

יסודות הנדסת תוכנה

הרצאה 9

Composite

הסיפור מוטיבציה <= אותו כלי גרפי שיאפשר לנו להרכיב משהו כללי מחלקים נפרדים

ואז הפעולות יעבדו על הדבר הכללי אותו דבר

(כמו שהראו בכיתה בפאור פוינט בנפרד הריבוע והמשולש זזים אבל אם נחבר

אותם ונזיז אז יזוזו יחד כאובייקט אחד = ההתייחסות זהה)

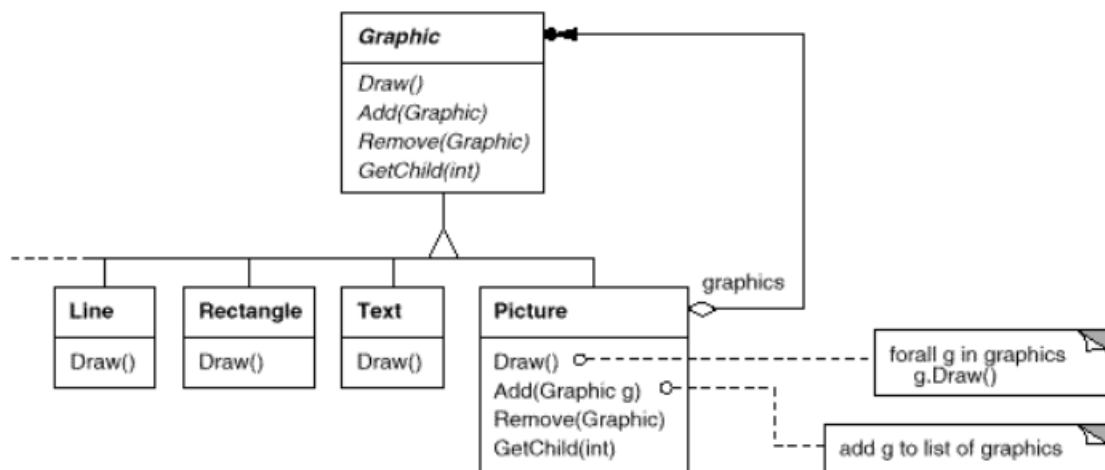
נרצה להיות מסוגלים לייצג אובייקט מורכב

והממשק שהלקוח מקבל הוא אותו ממשק

הפתרון -- < מחלקה שמייצגת אובייקטים מורכבים ופשוטים ואז יש 2 מחלקות יורשות

שאחת מהן יודעת לייצר הרכבה של אובייקטים פשוטים למשהו יותר מורכב

המבנה של composite יכול להיות עץ או גרף



המשתתפים

Component <= אחראית על להגדיר את אותו ממשק (שאמור להיות משותף)

יכול להגדיר את התנהגות ברירת המחדל (אם קיימת כזאת)

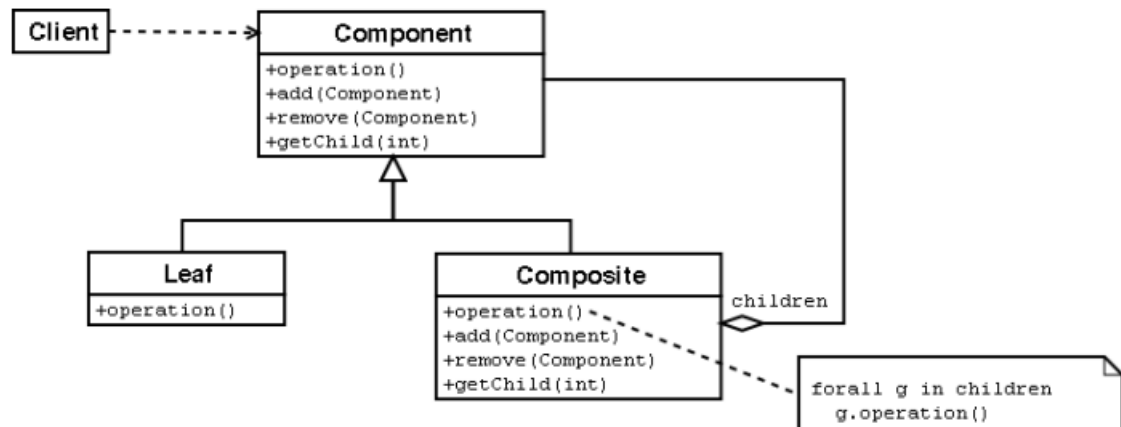
והיא יכולה להגדיר את פעולות הטיפול בילדים

בדוגמא שלנו זה `graphic`

Leaf <= מייצג את האובייקט קצה/הפרימיטיבי/הפשוט, אין לו ילדים והוא יודע לבצע את הפעולות שלו

Composite <= איך אני מטפלת בילדים, מחזיקים רפרנס לילדים ויממש את כל הפעולות שאני צריכה להורה ולילד

Client <= זה שעושה מניפולציה על האובייקט ולא יודע כלום, מבחינתו יש ממשק אחד



מבנה שמאפשר לי להגדיר יחס של שלם וחלקים, עץ עם אבות ובנים
אותו לקוח יכול להתייחס לעלה ולcomponent באותה צורה

Intent and context

לייצר אובייקטים מורכבים

והיתרון פה הוא שאנחנו מתייחסים לאובייקטים בצורה אחידה

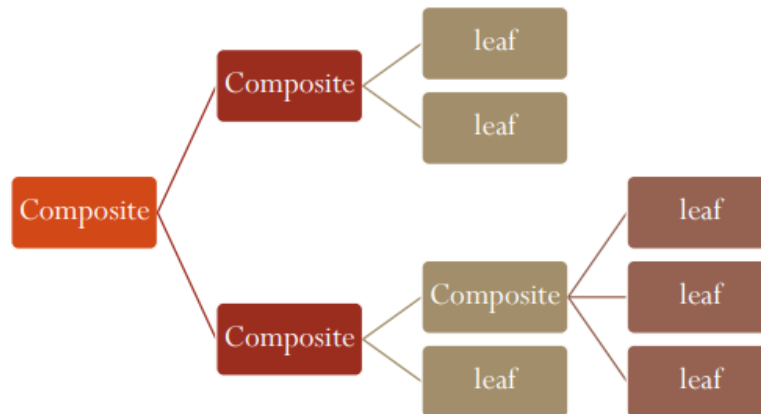
ונרצה להשתמש בזה כשאני רוצה להראות יחסים של שלם וחלקים, היררכיה וכשאני רוצה שהלקוח

יתנהג לכל האובייקטים באותה צורה

Component = node

Leaf = file

Composite = directory



נרצה למצוא באמצעות פעולת find את כל הקבצים שמכילים תת מחרוזת מסוימת

class Node

```

abstract class Node {
    Node(String n, Directory p){
        _name = n; _parent = p;
        if (_parent != null){ p.add(this); }
    }
    public String getName(){ return _name; }
    public String getAbsolutePath(){
        if (_parent != null){
            return _parent.getAbsolutePath() + getName();
        }
        return getName();
    }
    public abstract Vector find(String s);
    protected String _name;
    protected Directory _parent;
}
  
```

class File

```
class File extends Node {
    private String _contents;

    File(String n, Directory p, String c){
        super(n,p); _contents = c;
    }
    public Vector find(String s){
        Vector result = new Vector();
        if (getName().indexOf(s) != -1){
            // s is found
            result.add(getAbsolutePath());
        }
        return result;
    }
}
```

class Directory (1)

```
class Directory extends Node {
    private Vector _children;
    Directory(String n){ this(n, null); }
    Directory(String n, Directory p){
        super(n,p);
        _children = new Vector();
    }
    public String getAbsolutePath(){
        return super.getAbsolutePath() + "/";
    }
    public void add(Node n){
        _children.addElement(n);
    }
    ...
}
```

class Directory (2)

```
...
public Vector find(String s){
    Vector result = new Vector();
    if (getName().indexOf(s) != -1){
        result.add(getAbsolutePath());
    }
    for (int t=0; t < _children.size(); t++){
        Node child = (Node)_children.elementAt(t);
        result.addAll(child.find(s));
    }
    return result;
}
}
```

class Main

```
public class Main {
    public static void main(String[] args){
        Directory root = new Directory("");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("foo", usr);
        File bar1 = new File("bar1", usr, "abcdef");
        File bar2 = new File("xbar2", usr, "abcdef");
        File bar3 = new File("yybarzz3", usr,
"abcdef");
        System.out.println(root.find("bar"));
    }
}
```

output

```
[/usr/bar1,/usr/xbar2,/usr/yybarzz3]
```

סוגיות שעולות

יש פעולות שנרצה להוריש לעלים והן לא רלוונטיות לאובייקט המורכב
כל פעולות ההוספה ומחיקת של ילד רלוונטיות רק לאובייקט המורכב ואנחנו מורשים את זה גם לעלים
אז השאלה עד כמה זה נכון למקם את הדברים אצל האב, כי ככה מגיעים למצב שמורשים דברים
שהם לא רלוונטיים לכל היורשים
מה שכן זה מקל על הוספת סוגים חדשים של רכיבים(לא משנים כלום אצל הלקוח)

והלוקח משתמש בהכל בצורה אחידה

מה לגבי יישום?

נרצה להחזיק רפרנס לאבא

יש פה שיקול מצד אחד יש אינטרפייס שלא מתאים לכולם ומצד שני רוצים לתת את המקסימום

כי זה גם לא יפה לרשת פעולות ולנוון אותן

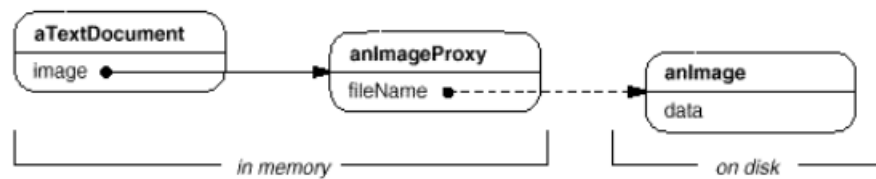
Proxy

סיפור מוטיבציה => עורך מסמכים ונרצה לאפשר לו להכניס גם תמונות

לעיתים לא צריך לעלות ישר כי אם נכנסתי לקובץ ודפדפתי רק במבוא אז אין צורך להעלאות את

התמונות שנמצאות בדף 300

או אם אני רק בתחילת המסמך גם אין צורך, ואפשר לדחות לרגע שבו נצטרך



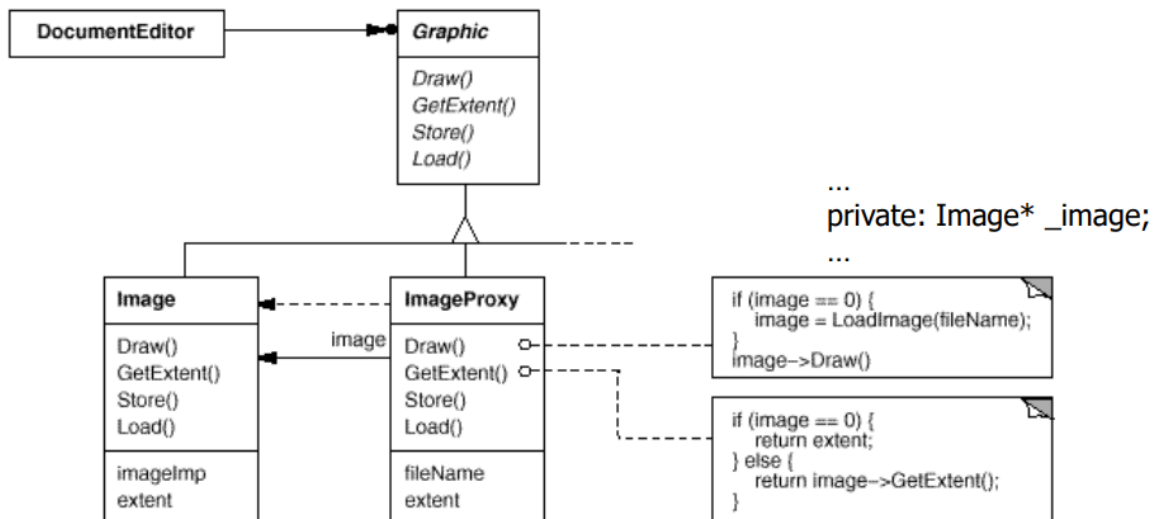
הפתרון הוא להשתמש באובייקט אחר

בדוגמא שלנו הוא יתפוס מקום של הקובץ הזה, של התמונה והוא רק מחזיק את הגודל של התמונה

ורק כשאגיע למצב שאצטרך את התמונה אז יהיה לי את התמונה

התמונה מחזיקה את הפרוקסי שמחזיק רפרנס לתמונה האמיתית

וברגע שצריך הוא מביא את התמונה האמיתית



המשתתפים

Proxy \leq אחראי על להחזיק רפרנס לreal subject

לספק ממשק הזה לשל של subect

הלקוח יחזיק את הפרוקסי והפרוקסי הוא זה שיחליט על הגישה למוצר האמיתי

יש כמה סוגי פרוקסי

פרוקסי שיוודע להיות נציג למשהו שנמצא במרחב כתובות אחר (מגדיר מתי אפשר לגשת)

וירטואל פרוקסי והוא מחזיק מידע יקר, ממלא מקום בינתיים

protection proxy והוא שולט על הגישה, למי מותר ולמי אסור

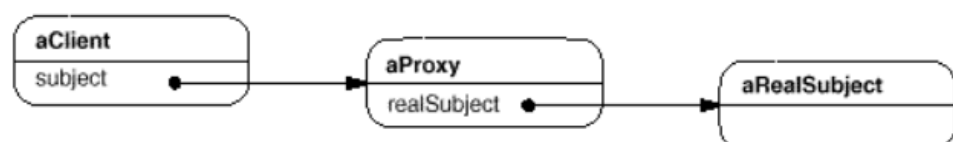
Smart counter – יכול לעשות ספירות כמו כמה פעמים קראו לsubject או איזה פעולות

קראו לsubject

Subject \leq מגדיר את הממשק המשותף של הreal subject ושל הפרוקסי

Real subject \leq מגדיר את האובייקט האמיתי שבשבילו יצרנו את הפרוקסי

באופן כללי:



מבחינת לקוח הוא לא יודע מי נותן לו תשובה הפרוקסי או הreal subject

Intent and context

פרוקסי משמש כממלא מקום לאובייקט הרצוי

נשתמש בו כשרוצים להימנע מייצור אובייקטים יקרים כאשר אפשר לדחות את יצירתם

או כאשר רוצים לשלוט בגישה לאובייקט מסוים

לדוגמא:

נמשיך עם UNIX

ישמש לנו כפרוקסי

```
class Link extends Node {  
  
    private Node _realNode;  
                                //either file or directory  
  
    Link(String n, Node w, Directory p){  
        super(n,p); _realNode = w;  
    }  
    public String getAbsolutePath(){  
        return super.getAbsolutePath() + "@";  
    }  
}  
...
```

```
abstract class Node {  
    Node(String n, Directory p){  
        _name = n; _parent = p;  
        if (_parent != null){ p.add(this); }  
    }  
    public String getName(){ return _name; }  
    public String getAbsolutePath(){  
        if (_parent != null){  
            return _parent.getAbsolutePath() + getName();  
        }  
        return getName();  
    }  
}
```


פעולת find מחזירה את
כל הקבצים שיש להם תת
מחרוזת מסוימת

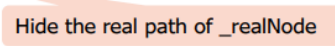
והרי לא נרצה לחשוף את
כל הpath שלנו

אז מסתירים את המסלול
האמיתי על ידי הlink

class Link (2)

```
public Vector find(String s){
    Vector result = new Vector();
    if (getName().indexOf(s) != -1){
        result.add(getAbsolutePath()); // added : /.../.../linkName@
    }

    Vector resultsViaLink = _realNode.find(s);
    String realNodePath = _realNode.getAbsolutePath();
    int n = realNodePath.length();
    for (int t=0; t < resultsViaLink.size(); t++){
        String r = (String)resultsViaLink.elementAt(t);
        String rr = super.getAbsolutePath() + "/" +
            r.substring(n);
        result.add(rr);
    }
    return result;
}
```



Link-to-link הוא קישור
לusr והוא מסתיר זאת

class Main

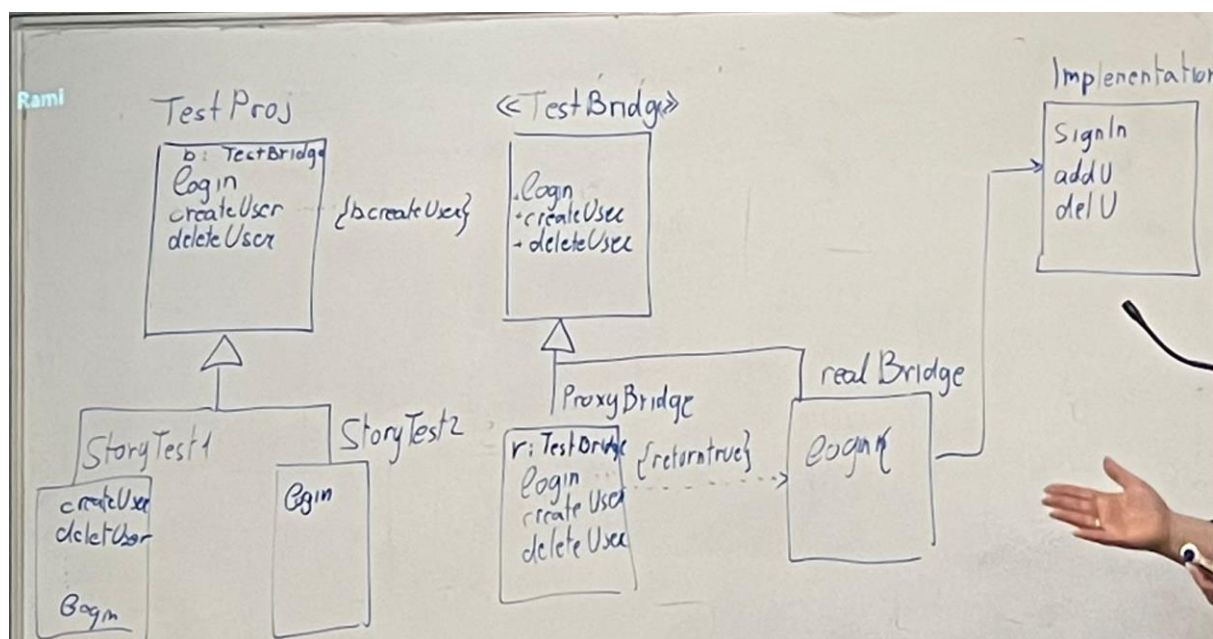
```
public class Main {
    public static void main(String[] args){
        Directory root = new Directory("");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("foo", usr);
        File bar1 = new File("bar1", foo, "abcdef");
        File bar2 = new File("xbar2", foo, "abcdef");
        File bar3 = new File("yybarzz3", foo, "abcdef");
        Link link = new Link("link-to-usr", usr, root);
        Link linkToLink =
            new Link("link-to-link", link, root);
        System.out.println(root.find("bar"));
    }
}
```

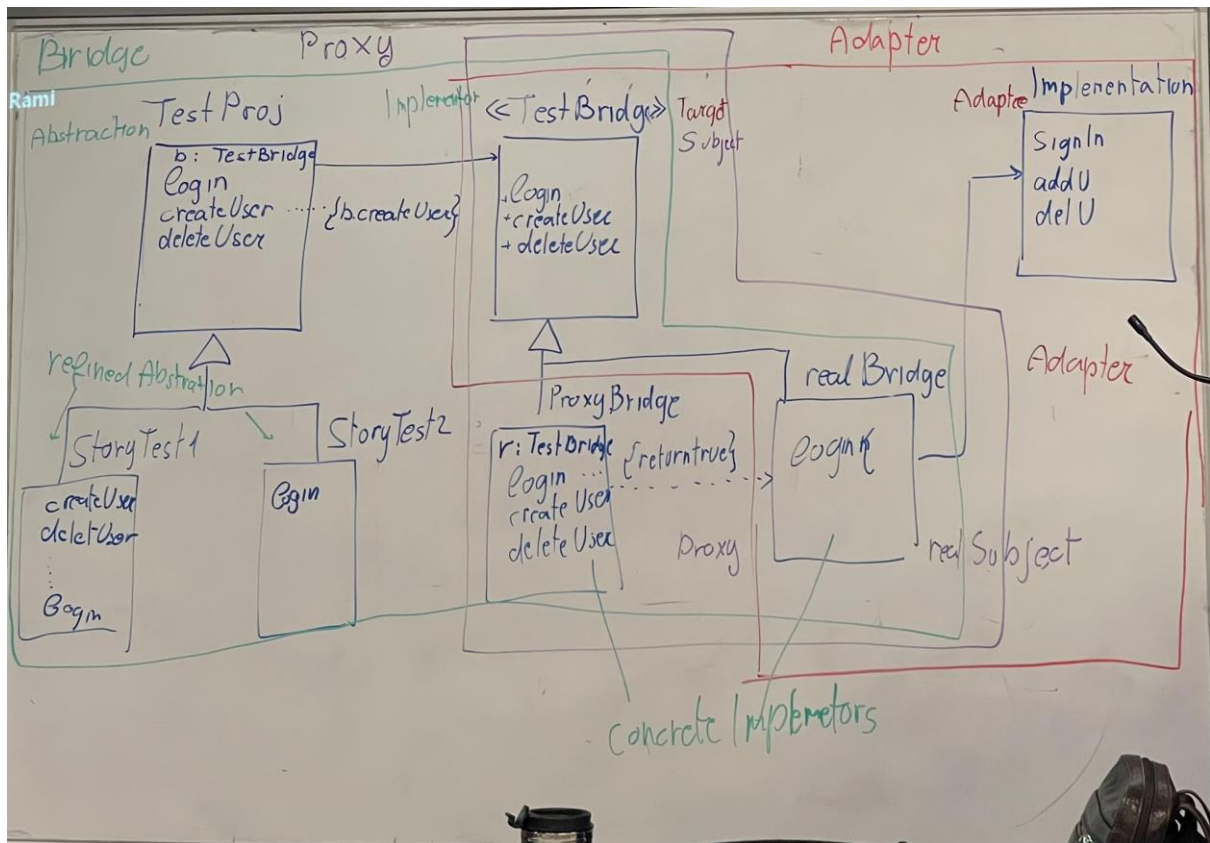
output

```
[/usr/foo/bar1,
 /usr/foo/xbar2,
 /usr/foo/yybarzz3,
 /link-to-usr/foo/bar1,
 /link-to-usr/foo/xbar2,
 /link-to-usr/foo/yybarzz3,
 /link-to-link/foo/bar1,
 /link-to-link/foo/xbar2,
 /link-to-link/foo/yybarzz3]
```

Proxy vs Adapter

באדפטר יש אינטרפייס שונה לעומת זאת בפרוקסי מדובר על אותו אינטרפייס
באדפטר המטרה להשתמש באותו אובייקט כדי לבצע את ההתאמה





Behavioral patterns

אופן מימוש התנהגות בעזרת אינטראקציה בין כמה מחלקות

Observer

סיפור המוטיבציה <= גיליון נתונים עם טבלה שמכילה את הדאטה שלי

רוצים לעשות תצוגות שונות על אותו דבר

כל פעם שהנתונים ישתנו גם הטבלה תשתנה

אין הגבלה על כמה שמסתכלים על הדאטה

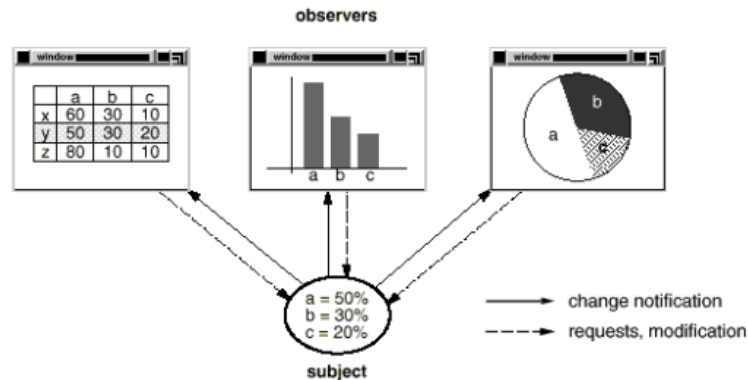
זוהי הרעיון של observer

Subject – זה שמעניין ועליו רוצים לדעת

Observer – אלה שרוצים לדעת מה קורה

הנתון מודיע שיש שינוי, הobserver הולך לבדוק מה השינוי, ואם השינוי נוגע לי אז אשנה

ואם לא אז לא



משתתפים

Subject <= מחזיק רשימה של observers שלו ומממש שיטות שמעניינות אותו

attach לחבר משהו חדש לרשימה, detach להסיר משהו מהרשימה

Notify תעבור על כל observers ברשימה ותעשה עליהם update

Observer <= כל מי שרוצה להיות "מנוי" צריך לממש את הממשק, את שיטת update על מנת

שהיה אפשר להודיע על שינויים

concreteSubject <= יורש מתוך subject שולח הודעה לobservers כאשר המצב משתנה

יכולות להיות לו פעולות כמו

setState משהו יכול לשנות את מצבו getState אני יכולה לספק למי

ששואל אותי מה המצב על subject

concreteObserver <= מחזיק רפרנס לconcreteSubject

intent and context

לייצר קשרים של 1 לרבים, כאשר לרבים יש תלות ב1

הרבה תלויים ב1 והם מעוניינים לדעת בכל פעם שיש שינוי במצב של ה1 הזה

נרצה להשתמש בזה כאשר יש 2 אבסטרקציות שאחת מתעניינת בשנייה

(אין ביניהן איזה קשר או היררכיה)

ונרצה להיות מסוגלים להודיע על שינוי באובייקט לאלה ש"רשומים אליו" ולא באמת אכפת לי מי הם

(אוספים אותם לרשימה וזהו)

תהיות וכל מיני

Observer יכול להסתכל על יותר מsubject אחד

מי עושה טריגר לupdate? הsubject קורא לnotify

ומי שעושה טריגר לnotify הוא העדכון בstates

נוודא שהstate מוכן ואז נקרא לnotify

רוצים להיות במצב שבו אני לא אסתכל על subject שהוא נמחק
הוא צריך לדאוג לפני שהוא משמיד את עצמו להודיע ואז להשמיד את עצמו

יש 2 שיטות לעדכון observern

Push <= Sn שולח הודעה ומודיע מה השתנה, נותן מידע מפורט לגבי השינוי
Pull <= S שולח הודעה שהיה עדכון והאובייקט הולך ומושך כדי לגלות איזה שינוי קרה

Observer יכול גם להירשם לשירות מסוים

ואז אם קרה שינוי מסוים שתקף רק למי שנרשם אז רק הם אלה שיקבלו את העדכון על כך

לדוגמא:

נוסיף למערכת הקבצים observer עבור כל שינוי שקורה בקבצים ומדפיס הודעה על כך שקרה שינוי

Interface Observer & Class FileObserver

```
interface Observer {
    public void update();
}

class FileObserver implements Observer {
    FileObserver(File f) {
        f.attach(this);
        _subject = f;
    }
    public void update() {
        System.out.println("file " +
            _subject.getAbsolutePath() +
            " has changed.");
    }
    private File _subject;
}
```

Updated Class File (1)

```
class File extends Node {
    File(String n, Directory p, String c){
        super(n,p);
        _contents = c;
    }
    public void attach(Observer o){
        if (!_observers.contains(o)){
            _observers.add(o);
        }
    }
    public void detach(Observer o){
        _observers.remove(o);
    }
    ...
}
```

Updated Class File (2)

```
...
public void notifyObservers(){
    for (int t=0; t < _observers.size(); t++){
        ((Observer)_observers.elementAt(t)).update();
    }
}
public void write(String s){
    _contents = s;
    notifyObservers();
}

private String _contents;
private Vector _observers = new Vector();
}
```

Updated Client

```
public class Main {  
    public static void main(String[] args){  
        Directory root = new Directory("");  
        File core = new File("core", root, "hello");  
        Directory usr = new Directory("usr", root);  
        File bar1 = new File("bar1", usr, "abcdef");  
  
        // create observer for file bar1  
        FileObserver obs = new FileObserver(bar1);  
        bar1.write("abracadabra");  
        bar1.write("ffffff");  
        bar1.write("gggggg");  
    }  
}
```

Output

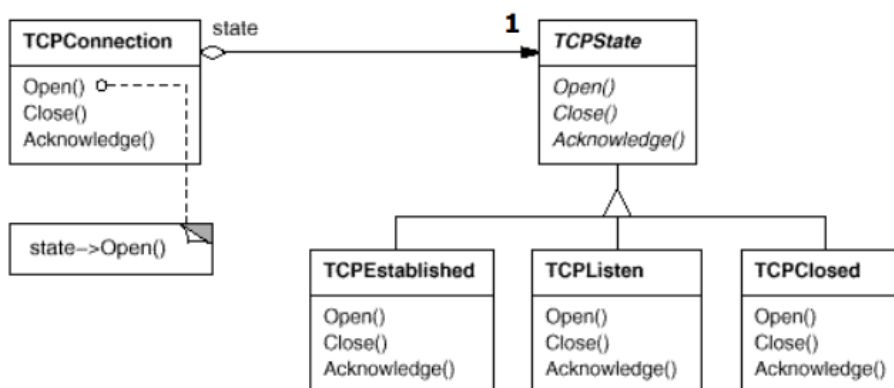
- file /usr/bar1 has changed.
- file /usr/bar1 has changed.
- file /usr/bar1 has changed.

State

לדבר על התנהגות שונה בהתאם לstate

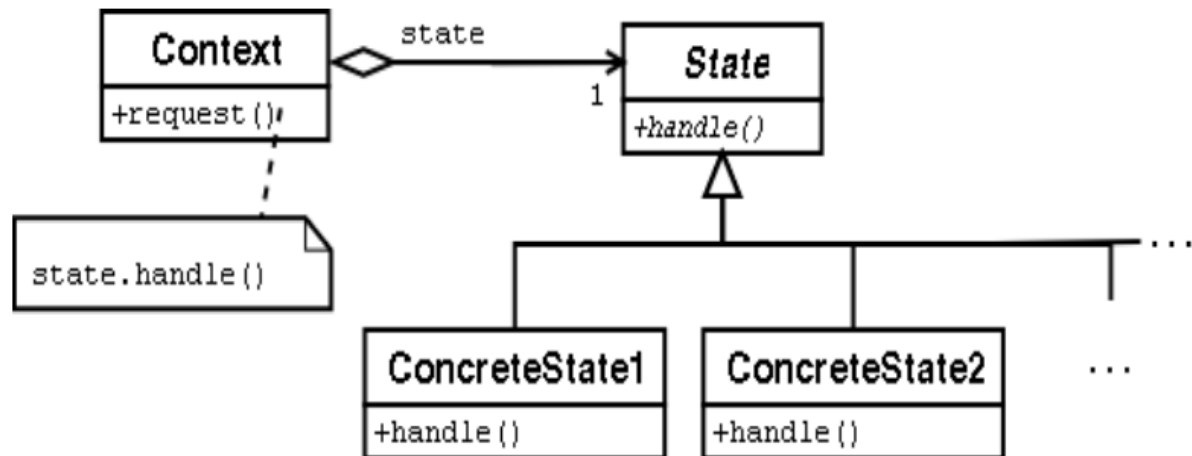
התנהגות האובייקט משתנה בהתאם לstate שבו הוא נמצא

- Key idea: Introduce an abstract class *TCPState* to represent the states of the network connection.



אני לא רוצה לכל פעולה לשאול באיזה מצב אני נמצאת, אלא, כל ההתנהגות התלויה במצב להעביר למקום אחר.

מה שלא קשור למצב - נשאר באובייקט, אבל מה שתלוי במצב - נוציא החוצה



משתתפים:

Context <= האובייקט שמדברים עליו, שמשנה את ההתנהגות בהתאם לstate

מגדיר אינטרפייס ללקוח

ומחזיק רפרנס לconcreteState

State <= מגדיר אינטרפייס שמגדירה את הפעולות שתלויות במצב

concreteState <= לכל state הוא ממש את הפעולות שהוגדרו, כמו הנציג של context

intent and context

המטרה המרכזית להפריד התנהגות שהיא תלויה במצב ולאפשר לאובייקט לשנות התנהגות

נשתמש כאשר התנהגות האובייקט תלויה במצב האובייקט

כשיש הרבה תנאים