הוא יכול לשЛОח או לקבל בקשות ממנו.

אם במערכת צריך רק מתוור אחד אז ניתן להימנע מ Abstract Mediator

דוגמא קוד- צ'אט קבוצתי בין קולגות:

```

interface ChatMediator {
    void sendMessage(String msg, User user);
    void addUser(User user);
}
  
```

תילה נבנה אינטראפיט של מתוור בין חברים הקבוצה. אחר כך נבנה את user. נשים לב שבפניהם יש לנו שדה של מתוור.

```

abstract class User { //Colleague
    protected ChatMediator mediator;
    protected String name;
    public User(ChatMediator med, String name) {...}
    public abstract void send(String msg);
    public abstract void receive(String msg);
}
  
```

```

class ChatMediatorImpl implements ChatMediator {
    private List<User> users;
    public ChatMediatorImpl() { this.users = new ArrayList<>(); }
    @Override
    public void addUser(User user) { this.users.add(user); }
    @Override
    public void sendMessage(String msg, User user) {
        for (User u:this.users) {
            if(u != user) {
                u.receive(msg);
            }
        }
    }
}

```

כעת נctrar  
לממש את  
המתוור. נשים  
לב שהוא Ch'ib  
להכיר את כל  
המשתמשים ולכן  
הוא מחזיק אותם  
ברשימה. כמו כן,  
בשליחת הודעה  
cols יקבלו אותה  
חו'ץ מזה של  
אותה.

```

class UserImpl extends User {
    public UserImpl(ChatMediator med, String name) { super(med, name); }
    @Override
    public void send(String msg) {
        System.out.println(this.name+": Sending Message="+msg);
        mediator.sendMessage(msg, user: this);
    }
    @Override
    public void receive(String msg) {
        System.out.println(this.name+": Received Message:"+msg);
    }
}

```

השלב האחרון  
יהי למשמש  
את ה `user`.

```

ChatMediator mediator = new ChatMediatorImpl();
User user1 = new UserImpl(mediator, name: "Jason");
User user2 = new UserImpl(mediator, name: "Jennifer");
User user3 = new UserImpl(mediator, name: "Lucy");
mediator.addUser(user1);
mediator.addUser(user2);
mediator.addUser(user3);
user1.send( msg: "Hi All");

```

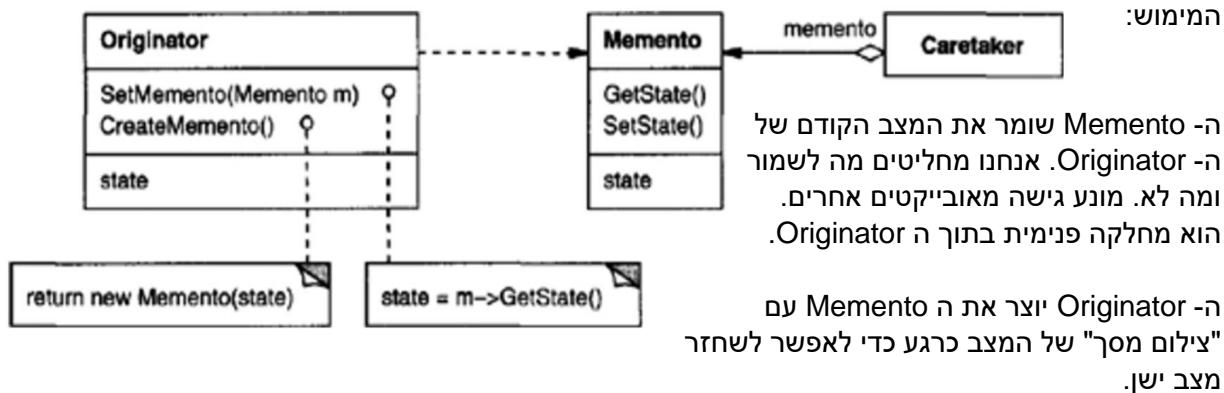
צד הלקוח:  
Jason שולח  
הודעה לכלם  
าง'ניפר ולוסי  
מקבלות אותה.

## Memento .18

המטרה בפטון היא לשמר מצב של אובייקט כדי שבעתיד יוכל לחזור במצב הקודם שלו. המימוש נעשה באופן שהמצב השמור לא נגיש לאובייקט. לא שובר אינטנסיביות. סוג של בקשת undo במצב של אובייקט שכבר עבר שינוי מסוים.

דוגמאות:  
 .word ctrl+Z  
 .rollback transaction in DB  
 שמירת מצב התקדמות במסחך מחשב.

בעיה היחידה בפטון הוא ששמירת המון מצבים או שמירת מצב כל המערכת יכול לפגוע ביציעים.



ה- Caretaker אחראי על שמירתו של ה Memento. מחלוקת עזר שדרוכה אפשר לשמר ולחקות מצבים שונים דרך ה Memento עצמו.

ה- Memento עצמו פשוט. רק ה Originator יכול לעשות את אותם פעולות.

דוגמת קוד: text editor

```

public class FileWriterUtil { //Originator
    private String filename;
    private StringBuilder content;
    public FileWriterUtil(String file) {...}
    @Override
    public String toString() { return this.content.toString(); }
    public void print() { System.out.println(this + "\n"); }
    public void write(String str) { content.append(str); }
    // creates the memento
    public Memento save() {
        return new Memento(this.filename, this.content);
    }
    // restore into its earlier state
    public void undoToLastSave(Object obj) {
        Memento memento = (Memento) obj;
        this.filename = memento.filename;
        this.content = memento.content;
    }
}
  
```

נתחיל עם המחלקת הנ"ל שמייצגת קובץ לכתיבה. נשים לב שהמתודה save אנחנו יוצרים את ה Memento עם המצב הנוכחי. וב- undo אנחנו משוחזרים את המצב לשיבלו כפרמטר.

```

private class Memento {
    private String filename;
    private StringBuilder content;

    public Memento(String file, StringBuilder content) {
        this.filename = file;
        this.content = new StringBuilder(content);
    }
}

```

```

public class FileWriterCaretaker {
    private Object obj;
    public void save(FileWriterUtil fileWriter) {
        this.obj = fileWriter.save();
    }
    public void undo(FileWriterUtil fileWriter) {
        fileWriter.undoToLastSave(obj);
    }
}

```

```

FileWriterCaretaker caretaker = new FileWriterCaretaker();
FileWriterUtil fileWriter = new FileWriterUtil("data.txt");
fileWriter.write(str: "First Set of Data:");
fileWriter.write(str: "\nMyra\nLucy\n");
fileWriter.print();
caretaker.save(fileWriter);
fileWriter.write(str: "Second set of data:\nJason\n");
fileWriter.print();
caretaker.undo(fileWriter);
fileWriter.print();

```

השלב הבא זה ליצור את המחלקה **Memento** שנמצאת בתוך ה-**.FileWriterUtil**. ניתן להראות שהוא מחזיק את אותו שדות כי הוא רוצה לשמר על כל המידע.

השלב האחרון הוא ליצור את ה-**Caretaker**. ניתן להראות שהוא לא מכיר את ה-**Memento**. רק דרכו אנחנו יכולים לעשות **save** ו- **undo** לאובייקט **Originator**.

צד הלקוח:

אפשר לראות שאנחנו פותחים קובץ ומתחילה לכתוב בו. שומרים את השינויים וממשיכים לכתוב שוב. אחרי זה עושים סוף ומה שקרה כאן אנחנו

משחזרים את הנתונים שהיינו נשמרו בפעם האחרונה את תוכן הקובץ.

## Observer .19

הפתרן מגדיר קשר תלותי של אחד לרבים, כשאם מצבו של אובייקט משתנה כל האובייקטים התלויים בו יעדכנו בהתאם.  
אנו רוצים להפחית את ה coupling בין ה- `Notifier` לבין ה- `Notifyee`.  
ה- `observers` רושמים את עצם המשקיפים / מאזורים לאובייקט מסוים וכשהאובייקט משתנה הם אוטומטיות מודעים לזה.  
אם הם מאבדים עניין באותו אובייקט הם יכולים לבטל את הרישום שלהם.

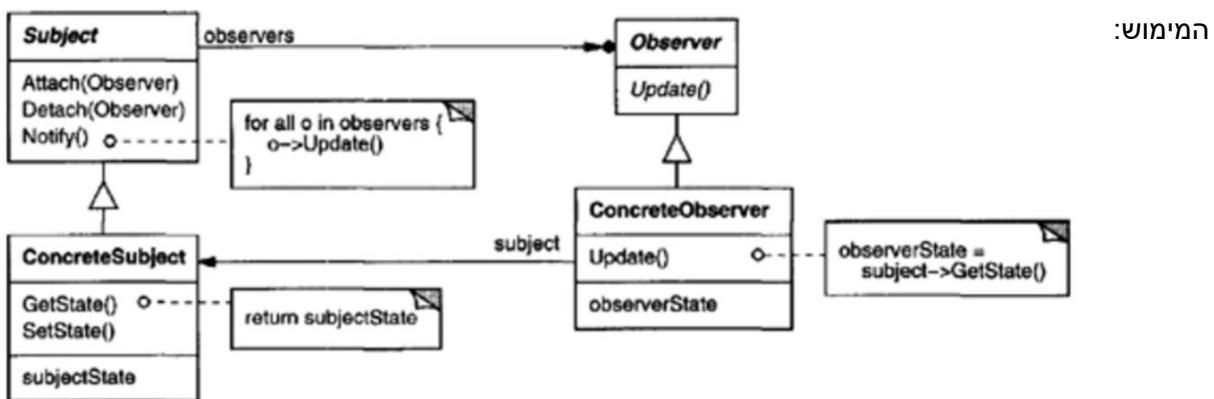
**דוגמא:**  
להירשם כמנוי לעיתון. בכל פעם שיש עיתון חדש אתה מקבל אותו ומתוודע על כך, כאשרת מבטל את הרישום אז מפסיקים עם זה.

מתי משתמש בפתרן?

כששוני באובייקט אחד מצריך שינויים בכמה אובייקטים.

כשכמה אובייקטים תלויים במצבו של אובייקט אחד.

כשנרצה שאובייקט אחד יודיע לאובייקט אחר מבל' היכולת לדעת מי הוא.



\*\* `java.util.Observable` מימושו את ה class `Observable` אבל לא נלמד עליו יותר מדי מכיוון שלא משתמשים בו הרבה, כי חיברים לרשות מימנו ולכך לא יוכל עוד לרשות מאחרים, ואין אינטראפיס שלו.

```

interface Subject {
    void register(Observer obj);
    void unregister(Observer obj);
    void notifyObservers(); //ALL
    Object getUpdate(Observer obj);
}

```

```

interface Observer {
    void update();
    void setSubject(Subject sub);
}

```

```

class MyTopic implements Subject {
    private List<Observer> observers;
    private String message;
    private boolean changed;
    public MyTopic() { observers = new ArrayList<>(); }
    @Override
    public void register(Observer obj) {
        if (obj == null)
            throw new NullPointerException("Null Observer");
        if (!observers.contains(obj)) observers.add(obj);
    }
    @Override
    public void unregister(Observer obj) { observers.remove(obj); }
    @Override
    public void notifyObservers() {
        if (!changed) return;
        this.changed = false;
        for (Observer obj : observers) {
            obj.update();
        }
    }
    @Override
    public Object getUpdate(Observer obj) { return this.message; }
    public void postMessage(String msg) { //trigger notifications
        System.out.println("Message Posted to Topic:" + msg);
        this.message = msg;
        this.changed = true;
        notifyObservers();
    }
}

```

דוגמא כללית:  
 תחילה נבנה את ה Subject עם יכולת לרשום או לבטל רישום של Observer. ברגע, יכולת להודיע לכלום על שינוי ולקבל את אותו עדכון.

אחר כך נבנה את ה Observer עם מетодה שלעדכן אותו כשקורה משנה אצל ה Subject שרשום אצל.

השלב השלישי זה לבנות אתה מימוש ל Subject מכל רשימה של .observers.

המטרה: כשה message משתנה נודיע לכל מי שמאזין.

```

class MyTopicSubscriber implements Observer {
    private String name;
    // not required
    private Subject topic;
    public MyTopicSubscriber(String nm) { this.name=nm; }
    @Override
    public void update() {
        //could take data of Subject
        String msg = (String) topic.getUpdate( obj: this );
        if(msg == null){
            System.out.println(name+": No new message");
        }else{
            System.out.println(name+": Consuming message::"+msg);
        }
    }
    @Override
    public void setSubject(Subject sub) { this.topic=sub; }
}

```

השלב האחרון  
הוא לבנות  
מיימוש ל  
.Observer  
אפשר לראות  
שיש לו עבודה  
לעשות  
כשעדכנו אותו  
אם מצבו של  
topic ה-  
השתנה.

```

MyTopic topic = new MyTopic();
Observer obj1 = new MyTopicSubscriber( nm: "Obj1" );
Observer obj2 = new MyTopicSubscriber( nm: "Obj2" );
Observer obj3 = new MyTopicSubscriber( nm: "Obj3" );
topic.register(obj1);
topic.register(obj2);
topic.register(obj3);
obj1.setSubject(topic);
obj2.setSubject(topic);
obj3.setSubject(topic);
obj1.update();
topic.postMessage( msg: "New Message" );

```

צד לקוח:

```

Obj1:: No new message
Message Posted to Topic:New Message
Obj1:: Consuming message::New Message
Obj2:: Consuming message::New Message
Obj3:: Consuming message::New Message

```

פלט:

## State .20

הפתרן מאפשר לאובייקט לשנות את התנהוגותו כאשר מצבו הפנימי משתנה. ההתנהוגות משתנה בזמן ריצה לפי המצב.  
בדרך כלל אנחנו משתמשים ב `switch case` כנדרצה לבצע פעולות שונות תלוין במצב מסוים, ובעזרת פתרן זה אנחנו מסירים את הלוגיקה המותנת בהם.

דוגמאות:

תמרור: עוצר אם אודום, תישע אם ירוק, תאט...  
תקשורת TCP שלו כמה מצבים: מחובר, מאזין/מחכה לחבר, וסגור לחבר.

מתי משתמש בפתרן?

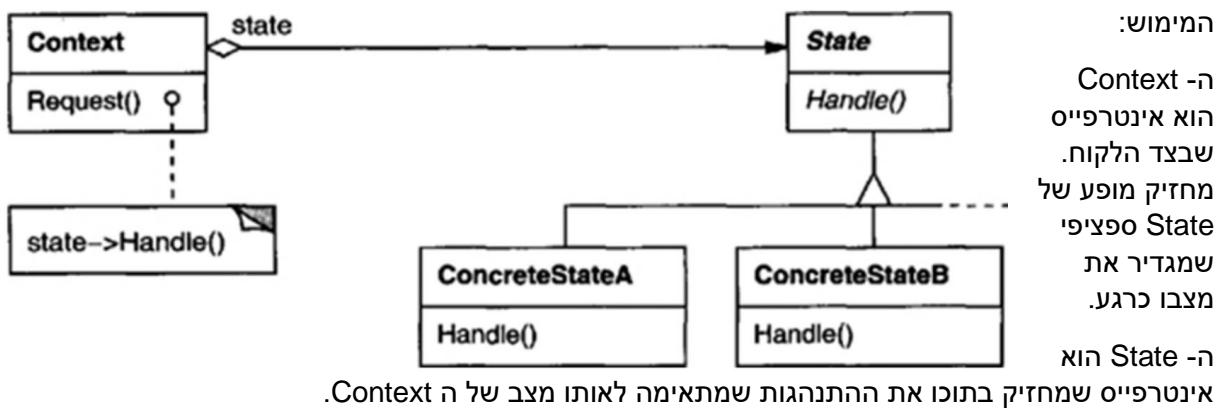
כשהתנהוגות של אובייקט תלויות במצבו, והוא צריך לשנות התנהוגות שלו בזמן ריצה.  
כשייש המונן תנאים שתלויים במצבו של אובייקט.

יתרונות:

מכнос את כל התנהוגיות הקשורות למצב מסוים לאובייקט אחד. שיפור ב cohesion.  
מסיר את הבלוקים הגדולים של `if-else` או `switch case`. הקוד הופך ליותר גמיש ותחזוקתי.

חיסרונות:

מגדיל את כמות המחלקות והאובייקטים.



```

abstract class RemoteControl { //State
    public abstract void pressSwitch(TV context);
}

class On extends RemoteControl {
    @Override
    public void pressSwitch(TV context) {
        System.out.println("I am already On."
            + "Going to be Off now");
        context.setState(new Off());
    }
}

class Off extends RemoteControl {
    @Override
    public void pressSwitch(TV context){
        System.out.println("I am Off."
            + "Going to be On now");
        context.setState(new On());
    }
}

```

דוגמת קוד-שלט טלוויזיה:

תחליה נבנה את השלט  
שיציג את מצבה של  
הטלוויזיה. יש לו מתודה  
של לחיצת כפתור  
שאמורה להדילק ולכבות  
את הטלוויזיה שהוא  
מקבל כפרמטר.

לחולקה שני שימושים  
בדיווק כשני מצביו של ה-  
.Context

השלב השני יהיה ליצור  
Context את הטלוויזיה ה-  
שלנו. אשר מחזיקה מופע  
של מצבה הנוכחי  
(השלט).

אפשר לראות שהטלוויזיה  
עשה דלקתיה שמתודת  
של לחיצת הכפתור היא

בעצם מפעילה את המתודה שדיברנו עלי' של השלט.

```

public class TV { //Context
    private RemoteControl state;
    public RemoteControl getState() { return state; }
    public void setState(RemoteControl state) { this.state = state; }
    public TV(RemoteControl state) { this.state=state; }
    public void pressButton() { state.pressSwitch(context: this); }
}

```

```

Off initialState = new Off();
TV tv = new TV(initialState);
tv.pressButton();
tv.pressButton();

```

צד הלקוח:

כך אנחנו בעצם יוצרים  
טלוויזיה שהיא בהתחלה  
כוביה ואז מדליקים אותה  
ואז מכבים אותה.

\*\* ניתן לשים לב שוכנו לעצמו לגמר בדיקות if-else על אם המצב הוא on או off.

## Strategy .21

הפתרן מגדיר משפחה של אלגוריתמים, כל אחד מאוחסן באובייקט בנפרד. האלגוריתמים עושים את אותם דברים, רק השימוש שונה. ובכך אפשר להחליף אלגוריתם בקלות. אנחנו יכולים לבחור את ההתנהגות של אלגוריתם בזמן ריצה.

דוגמאות:

אסטרטגיה של קבוצת כדורים יכולה להיות תקיפה או הגנה.  
אחסון נתונים בשני מקומות, כשהראשון הتمלא נמלא בשני.  
(`Comparator.Comparator`) נעזר ב `Collection.sort()` עצם אסטרטגיית המילוי.

### Strategy vs. State

נחשב על ה `Strategy` כעל מספר מחלקות ירושות שמחילות על מימוש של שלבים באלגוריתם.  
נחשב על ה `State` כאל תחליף לleshim המונן תנאים בהקשר מסוים. כאן אנחנו משתמשים בدلגציה.  
דבר אחד שהוא שונה בשנייהם הוא שב `State` בתוך `Context` יש לנו מופיע של ה `state` וב- `Strategy` אנחנו מעבירים את האסטרטגיה כפרמטר למתחודה בתוך ה `Context` ולא מחזיקים אותו בפנים.

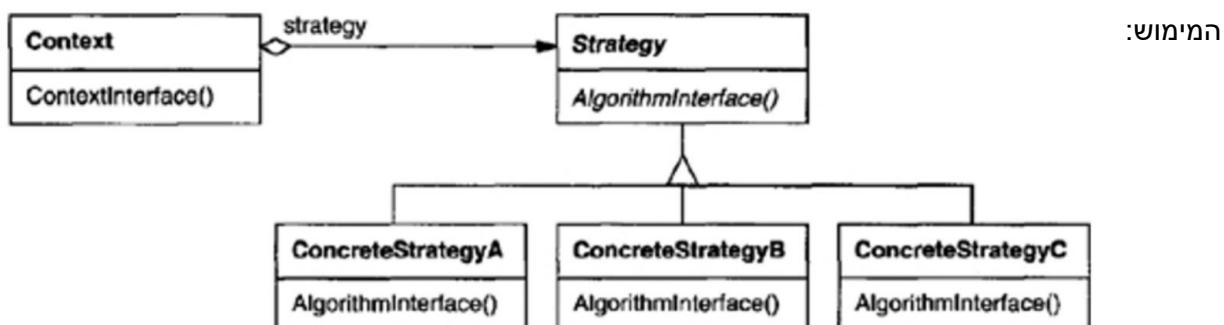
יתרונות נוספים:

קל להוסיף אסטרטגיות חדשות.  
אפשר ללקוח לבחור את האסטרטגיה שהוא רוצה מבלי להיעזר ב `switch case` או `if-else`.  
אפשר לunit-testning כי כל אסטרטגיה נמצאת באובייקט אחר ואפשר לבדוק כל אחד בנפרד.

חיסרון: הלוקח חייב לדעת על סוג האסטרטגיות שנומכחים.

מתי משתמש בפתרן?

צריכה לבחור או לשנות אלגוריתם בזמן ריצה.  
שאלת אלגוריתם משתמש בנתונים שנרצה שישו חסומים למשתמש. כך גם להסתדרת מרכיבות של אלגוריתם מהלוקה.  
כשיש לנו `switch case` או `if-else`.



ה- `Strategy` הוא אינטראפייס משותף לכל האלגוריתמים שאנו נומכחים בהם.  
ה `Context` משתמש בהם.

ה- `Context` שומר ממופע או מקבל כפרמטר את ה `ConcreteStrategy` שתבחר את ה `ConcreteStrategy` או `Context` אל ה `ConcreteStrategy`.

```

interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardStrategy implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;
    public CreditCardStrategy(String nm, String ccNum, String cvv, String dte) {
        name = nm;
        cardNumber = ccNum;
        cvv = cvv;
        dateOfExpiry = dte;
    }
    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid with credit/debit card");
    }
}

```

דוגמת קוד –  
תשלום לחנות  
אינטרנטית:  
תחלת ליצור  
את אסטרטגיית  
התשלומים.  
איןטרפיס עם  
מתודה של  
לשלה.

למחלקה שני  
מיושמים:  
אפשר לשלם  
בקרטיים  
או אשראי או  
פייפל.

השלב הבא  
הוא לייצר את  
context ה  
שזה בעצם  
עגלת הקניות.

שבתוכה יש  
רשימה של  
מוצרים  
 items  
שלכל מוצר יש קוד ומחיר.

```

class ShoppingCart { //Context
    List<Item> items;
    public ShoppingCart() { this.items=new ArrayList<Item>(); }
    public void addItem(Item item) { this.items.add(item); }
    public void removeItem(Item item) { this.items.remove(item); }
    public int calculateTotal() {...}
    public void pay(PaymentStrategy paymentMethod){
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}

```

המתודה calculateTotal מחשבת את סכום מחירים של כל המוצרים.

במתודה pay הנקו משלם באמצעות אסטרטגיית תשלום כרצוננו. אנחנו מחשבים את הסכום  
הכול ועשויים דרגציה לתשלום למחלקה של PaymentStrategy.

```

ShoppingCart cart = new ShoppingCart();
Item item1 = new Item(upc: "1234", cost: 10);
Item item2 = new Item(upc: "5678", cost: 40);
cart.addItem(item1);
cart.addItem(item2);
cart.pay(new PaypalStrategy(email: "myEmail", pwd: "mypassword"));
cart.pay(new CreditCardStrategy(
    nm: "A.H", ccNum: "***", cvv: "123", expiryDate: "12/27"));

```

צד הלקוח:  
מייצרים  
עגלת קניות,  
מוסיפים שני  
 מוצרים,  
ומשלמים  
עליהם בשתי  
דרכים  
שונות.

## Template Method .22

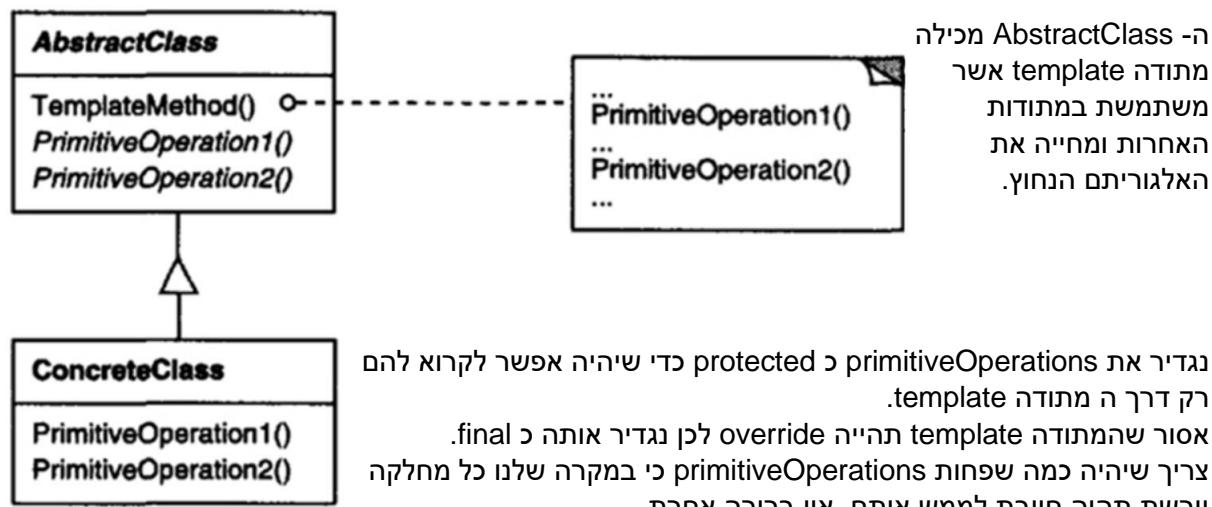
הפטן מגדיר שלד של אלגוריתם עבור פעולה מסוימת. כל מחלוקת תיצג שלב בטהילך. בעצם אנחנו בונים תבנית לאלגוריתם זהה בעצם מתודה שגדירה את האלגוריתם כקבוצה של שלבים.

דוגמא:  
הכנת פיצה תמיד תיראה אותו דבר פרט לשינויים קטנים תלויים ברגע הלקות.

מותי השתמש בפטן?  
כשרצה לתמוך במספר אלגוריתמים שונים רעונית אותו דבר, אבל שונים במימוש בכל שלב.  
כשרצה להימנע משכפול קוד.

המימוש:  
בדרך כלל מחלוקת ירושות קוראות למетодות בחלוקת האב. בפטן זהה מחלוקת האב קוראות למethodות אצל מחלוקת היירושות, נקרא "עקרון ההולוייד" – "אל תתקשר אלינו, אנחנו נתקשר אליו". ה- main template method נמצא בחלוקת האב, שהוא אבסטרקטית. בחלוקת האב יש למетодות רגילים, אבסטרקטיות, ו-.hooks

מתודה hook - מתודה שמוצהרת בחלוקת אבסטרקטית ויש לה מימוש ריק או default. היא נותנת את יכולת לחלוקת ירושת "להינעל" בנסיבות מסוימת באלגוריתם. המחלוקת הירושה יכולה להתעלם ממетодה זו.



```

abstract class HouseTemplate {
    public final void buildHouse() {
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built");
    }
    private void buildWindows() { //hook method
        System.out.println("Building Glass Windows");
    }
    public abstract void buildWalls();
    public abstract void buildPillars();
    private void buildFoundation() {
        System.out.println("Building foundation with cement");
    }
}

```

דוגמא- בניית בית  
בשלבים:

תחילה נבנה את המחלקה שאחראית על בניית אובייקט של בית. בתוכה המתודה `.template` היא `final`.  
נשים לב שהמתודה בין המתודות הוא הכרח.

למחלקה חזו שתי מימושים:  
- `WoodenHouse`  
`GlassHouse`

ممמשות בעצם את שתי המתודות האבסטרקטיות.

```

HouseTemplate houseType = new WoodenHouse();
houseType.buildHouse();

houseType = new GlassHouse();
houseType.buildHouse();

```

צד הלוקות:

## Visitor .23

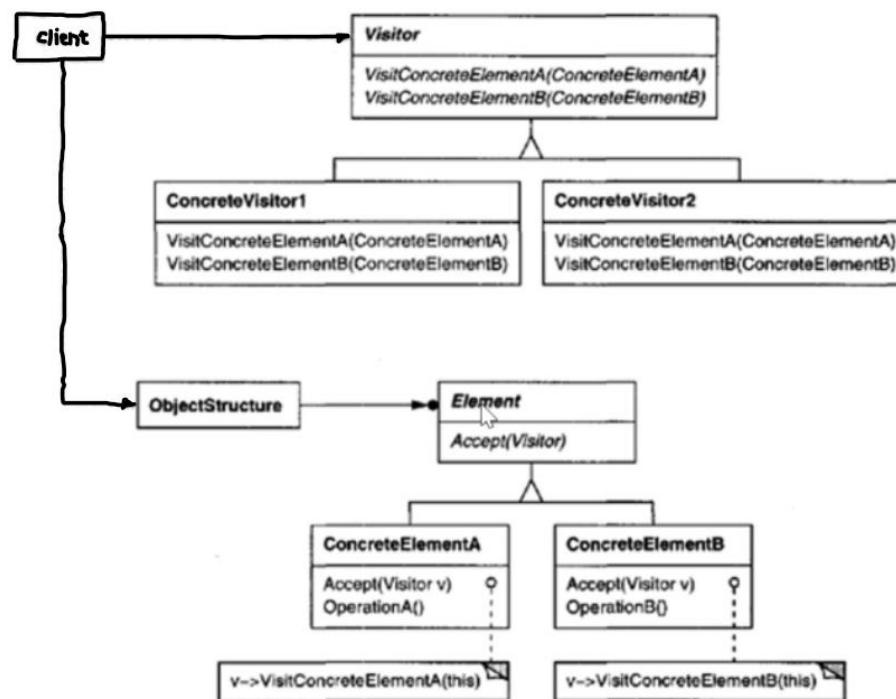
הפטן מייצג פעולה לביצוע על אובייקט בתוך קבוצה של אובייקטים. הפטן מספק פעולה חדשה על אובייקט מוביל לשנות את המחלקה שלו, בזמן ריצה. מה שתורם להפחיתה ה coupling בין הפעולה החדשה למחלקה של האובייקט. אפשר להבהיר פעולה מאובייקט אחד למחלקה אחרת. הפטן מאד שימושי כשותממשים עם APIs ציבוריים.

**דוגמא:**

עגלת קניות באינטרנט. אנחנו יכולים לחשב את הסכום הכללי של הקניות. אנחנו מעבירים את חישוב הסכום של מוצר ייחד למחלקה אחרת בעזרת הפטן זה.

**מתי נשתמש בפטן?**

כשיש לנו קבוצה של אובייקטים אינטראקטיביים שונים, ואני רוצה לבצע פעולה על כל אחד מהם תלוי המחלקה הSPECIFIC שלהם. אז כדי שהקוד יראה יותר נקי ונשימוש בפטן זה ונגידיר מחלקה משותפת לכל האובייקטים עם פעולות משותפות. כשאני רוצה להפחית את ה coupling הקשור מהאובייקט הקולט שאני משתמש בו. כשיש לי קבוצה של אובייקטים משותף לרבות אפליקציות, נשימוש בפטן בכר שנוסיף פעולות חדשות לכל אפליקציה בנפרד.



**השימוש:**

ה- **Visitor** הוא אינטראקטיביס שמאגדיר את פעולות הביקור לכל המחלקות שצריכות תמייה בה.

ה- **Element** הוא אינטראקטיביס נוסף שמאגדיר את פעולות הקבלה אשר מקבלת visitor כפרמטר.

ה- **ObjectStructure** הוא בעצם הקבוצה של האובייקטים.

דוגמת קוד - עגלת קניות:

המטרה בגלות סכום כולל של כל המוצרים מוביל להוסיף את הfonkציונליות הzzאת למחקות שלנו ובהנחה שאפשר לגלוות את הסכום של כל מוצר בנפרד.

```
interface ItemElement {  
    int accept(ShoppingCartVisitor visitor);  
}
```

תחילה נוצר אינטראפיס של Element, המוצר, עם מתודה שמקבלת visitor במטרה "לבקש" אותו ושלוח את המופיע של המחלקה ספציפית שלו, המוצר הספציפי.

המתודה מחזירה int כי המטרה שלנו להציג את המחיר של המוצר.

לאינטראפיס שני מימושים Book ו- Fruit. לכל אחד מהם יש המתודה .int getPrice()

```
@Override  
public int accept(ShoppingCartVisitor visitor)  
{ return visitor.visit(book); }
```

כך בעצם נמשך את המתודה של ה accept שקוראת למетодה visit ספציפית.

```
interface ShoppingCartVisitor {  
    int visit(Book book);  
    int visit(Fruit fruit);  
}
```

השלב הבא הוא לבנות את האינטראפיס של ה Visitor אשר יש לו מתודת visit ככמויות המוצרים במערכת. בעצם overloading על המתודה עם מוצרים שונים כפרמטר.

אנחנו רוצים זאת כי לכל מוצר נחשב את מחירו בצורה שונה. דוגמא: עבור ספר נחזיר את מחירו פחות 5 שקלים, ועבור פרי נחזיר מחיר לקלילו כפול משקל.

כעת הלקוח יכול ליצור את הfonkציות החדשה שרצה שהוא שהיא לקבל מחיר כולל של כל המוצרים השונים מוביל לדעת מהם:

```
private static int calculatePrice(ItemElement[] items) {  
    ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();  
    int sum = 0;  
    for(ItemElement item : items){  
        sum = sum + item.accept(visitor);  
    }  
    return sum;  
}  
  
public static void main(String[] args) {  
    ItemElement[] items = new ItemElement[]{book1, book2, fruit1, fruit2};  
    int total = calculatePrice(items);  
}
```

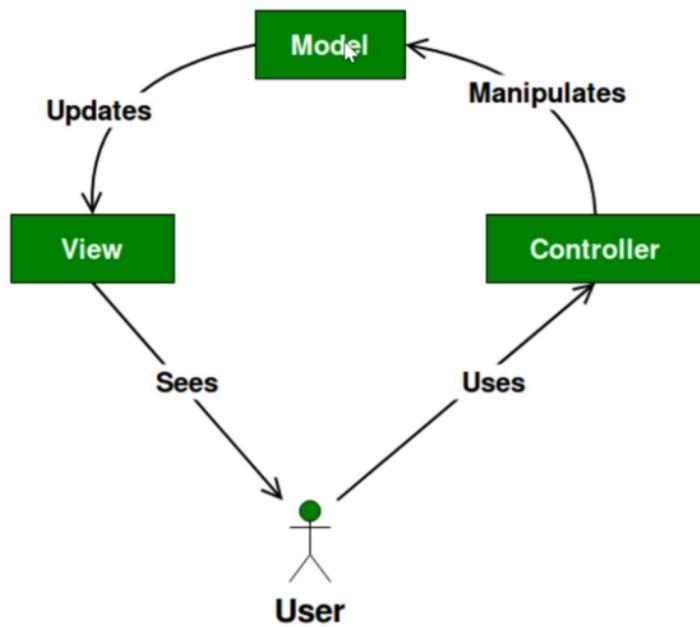
# MVC-Model View Controller- Overview

פטרן זה מציג את המערכת בשלושה חלקים נפרדים: מודל נתונים (model), תצוגה (view), ובקר (controller). דרך מוכרת לארגן מערכת קוד שלמה.  
שלושת החלקים לא תלויים אחד בשני.

ה- Model אחראי על אחסון הנתונים והלוגיקה של מה שצריך לבצע.

ה- View אחראי על הציגת הנתונים מה model אל המשתמש, בדרך צו או אחרת. הוא לא מכיר את ה model. הוא מקבל את כל המידע שהוא צריך דרך ה controller. ה view לא ידע מה הנתונים אמורים או מה המשתמש יכול לעשות איתם.

ה- Controller הוא המתווך שמי בין ה view ל model. הוא מודע ליריעים שהופעלו בעקבות התצוגה ופועל בהתאם וקורא למתחדות המתאימות אצל ה model. ה view מתוודע באופן אוטומטי כשה controller מופיע את עובdotנו.



בהתמונה ניתן לראות את כל מערכות  
היחסים:  
הלקוח רואה את ה view ומשתמש  
ב controller. controller מעודכן את ה view  
באמצעות ה model.

דוגמא מהמציאות: תכנון ארוחת חג  
ההודייה

ה- model הוא המקור המלא במצרכים.  
ה- controller הוא המתכוון, שמחלית מה  
להוציא קודם מהמקור, ואיר לאסוף את  
הדברים ביחיד כדי ליצור ארוחה.

ה- view הוא שולחן האוכל, הצלחות,  
והסכום.

יתרונות:  
מספר מתכנתים יכולים לעבוד על חלקים אחרים של המערכת מבלי להיות תלויים אחד  
בשני.  
קל להוסיף תצוגת חדשות.  
הופך את פיתוח המערכת ליותר מהיר.  
תומך בתכונות מקביל.  
מפחית את coupling בין החלקים. אם נדרש לשנות משהו בחולק אחד אין גע רק בו.  
ש דוגמת קוד לפטרן בריפוזיטורי.

קישור לריפזיטורי בGIT האב של כל הפתרונות שלמדנו עד עכשוו:

[https://github.com/AndrioHanania/23\\_Design\\_Pattern\\_Examples](https://github.com/AndrioHanania/23_Design_Pattern_Examples)