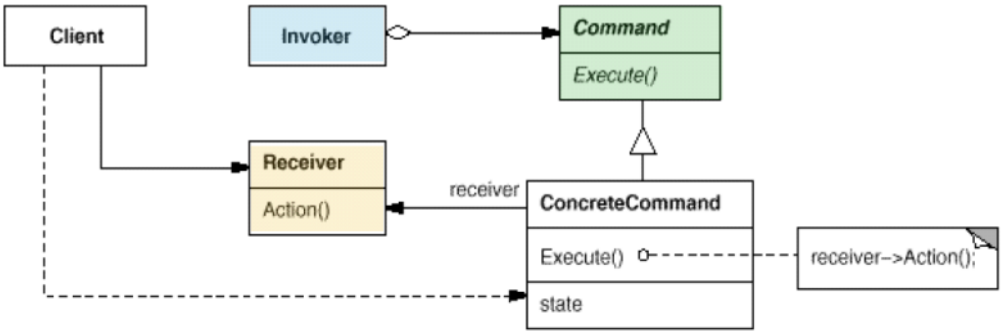
הרצאה 13 - Behavioral Patterns

יום שישי 08 ינואר 2021 12:25

14.1 Behavioral Patterns: Command

אנו רוצים ליצור מאפיין פולימורפי לכל הפקודות בתוכנית וכך לאגד אותם תחת אובייקט מסוג command. נרצה להפריד בין מי שקורא ל-command (invoker) לבין הפקודה הספציפית (specific command) ומי שמבצע/ מבוצע עליו (Receiver) .
כמובן שנרצה להשתמש באובייקטים ולא במתודות סטטיות כמו שראינו בעבר שכן באובייקטים ניתן לעשות דברים שלא ניתן לעשות בעזרת מתודות סטטיות וכד'. נתבונן בתרשים העיצוב של תבנית זו -



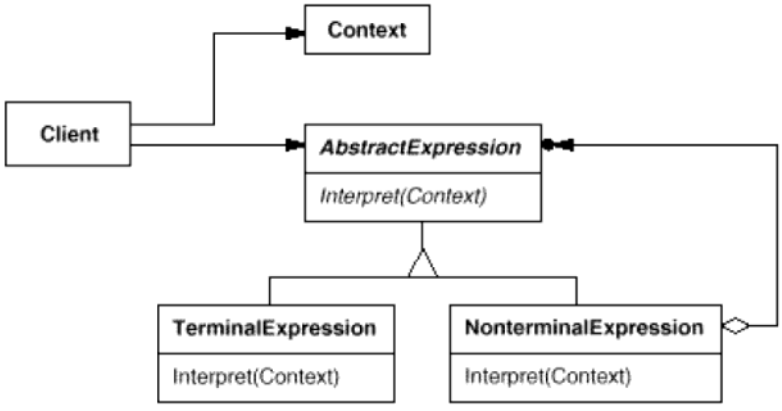
הלקוח מכיר את ה-Receiver, מי שמבצע את הפעולה או שהפעולה מתבצעת עליו. ה-Receiver מקבל את הפקודה מ-Concrete Command, שיורשת את Command. ה-Invoker, מי שקורא ל-command, מכיל אובייקט מסוג command וקורא למתודת ה-execute.
נתבונן בדוגמא למכניזם של בקר (controller) -

- הבקר יממש את תבנית העיצוב singleton כי נרצה רק אחד כזה
- כאשר רוצים להפעיל פקודה, נקבל מפתח ובעזרתו נמצא או ניצור את אובייקט הפקודה המתאים
- אובייקטים קיימים מאוחסנים בטבלת גיבוב (בדומה ל-flyweight pattern)
- אובייקטים חדשים נוצרים ב- Command Poll (בעזרת builder pattern)
- אובייקט הפקודה מוכנס לתור עדיפויות
- תהליכון שונה מוציא פקודות מהתור (כל עוד הוא לא ריק) ומבצע את הפקודה ב-thread בעזרת thread pool

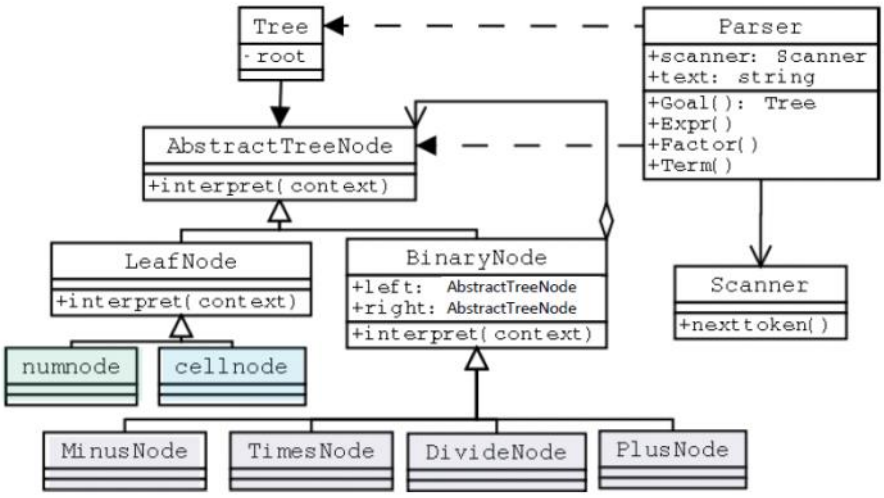
זה תיאור של מנגנון שמשתמשת בהרבה תבניות עיצוב. נעיר שאין צורך לכפות בכוח תבניות עיצוב אם לא נצרך, אך כדאי לתכנן נכון על מנת שבמידת הצורך נוכל להגדיל ולהרחיב את הקוד בקלות.

14.2 Behavioral Patterns: Interpreter

נתבונן בתרשים של תבנית העיצוב -

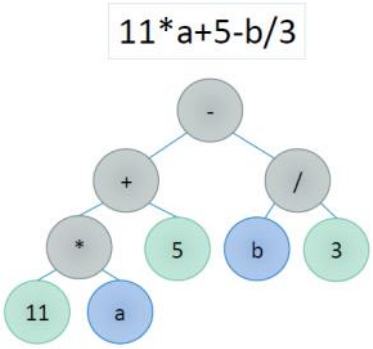


הלקוח מכיר תוכן (context) ואת המחלקה המופשטת Abstract Expression שיש לה מתודת Interpret לתוכן. את המחלקה המופשטת יורשות שתי מחלקות - Terminal Expression שהיא מחלקה "סופית" שמבצעת פירוש (Interpretation) לתוכן שקיבלה. ובנוסף המחלקה None terminal Expression, מחלקה שיכולה לבצע פירוש וכן להכיל בתורה Abstract Expression. בצורה כזו ניתן לקבל ביטויים מורכבים יחסית ולבצע חישוב שלהם.
נתבונן בדוגמא -



יש לנו עץ, שמחשב ביטויים מתמטיים. המחלקה המופשטת Abstract Tree Node, מייצג אובייקט שהוא קודקוד (או עלה) בעץ. למחלקה זו שתי מחלקות יורשות - Leaf Node (עלים), שיש לה שתי מחלקות יורשות גם כן - num Node (מייצג מספר), Cell Node (מייצג משתנים). מחלקה שנייה יורשת היא Binary Node ביטויים בינאריים - המחלקות היורשות הן Minus, Times, Divide, Plus .

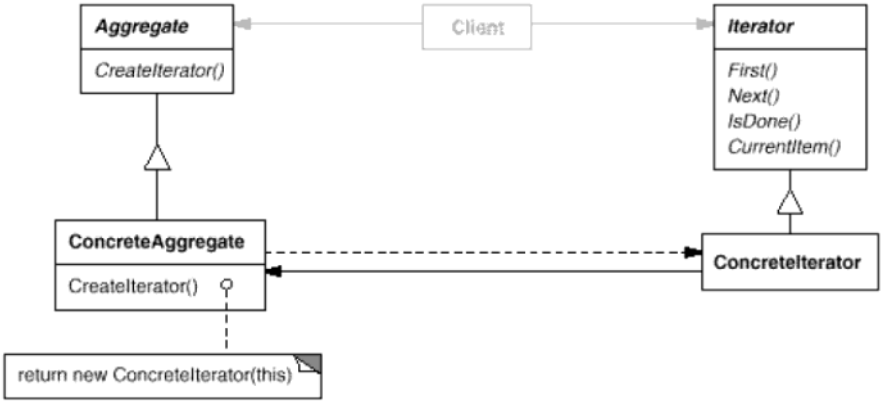
כעת נתבונן בביטוי ספציפי וכיצד הוא מיוצג -



נשים לב ש-1,5,3 אובייקטים מסוג numnode, a,b מסוג cellnode. האופרטורים /,-,+,* הינם binaryNode ומחולקים למחלקות בהתאמה. נשים לב שהרקורסיה בעץ היא קודם כל חישוב העץ השמאלי, לאחר מכן הימני, ולבסוף הפעולה המתאימה (בדוגמא שלנו חיסור) בין השניים.

14.3 Behavioral Patterns: Iterator

תבנית העיצוב הזה אומרת שלכל מבנה נתונים יש Iterator, וכדאי שה-API יהיה זהה (על מנת להקל על העבודה). מטרת תבנית העיצוב הזו היא כימוס, החבאת הדרך בה עובד מבנה הנתונים. נתבונן בתרשים של תבנית העיצוב -



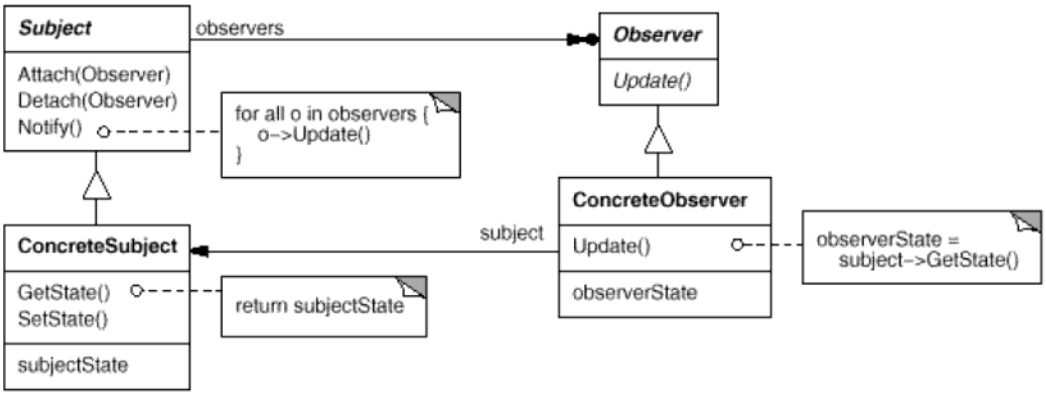
הלקוח מכיר את הממשק Aggregate (צבירה) שהיא מבנה הנתונים. לממשק זה מחלקה יורשת ספציפית, Concrete Aggregate שהיא יוצרת Concrete Iterator שמתאים לעבודה על מבנה הנתונים הספציפי, Concrete Aggregate.
ה-Concrete Iterator יורש מהממשק Iterator את הפעולה הנדרשות וממש בצורה מתאימה אל מול מבנה הנתונים הספציפי.

נשים לב, שכאשר ממשים או עובדים עם Iterator, צריך לקחת בחשבון מי ממש את האלג'? לדוגמא מעבר לא קודקודים בעץ יכול להתבצע במספר דרכים, מי בוחר באיזו דרך? בנוסף צריך לבדוק כמה עמיד בשינויים (Robust) ה-Iterator? כלומר, אם

מבצעים מעבר על רשימה ומחליטים למחוק איבר מסוים, האם נקבל חריגה או שנמשיך לעבור על שאר האיברים?

14.4 Behavioral Patterns: Observer

נתחיל בדוגמא - קיים מידע (subject) שניתן להציג אותו במספר דרכים. נרצה שבכל פעם שהמידע משתנה, גם התצוגות השונות מתעדכנות. כלומר, המידע צריך לעדכן את הצופים (observers) שלו שהמידע השתנה והם יבחרו אם לעדכן. מדוע שהמידע רק מעדכן שהיה שינוי ולא אומר מה היה השינוי? בתבנית זו הצופים מחליטים מתי לעדכן שכן יכול להיות שהמידע השתנה עשר פעמים בשנייה, ואין באפשרות הצופים לעדכן את עצמם כל כך הרבה. לכן הם יחליטו מתי הם מבצעים את העדכון. לאחר הדוגמא, נתבונן בתרשים תבנית העיצוב -



המחלקה Subject עם שלוש מתודות (Attach, Detach, Notify) שמאפשרות למחלקות מסוג Observer להיות צופים ומוכנים לקבלת שינויים. למחלקה Subject מחלקות יורשות Concrete Subject. ברגע שמתרחש שינוי ב Concrete Subject, היא תודיע לכל הצופים בעזרת מתודת Notify. כעת הצופים (Concrete Observer) יודעים שהתרחש שינוי ובוחרים בעצמם האם הם רוצים להפעיל את המתודה subject-> Get State.

כאשר מבצעים עדכון, צריך שיהיו שתי פרמטרים - הראשון, ממי הגיע העדכון , והשני, אודות, מה בעצם קרה.

14.5 Behavioral Patterns: Observer Example in #C

כעת נתבונן בדוגמא בשפת #C. בשפת #C תבנית עיצוב זו הייתה כל כך שימושית עד שהטמיעו אותה בשפה ממש. ראשית נראה הגדרה בשפה שנקראת delegate. טיפוס מסוג delegate מזכיר קצת מצביע לפונקציה בשפת C או ++C. הוא מחזיק בתוכו reference למתודה בעלת אותו מספר פרמטרים. לדוגמא -

- public void f(){...};
- public void g(){...};
- public **delegate** void myFunc();

x הוא אובייקט מסוג delegate של פונקציה ללא פרמטרים. הפונקציות f,g עונות בדיוק על הגדרה זו. בעזרת אופרטור ההשמה ניתן להשים את f או g ב-x וע"י הפעלה של ()x נפעיל את המתודה שנמצאת ב-x.

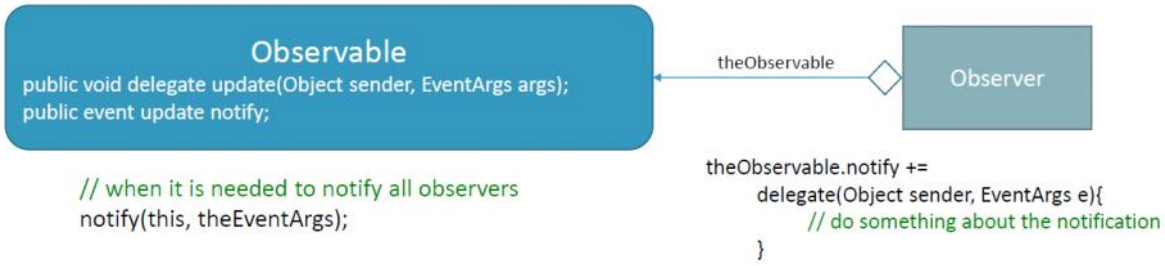
```
myFunc x;
x=f;
x(); // activate f()
x=g;
x(); // activate g()
```

כמו שקיים משתנה מסוג delegate קיים משתנה מסוג event -
כעת המשתנה x הוא מסוג event, במשתנה מסוג זה לא קיים אופרטור ההשמה אלא רק =+ או =-. היתרון הוא שניתן להשים יותר ממתודה אחת וע"י הפעלה של x, נוכל להפעיל גם את f ו-g ביחד. אמנם גם ב-delegate ניתן להעמיס יותר ממתודה אחת אבל הוא לא בטיחותי שכן ניתן למחוק בטעות ע"י אופרטור ההשמה את כל המתודות שהחזקנו במשתנה מסוג delegate.

```
event myFunc x;
x+=f;
x+=g;
x(); // activate f() and g()
x-=f;
x(); // activate only g()
```

כעת נעבור לראות כיצד delegate ו-event עוזרים לנו בתבנית העיצוב Observer. תהיה לנו מחלקה מסוג Observable (ניתן לצפיה - כלומר מחלקה שצופים בה ומעדכנים את השינויים), שתחזיק בתוכה מתודה delegate update שמקבל שתי פרמטרים - שולח, ואודות (כמו שהזכרנו לעיל). בנוסף נגדיר משתנה מסוג event update שנקרא לו notify. אל ה-notify נוכל להוסיף מתודות שיופעלו ברגע של עדכון.

המחלקה Observer (הצופה) מחזיקה אובייקט מסוג ה-Observable ומוסיפה לו פונקציה שתרצה להפעיל ברגע שמתבצע שינוי באובייקט. נציג תרשים שממחיש זאת -



כעת נעבור לדוגמא מורכבת יותר. נצרף את הקוד ולאחר מכן נסביר מה הוא עושה -

```
class AlarmClock {
    public Boolean stop;

    public delegate void whatToDo(String time);
    public event whatToDo customEvent;

    public void start()  {
     new Thread(
         delegate() {
        while (!stop) {
          String time = DateTime.Now.ToString("HH:mm:ss tt");

          customEvent(time);

          Thread.Sleep(1000);
        }
      }
     ).Start();
    }
}
```

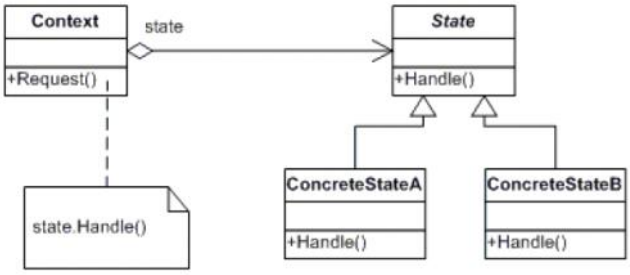
```
static void Main(string[] args) {
    AlarmClock ac = new AlarmClock();

    ac.customEvent += delegate(String time) {
        if (time.Equals("18:10:00 PM")) {
            Console.WriteLine("hello world!");
        }
    };
    ac.customEvent += delegate(String time) {
        Console.WriteLine(time);
    };

    ac.start();
    Thread.Sleep(3*60*1000);
    ac.stop = true;
    Console.ReadKey();
}
```

We have added 2 event handlers

הצד השמאלי הוא מחלק של שעון מעורר. במחלקה זו יש מתודה delegate הנקראת whatToDo וכן משתנה מסוג event whatToDo הנקרא customEvent.
הצד הימני הוא מימוש אפשרי של main. אנו יוצרים שעון מעורר חדש, ac. ל-ac.customEvent אנו מוסיפים שתי מתודות, אחת מדפיסה את הזמן והשנייה מדפיסה hello world כאשר השעה היא 18:10 בדיוק.

14.6 Behavioral Patterns: State

מה קורה כאשר יש לנו את אותה התנהגות אך ההתנהגות משתנה לפי מצב (state) מסוים? ניתן להשתמש בתנאי if, או switch case. בתבנית עיצוב זו מנצלים את הפולימורפיזם על מנת לשנות את ההתנהגות לפי המצב המתאים. נתבונן בתרשים של תבנית זו -



המחלקה context רוצה להפעיל פעולה (request). כעת נקרא למחלקה state שיש לה את המתודה handle. למחלקה זו מספר יורשים, Concrete State A, B כאשר כל אחד ממש את handle בהתאם למצב שלו. כעת כאשר נקרא ל-state.handle מתוך context המתודה המתאימה תקרא בהתאם למצב הספציפי שאנו נמצאים בו.
נמחיש בעזרת דוגמא. נניח שיש לנו אובייקט מסוג רובוט, שנע בהתאם למצבו הגופני - בריא או חולה. נרצה שתהיה לנו מחלקה של Health (זה מחלקת ה-state שלנו) עם המתודה move. בנוסף יש לה שתי מחלקות יורשות - Healthy ו-Faulty כל אחת ממשת את move בצורה שונה. כאשר נקרא ל- healthy.move הרובוט יזוז בהתאם למצב הבריאותי שלו.

14.7 Behavioral Patterns: Template vs Strategy

לסיום החלק של תבניות העיצוב, נשווה בין שתי תבניות עיצוב שראינו דוגמאות רבות לשימוש בהן - Template ו-Strategy.
נתבונן בדוגמא לתבנית העיצוב Template - נניח שיש לנו מחלקה ממיינת (sorter) בעל מתודת sort ומתודת compare.
בעזרת תבנית עיצוב, נוכל שמחלקות שונות יממשו את מתודת ה-compare ובכך נממש דרכים שונות למיון. החיסרון של שיטה זו היא שבכל פעם שנרצה לממש מיון שונה נצטרך לממש שוב ושוב את מתודת ה-compare במחלקות מתאימות. נקבל כפילות קוד רבה שאותה ראינו בדוגמאות רבות לאורך ההרצאות.
נמשיך עם הדוגמא של תבנית העיצוב Strategy. בעזרת תבנית עיצוב זו נוכל לממש מחלקה sorter עם מתודה sort שמכילה אובייקט מסוג comparator. את המחלקה comparator נוכל לממש בדרכים שונות, כאשר כל מחלקה יורשת צריכה לממש את מתודה ה-compare. נשים לב שאם נרצה לממש יותר מאלג' מימוש אחד, נוכל לעשות זאת מבלי להוסיף עוד קוד כפול.
ע"י הוספת אלג' מיון נעבור בעצם אל תבנית העיצוב Bridge.