פרופ' עמיר תומר
amir@amirtomer.com

פרק 9

עקרונות למיטוב התכן
Software Design Principles

הנושאים בפרק זה

- דפוסי תכן
- עקרונות לתכן יציב (SOLID)
- חלוקה למארזים וארגונם הנכון

דפוסי תכן (design patterns)

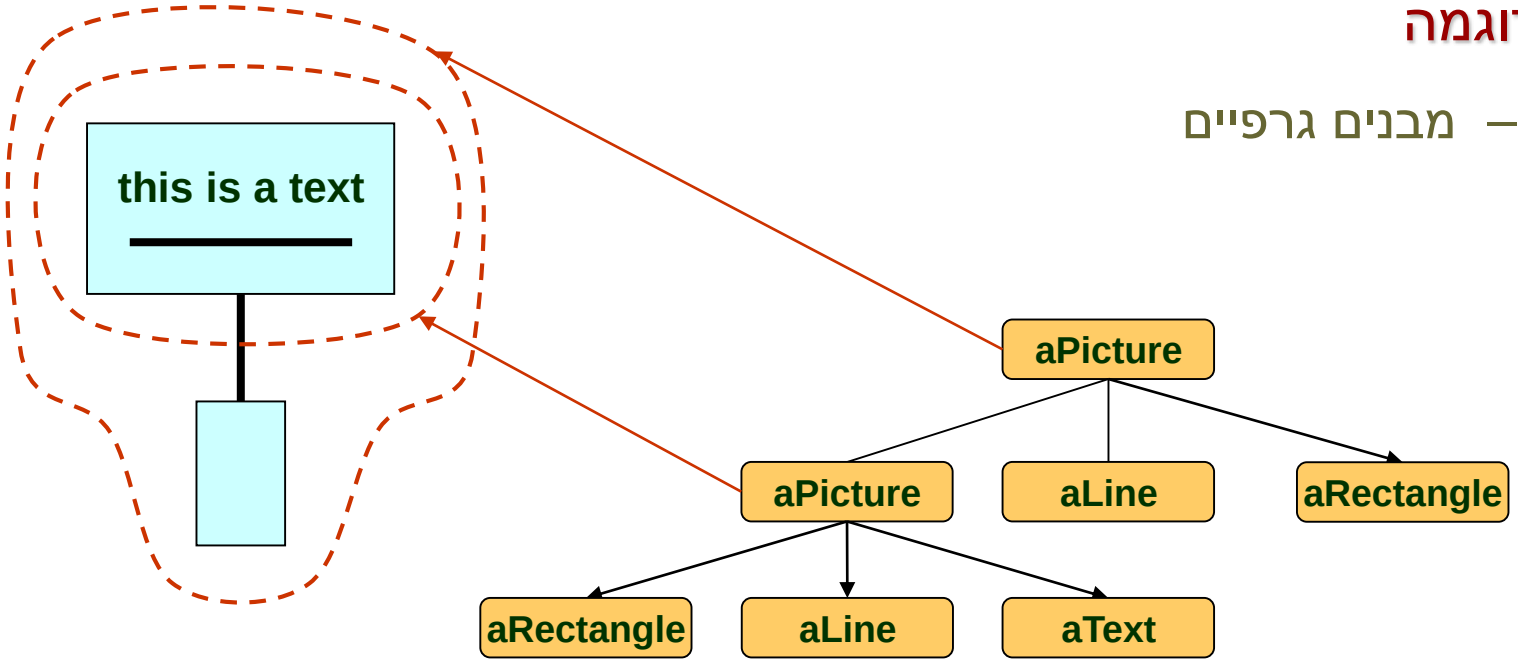
- פתרון עקרוני לבעיית-תכן (מונחה עצמים) החוזרת על עצמה
  - תבנית שיטתית של אופן הצגת הפתרון
  - ארגון כללי של עצמים ומחלקות לפתרון הבעיה
- נדרשת "תפירה" של הדפוס עבור פתרון ספציפי בהקשר נתון
- תכולה של דפוס תכן
  - תאור הבעיה
  - תאור הפתרון
  - תנאי השימוש בפתרון
  - השלכות הפתרון
  - הנחיות מימוש
  - דוגמאות מימוש

סוגי דפוסי תכן

- ע"פ "כנופיית הארבעה" (The GOF = Gang of Four)
  - E. Gamma, R. Helm, R. Johnson & J. Vlissides, Design Patterns, Addison-Wesley, 1995.
- דפוסי מבנה (structural patterns)
  - הגדרת מבנים מורכבים
- דפוסי יצירה (creational patterns)
  - יצירת אובייקטים בזמן ריצה
  - הקטנת התלות בין הגדרת המחלקות (במפרט) ובין האובייקטים בפועל (בקוד)
- דפוסי התנהגות (behavioral patterns)
  - הקצאת התנהגות לאובייקטים בזמן ריצה
  - העברת המיקוד מזרימת הבקרה אל אופן התקשורת בין האובייקטים

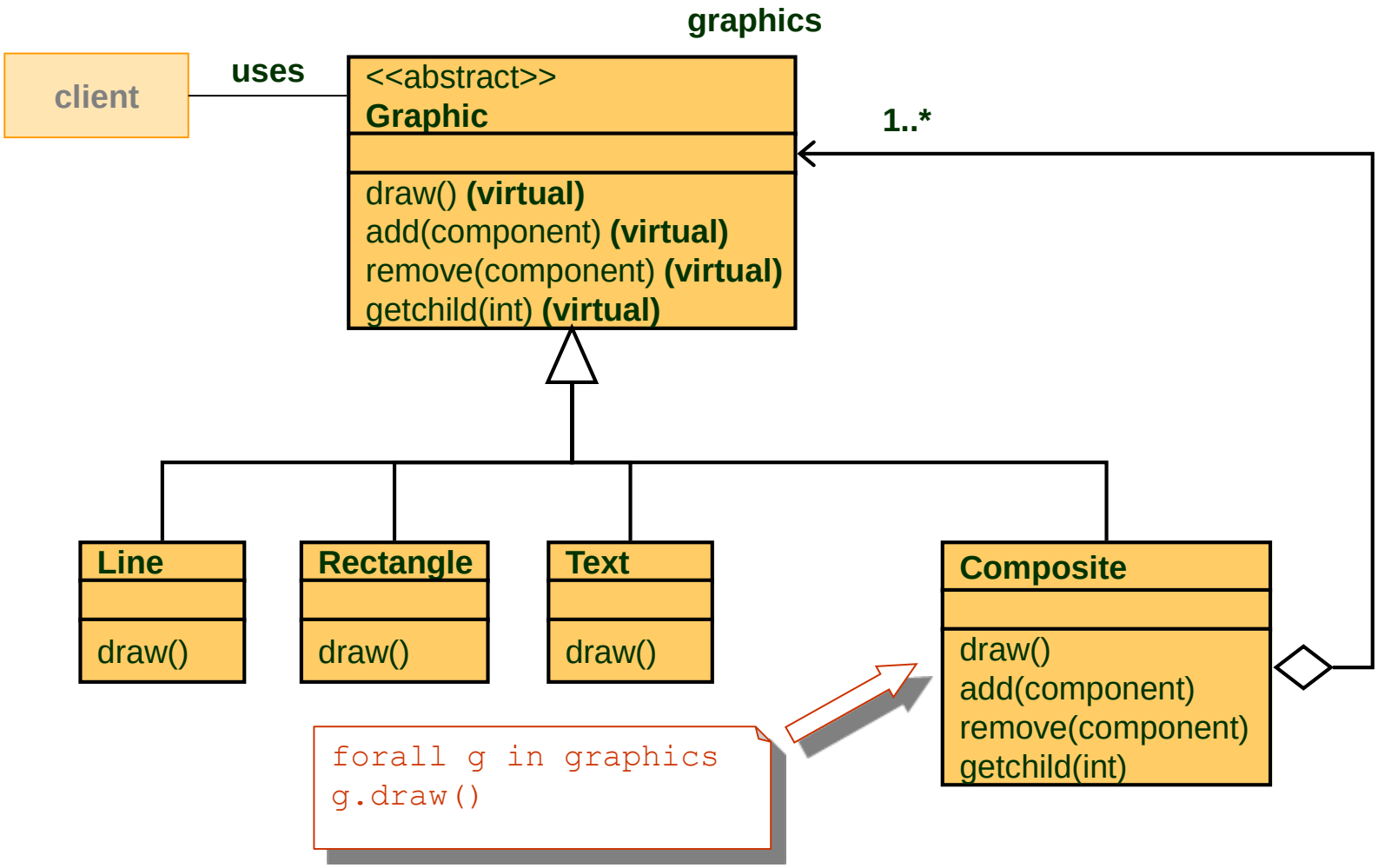
Composite Pattern (1)

- דפוס מבנה
- הבעיה
  - יצירת מבנה היררכי רקורסיבי
- דוגמה
  - מבנים גרפיים



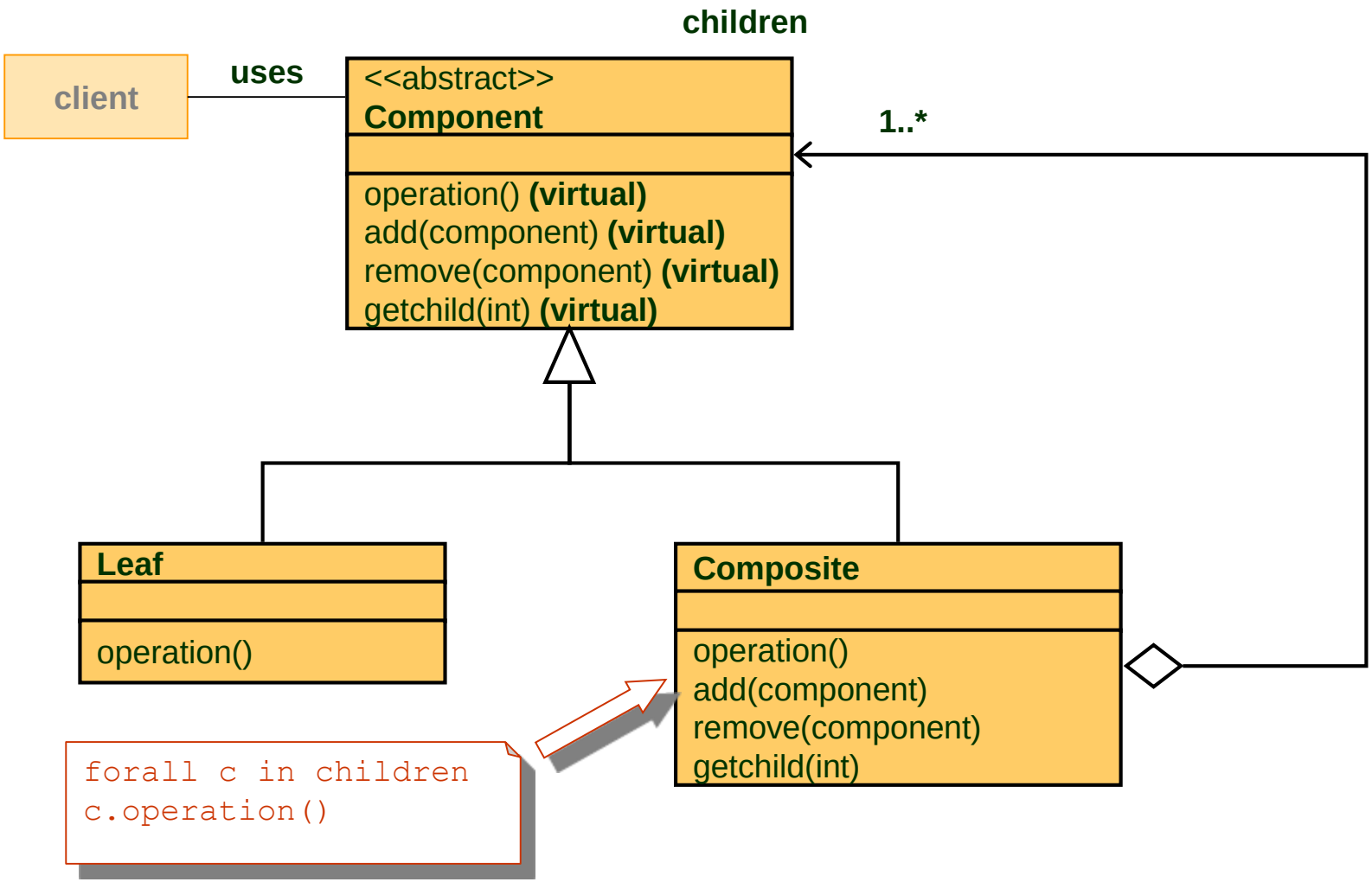
Composite Pattern (2)

- מימוש הדוגמה



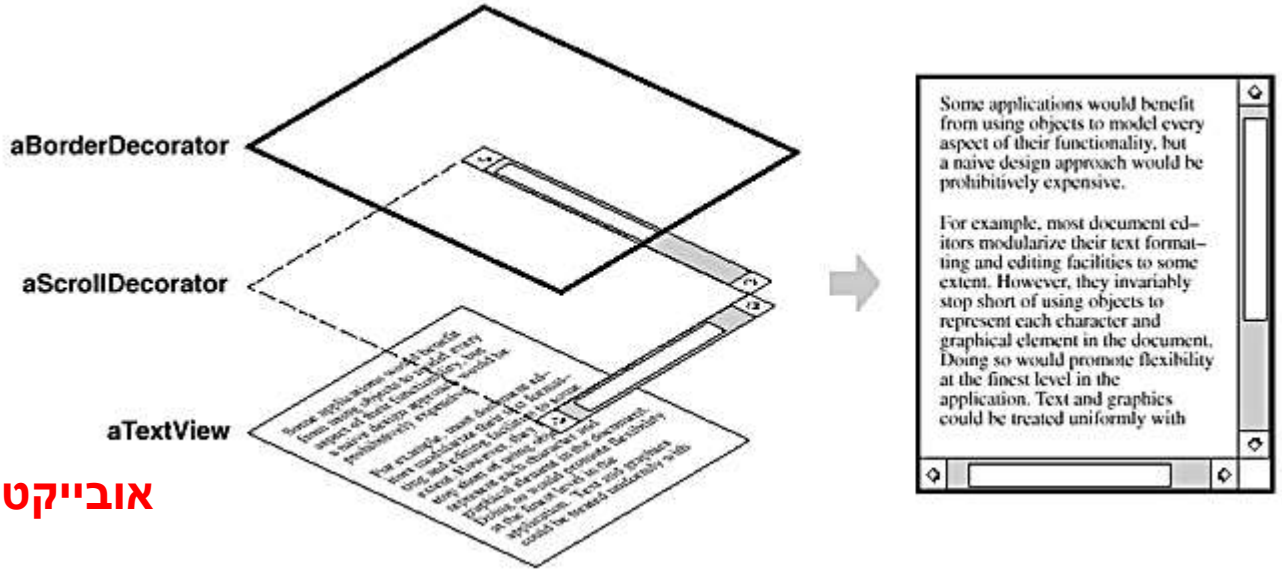
Composite Pattern (3)

- התבנית הכללית



Decorator Pattern (1)

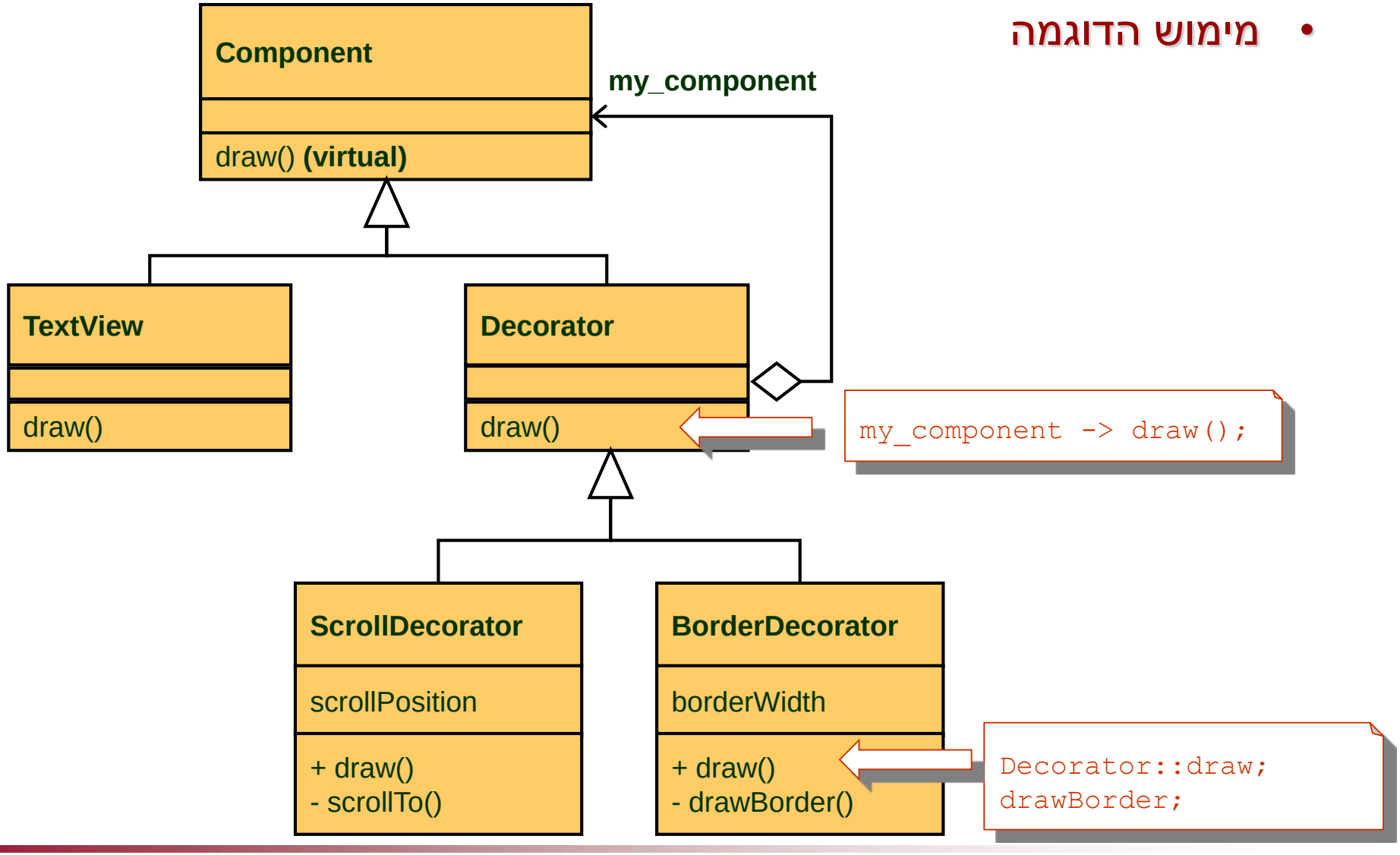
- דפוס מבנה
- הבעיה
  - הקניית יכולות משלימות למופעים ספציפיים של מחלקה
- דוגמה
  - חלון להצגת טקסט



אובייקט בסיסי

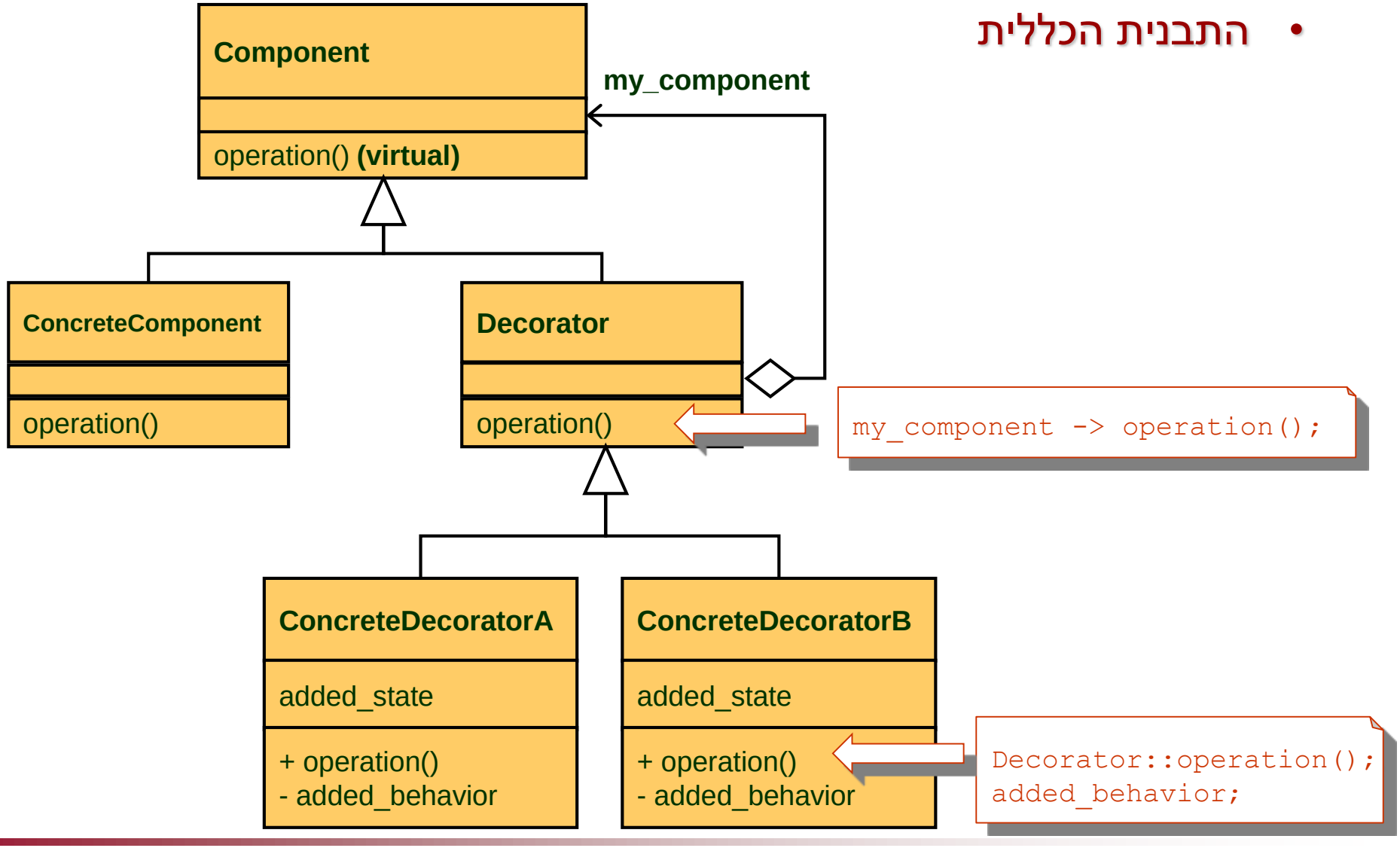
Decorator Pattern (2)

- מימוש הדוגמה



Decorator Pattern (3)

- התבנית הכללית

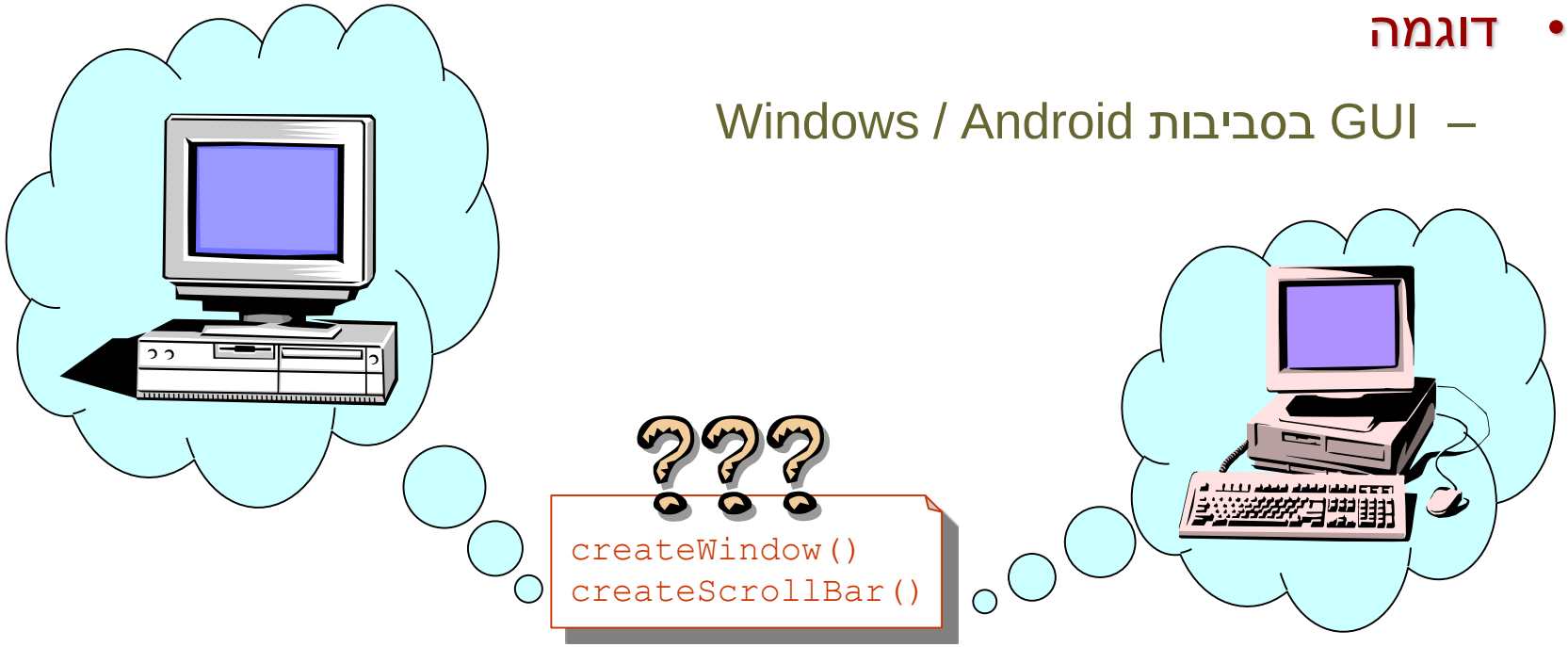


דוגמה בג'אווה – קפה+תוספות

http://en.wikipedia.org/wiki/Decorator_pattern

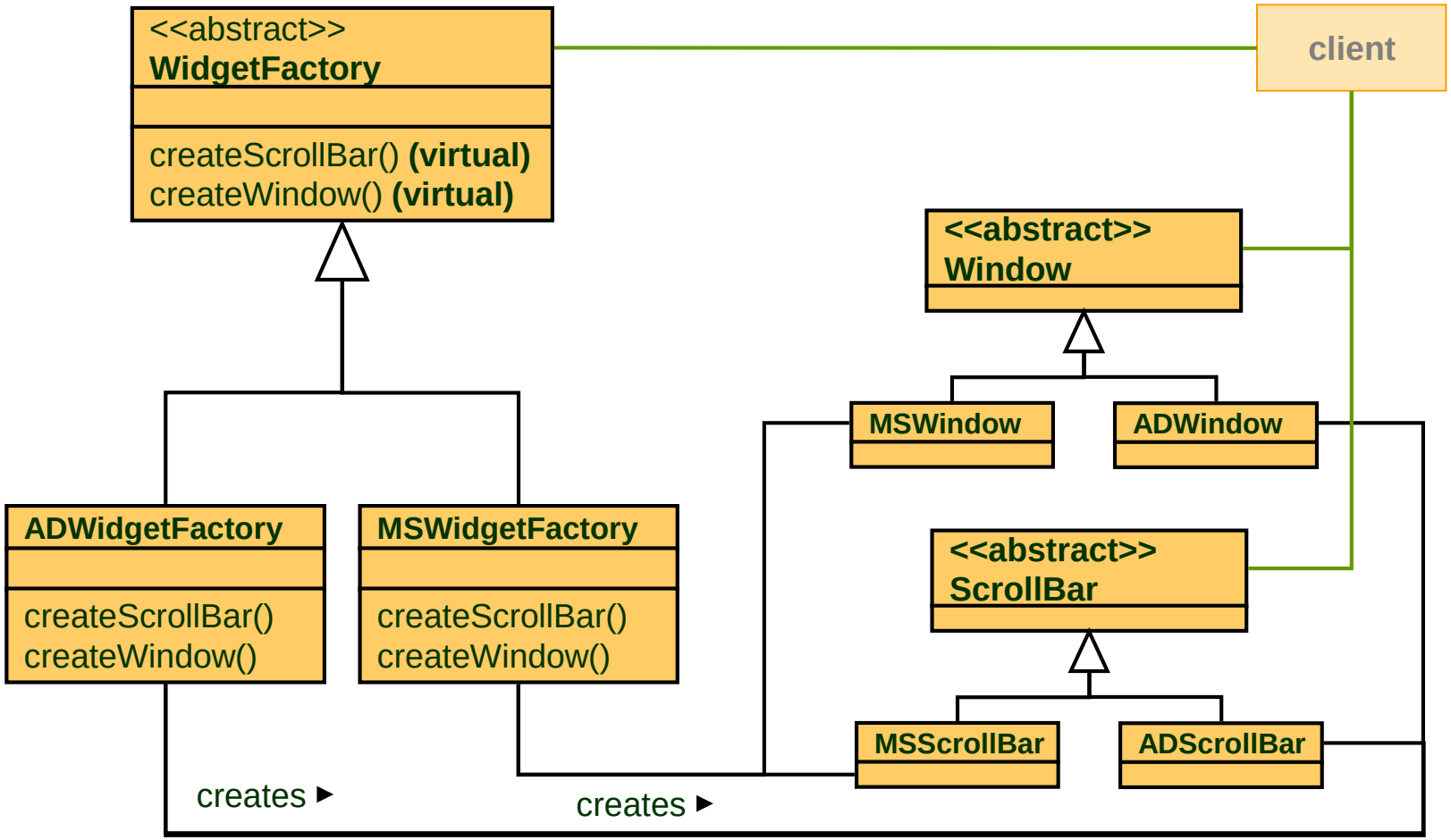
Abstract Factory Pattern (1)

- דפוס יצירה
- הבעיה
  - יצירה ושימוש באובייקטים בסביבה משתנה, ללא תלות במימוש הספציפי שלהם (כלומר, באופן שקוף ליישום)
- דוגמה
  - GUI בסביבות Windows / Android



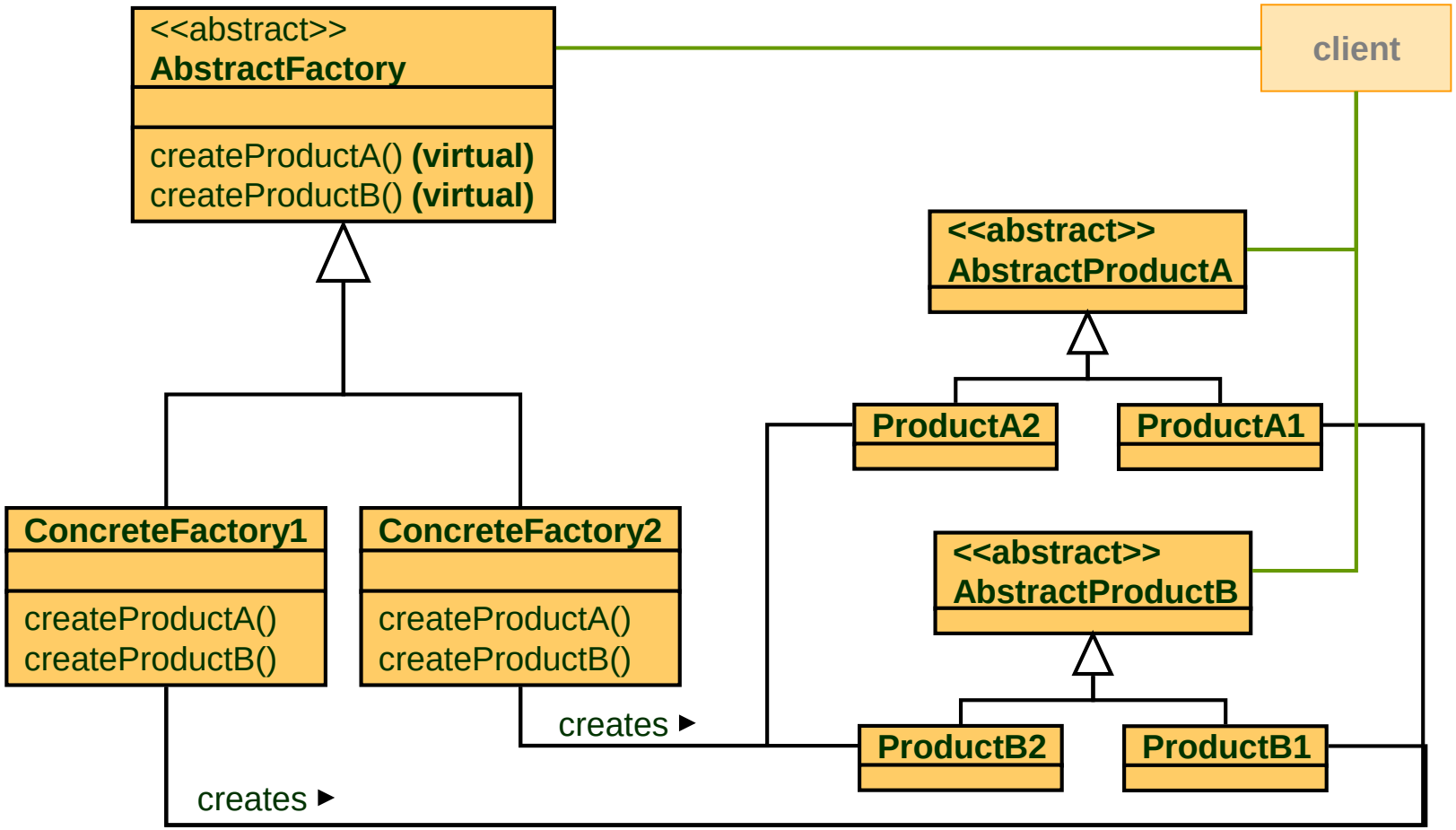
Abstract Factory Pattern (2)

- מימוש הדוגמה



Abstract Factory Pattern (3)

- התבנית הכללית

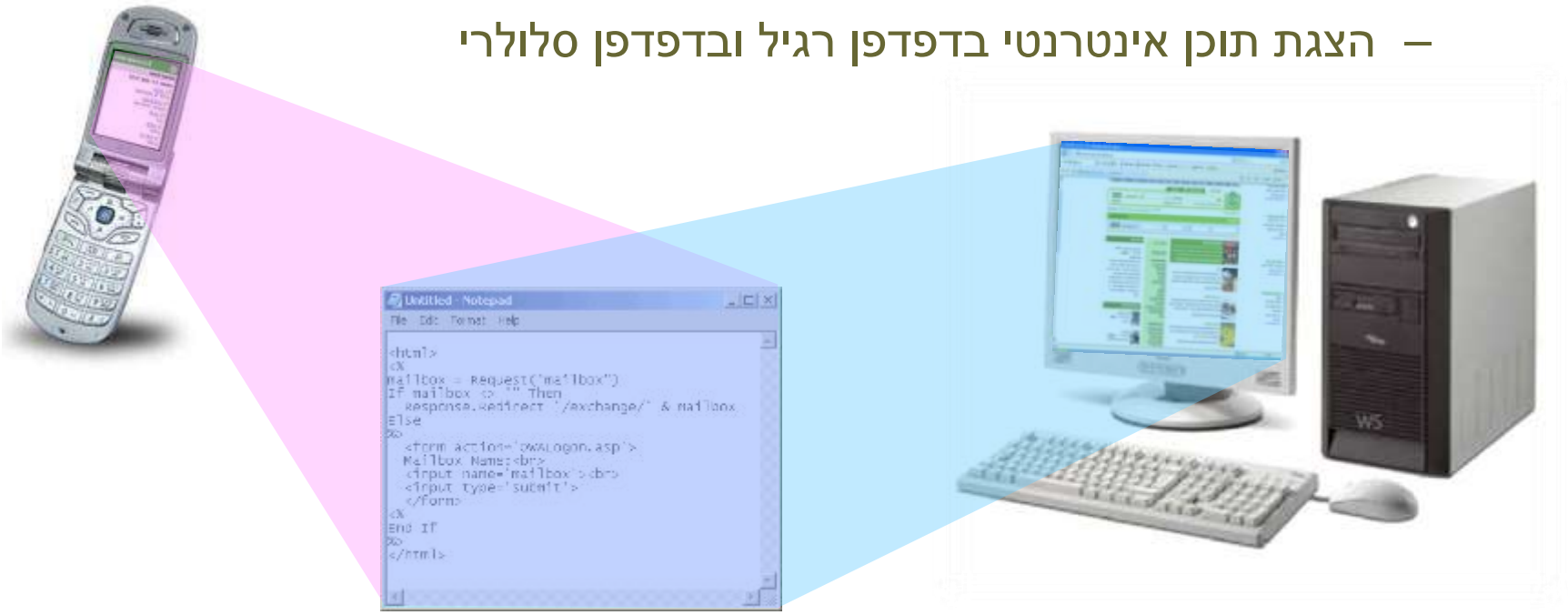


דוגמה בפסאודו-קוד

http://en.wikipedia.org/wiki/Abstract_factory_pattern

Observer Pattern (1)

- דפוס התנהגות
- הבעיה
  - הצגה של אותו מידע באופנים שונים
- דוגמה
  - הצגת תוכן אינטרנטי בדפדפן רגיל ובדפדפן סלולרי



Observer Pattern (2)

- Subject
  - מחזיק את הנתונים
  - מתחזק רשימה של משקיפים
  - מודיע לכל משקיף ברשימה על שינוי בנתונים
- Observer
  - משקיף אבסטרקטי
- Concrete Observer
  - משקיף עבור תצוגה ספציפית
  - מגיב להודעות לגבי שינוי בנתונים ע"י שינוי התצוגה בהתאם

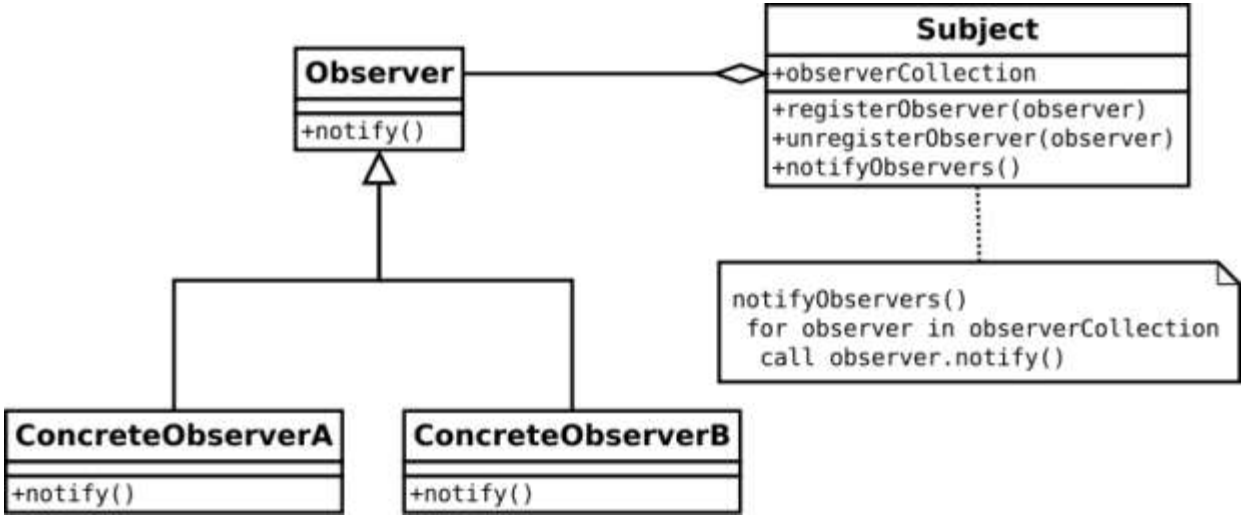


Diagram source: Wikipedia

Strategy Pattern (1)

- דפוס התנהגות
- הבעיה
  - ביצוע של מתודה באופנים שונים, תוך שקיפות למשתמש
- דוגמה
  - מתן הנחיית ניווט בצורה גראפית או בצורה קולית

Strategy Pattern (2)

- Context
  - שירות כלשהו שבאמצעותו ניתן לבצע משימה / חישוב באופנים שונים, מבלי לדעת את פרטי היישום
- Interface
  - הגדרת החתימה של האלגוריתם
- Implementation One/Two/…
  - יישומים שונים של האלגוריתם

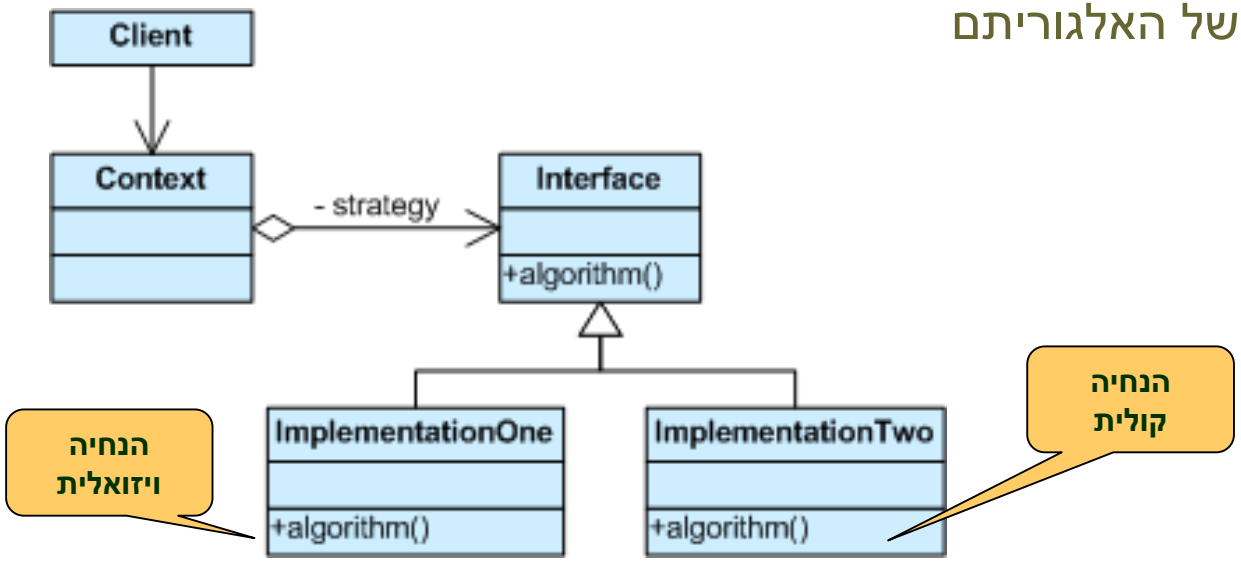
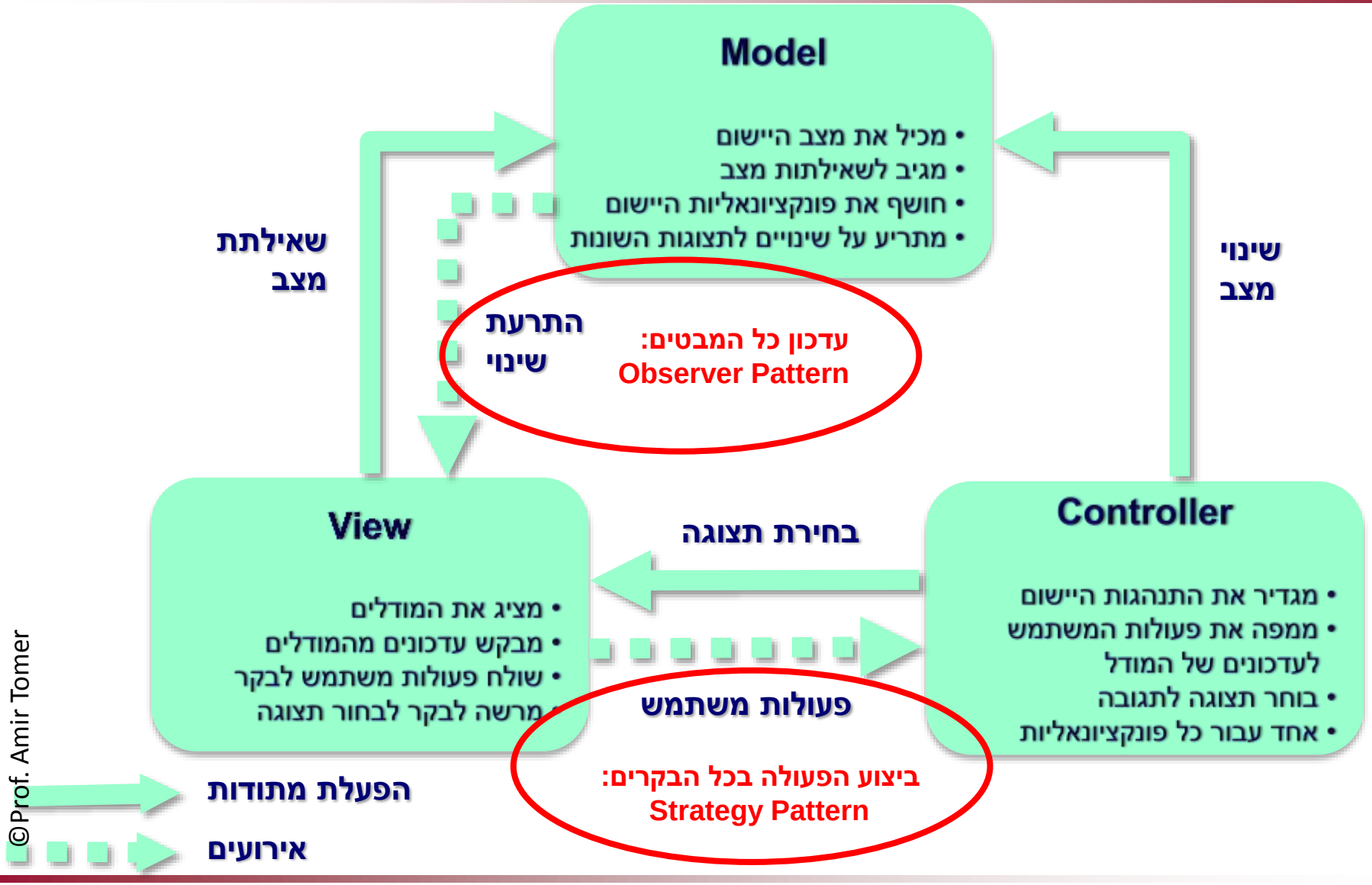
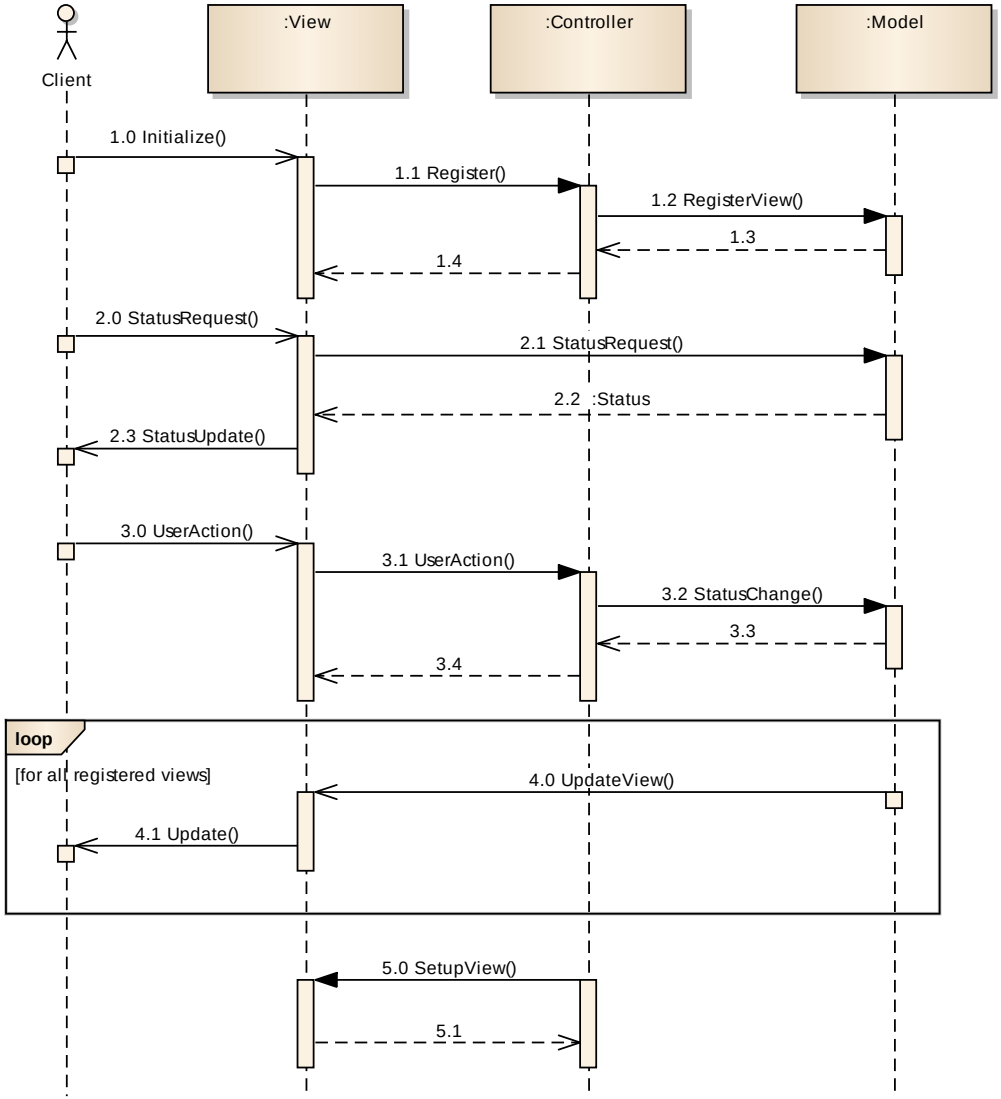


Diagram source: http://sourcemaking.com/design_patterns/strategy

MVC – המבנה הקונספטואלי



MVC – קונספט הפעולה (מודל דינאמי - תרחישים שונים)



1. איתחול ורישום המבט
2. שאילת יזומה למודל
3. פעולת משתמש לעדכון המודל
4. עדכון כל המבטים
5. בחירת תצוגה

אנטי-דפוסים

- דפוסי תכן נפוצים שיש להימנע מהם
- דוגמאות
  - God Object
    - מחלקה אחת ענקית המכילה את כל הפונקציונליות של התוכנית
      - תכנות פרוצדורלי במסווה של מונחה-עצמים
  - Poltergeist
    - אובייקט קצר-מועד המשמש כמתווך זמני להעברת מידע בין אובייקטים אחרים
  - Sequential coupling
    - מחלקה שדורשת שהמתודות שלה יופעלו על פי סדר מחייב
      - לדוגמה: Init, Begin, Start
  - Singletonitis
    - שימוש מופרז בדפוס-התכן Singleton
      - Singleton = מחלקה שקיים ממנה אובייקט בודד בתוכנה, לדוגמה: DataBase, Controller

הנושאים בפרק זה

- דפוסי תכן
- עקרונות לתכן יציב (SOLID)
- חלוקה למארזים וארגונם הנכון

SOLID – 5 עקרונות לתכן "יציב" מונחה עצמים

- **S**ingle Responsibility Principle (SRP)
- **O**pen Close Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

עיקרון האחריות היחידה – Single Responsibility Principle (SRP)

- לכל מחלקה יש אחריות אחת, ורק אחת
  - אחריות = סיבה להשתנות
    - לכל מחלקה צריכה להיות **סיבה אחת בלבד** להשתנות
- לדוגמה: מחלקה המייצגת הודעת דואר אלקטרוני
  - שתי סיבות לשינוי
    - שינוי בפורמט התוכן של הודעה (טקסט, HTML, ...)
    - שינוי בפרוטוקול הדואר (POP3, IMAP, ...)

מימוש הודעת דוא"ל – אחריות כפולה

אחריות למבנה ההודעה

```
interface IEmail {
public void setSender(String sender);
public void setReceiver(String receiver);
public void setContent(String content);
}

class Email implements IEmail {
public void setSender(String sender) {// set sender; }
public void setReceiver(String receiver) {// set receiver; }
public void setContent(String content) {// set content; }
```

אחריות לסוג התוכן

Source: www.oodesign.com

מימוש הודעת דוא"ל – מימוש יציב

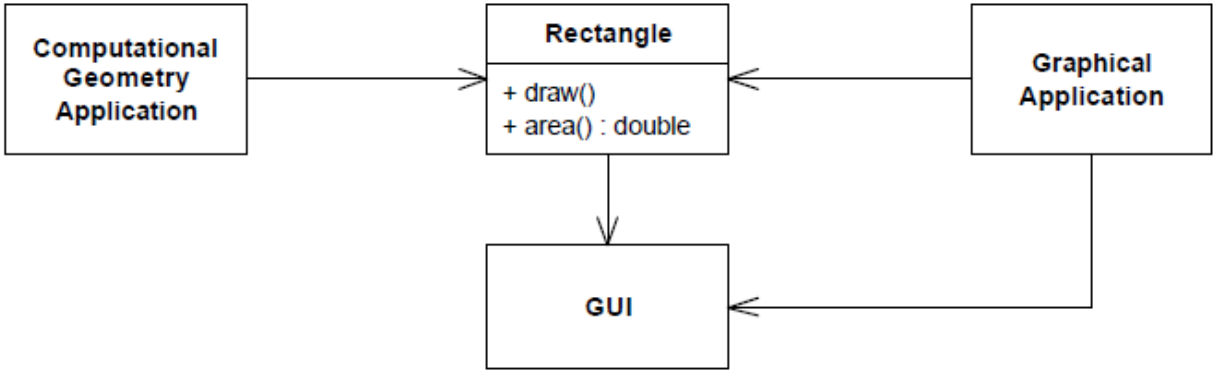
```
interface IEmail {interface IEmail {
public void setSender(String sender);
public void setReceiver(String receiver);
public void setContent(IContent content);
}

interface IContent {
public String getAsString();
}

class Content implements IContent {
public String getAsString(); // used for serialization
}

class Email implements IEmail {
public void setSender(String sender) {// set sender; }
public void setReceiver(String receiver) {// set receiver; }
public void setContent(Content content) {// set content; }
}
```

אחריות לסוג התוכן

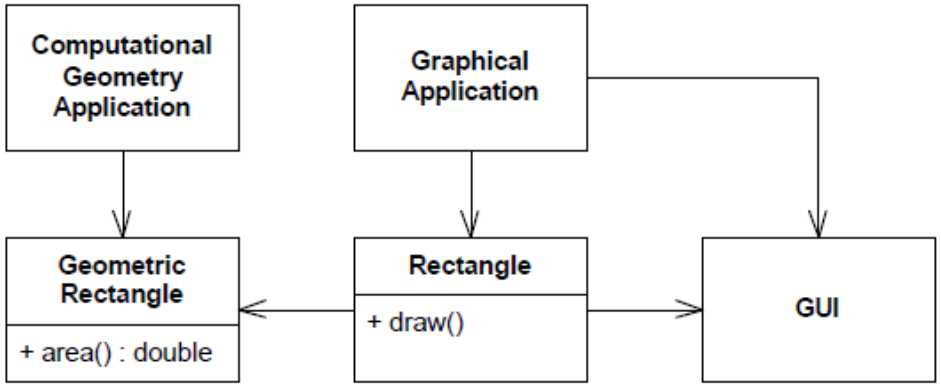
אחריות למבנה ההודעה

דוגמה נוספת – הפרדת אחריות בין הישות לבין ייצוגה

- אחריות כפולה



- הפרדת אחריות



Source: www.butunclebob.com

עיקרון האחריות היחידה: סיכום

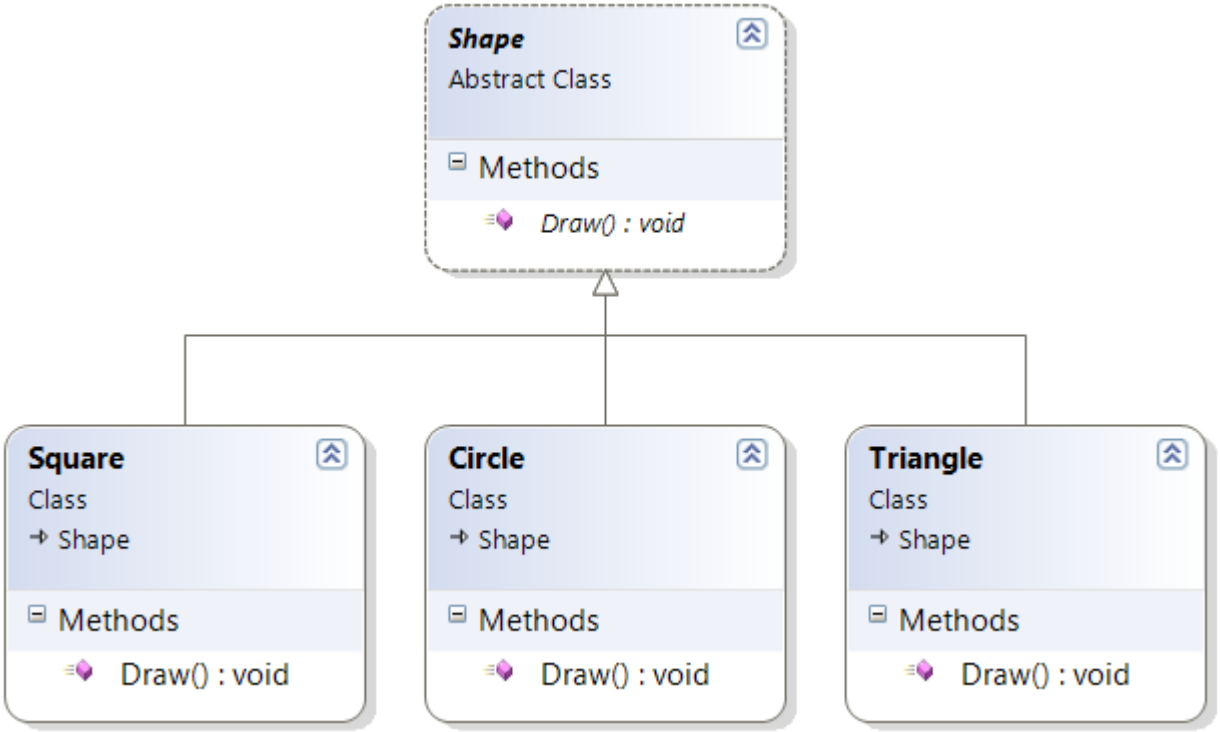
Source: www. themoderndeveloper.com

עיקרון הפתיחות-סגירות – Open-Close Principle (OCP)

- מחלקה צריכה להיות
  - פתוחה להרחבות
  - סגורה לשינויים
- המשמעות
  - שינויים צריכים להיעשות ע"י הוספת תת-מחלקות המממשים אותם

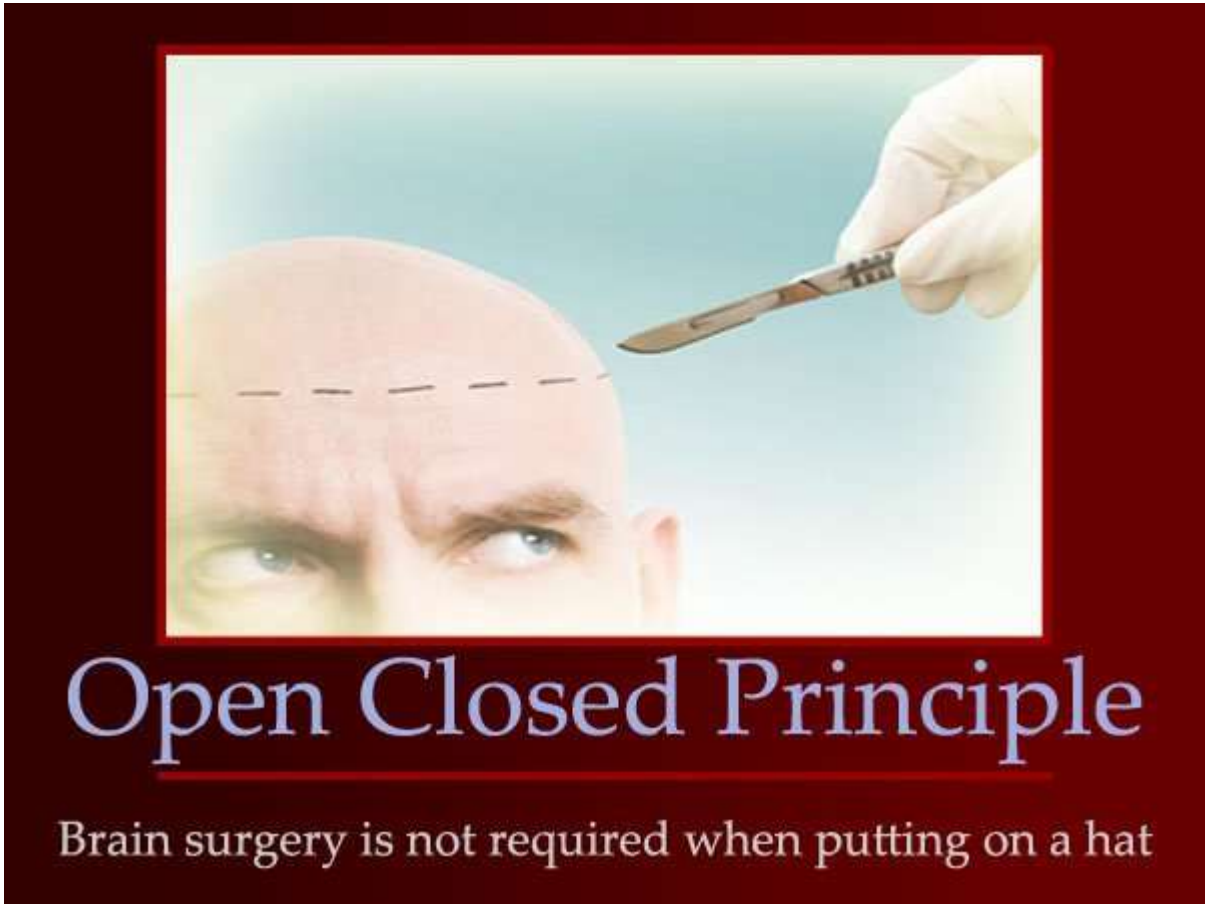
עיקרון הפתיחות-סגירות: דוגמה

- שימוש במחלקה אבסטרקטית ובתת-מחלקות ספציפיות



Source: www. jasondeoliveira.com

עיקרון הפתיחות-סגירות: סיכום



Source: www. themoderndeveloper.com

עיקרון ההחלפה של לישקוב – Liskov's Substitution Principle (LSP)

- אם S היא תת-מחלקה (יורשת) של T אזי ניתן להחליף כל מופע של T במופע של S מבלי שההתנהגות תשתנה
  - המשמעויות
    - מחלקה יורשת איננה יכולה לשנות את התנהגות מחלקת-האם
    - מי שפונה למתודה המוגדרת במחלקת האם איננו אמור לדעת לאיזו מהמחלקות-הבנות הוא פונה
- המקרה הקלאסי
  - האם ריבוע הוא מלבן? האם מעגל הוא אליפסה?
    - מבחינה מתימטית-גיאומטרית: הישות מימין היא מקרה פרטי של הישות משמאל
    - מבחינה הצהרתית-תכנותית: הישות הימנית היא ישות המוגדרת ע"י פרמטר אחד (אורך צלע, מרכז) בעוד שהשמאלית מוגדרת ע"י שני פרמטרים (אורך-רוחב, מוקדים)

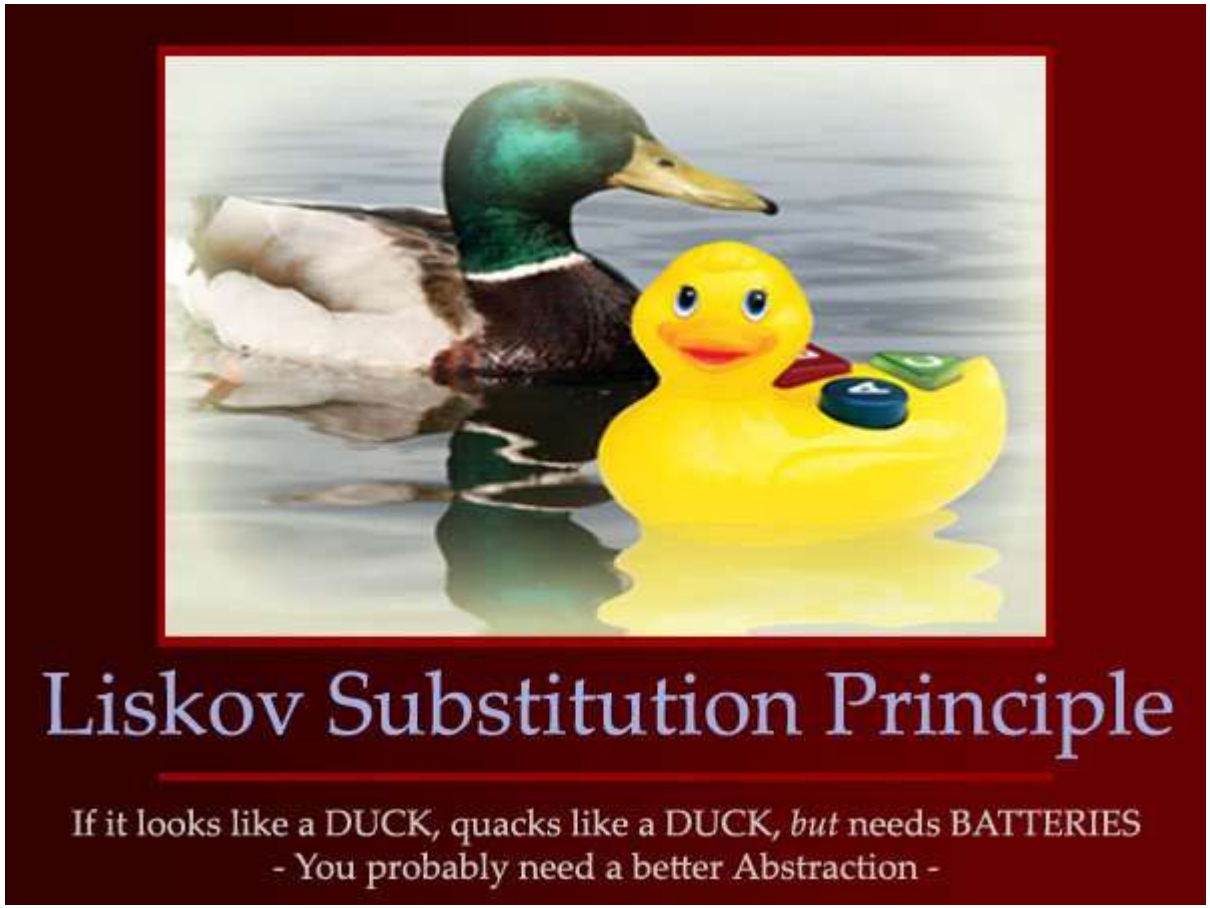
עיקרון ההחלפה של לישקוב: דוגמה

```
class Rectangle {
  protected int m_width;
  protected int m_height;

  public void setWidth(int width){
     m_width = width;
  }

  public void setHeight(int height){
     m_height = height;
  }

  public int getArea(){
     return m_width * m_height;
  }
}

class Square extends Rectangle {
  public void setWidth(int width){
     m_width = width;
     m_height = width;
  }

  public void setHeight(int height){
     m_width = height;
     m_height = height;
  }
}
```

"התחכמות"
ריבוע הוא מלבן
שארכו שווה
לרוחבו

```
class LspTest {
  private static Rectangle getNewRectangle() {
            /* it can be an object returned
                                 by some factory ... */
     return new Square();
}

  public static void main (String args[]) {
     Rectangle r = LspTest.getNewRectangle();
     r.setWidth(5);
     r.setHeight(10
     System.out.println(r.getArea
  }
}
```

עיקרון ההחלפה של לישקוב: סיכום



Source: www. themoderndeveloper.com

עיקרון היבדלות הממשקים – Interface Segregation Principle (ISP)

- אין להכריח לקוח להיות תלוי בממשק שהוא אינו משתמש בו
  - יש להחליף ממשק "שמן" בממשקים "רזים", כל אחד מותאם ללקוח ספציפי

עיקרון היבדלות הממשקים: דוגמה*

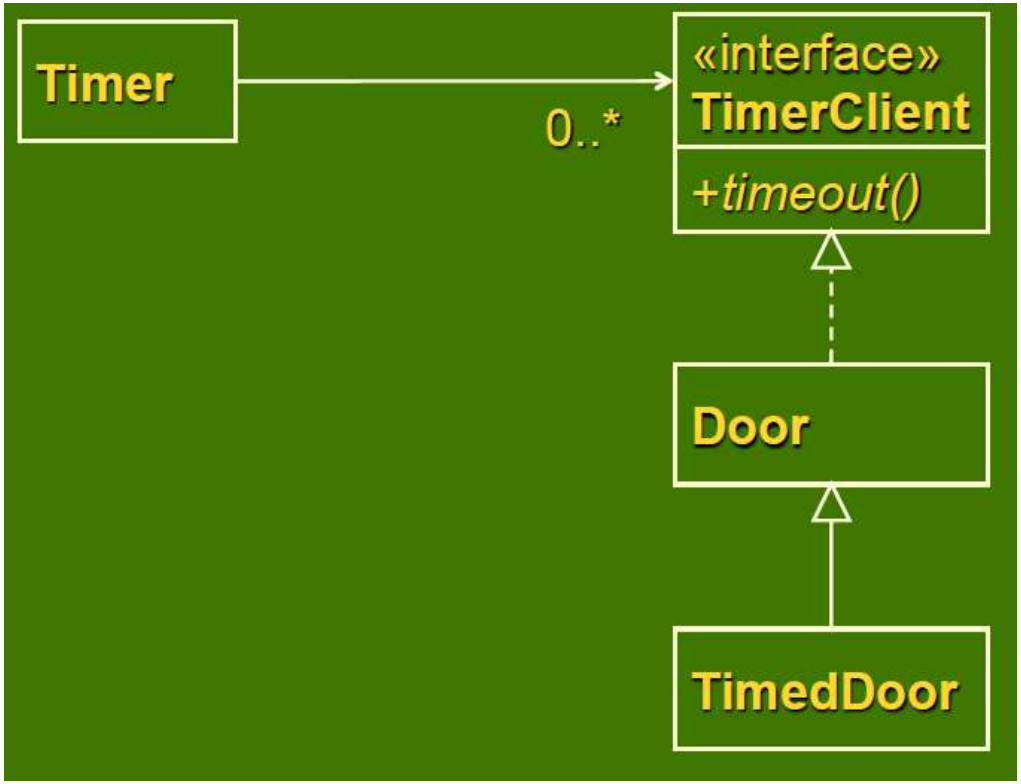
- דלת עם נעילה לאחר השהיה היא...
  - דלת רגילה
  - ישות עם מונה-זמן (timer)

```
public class Door {
   public void lock() { /* implementation */ }
   public void unlock() { /* implementation */ }
   public boolean isOpen() { /* implementation */ }
}

public class Timer {
   public void register(int timeout, TimerClient client) {
         /* implementation */
   }
}

public interface TimerClient {
   public void timeout();
}
```

*Source: http://www.slideshare.net/JouniSmed/designing-object-oriented-software-lecture-slides-2013

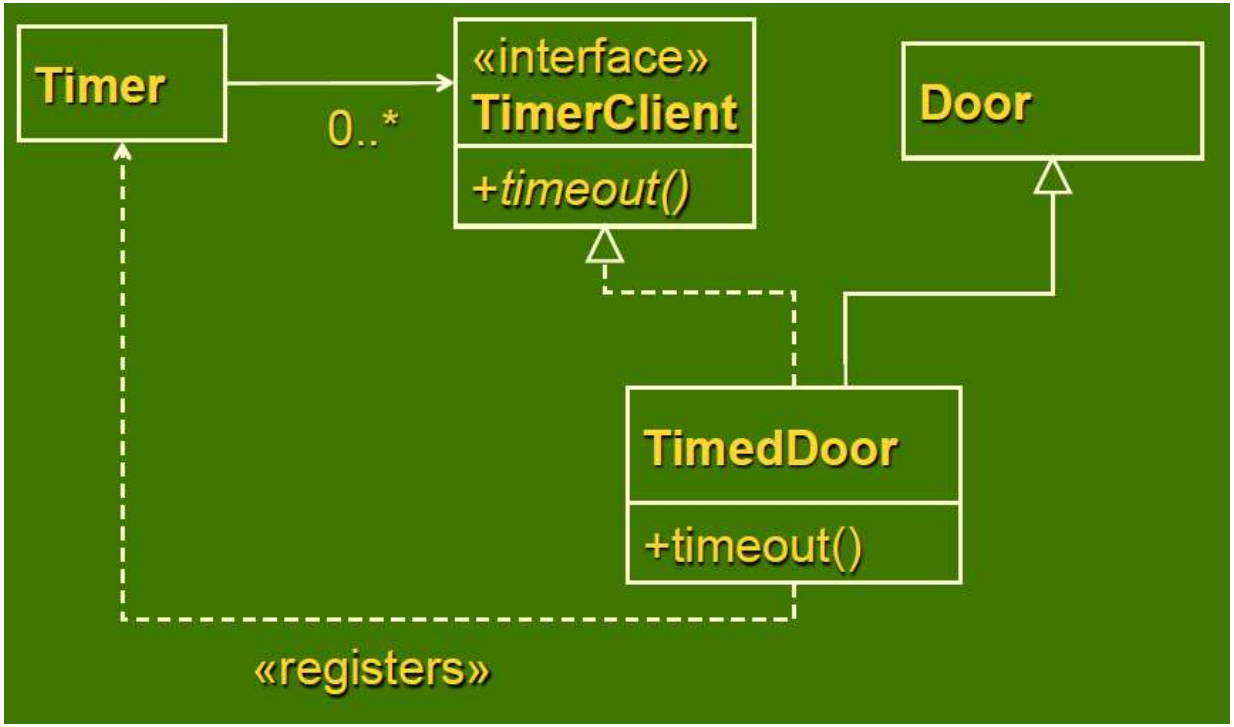
דלת מושהית: פתרון בעייתי

- "להכריח" כל דלת להיות דלת מושהית



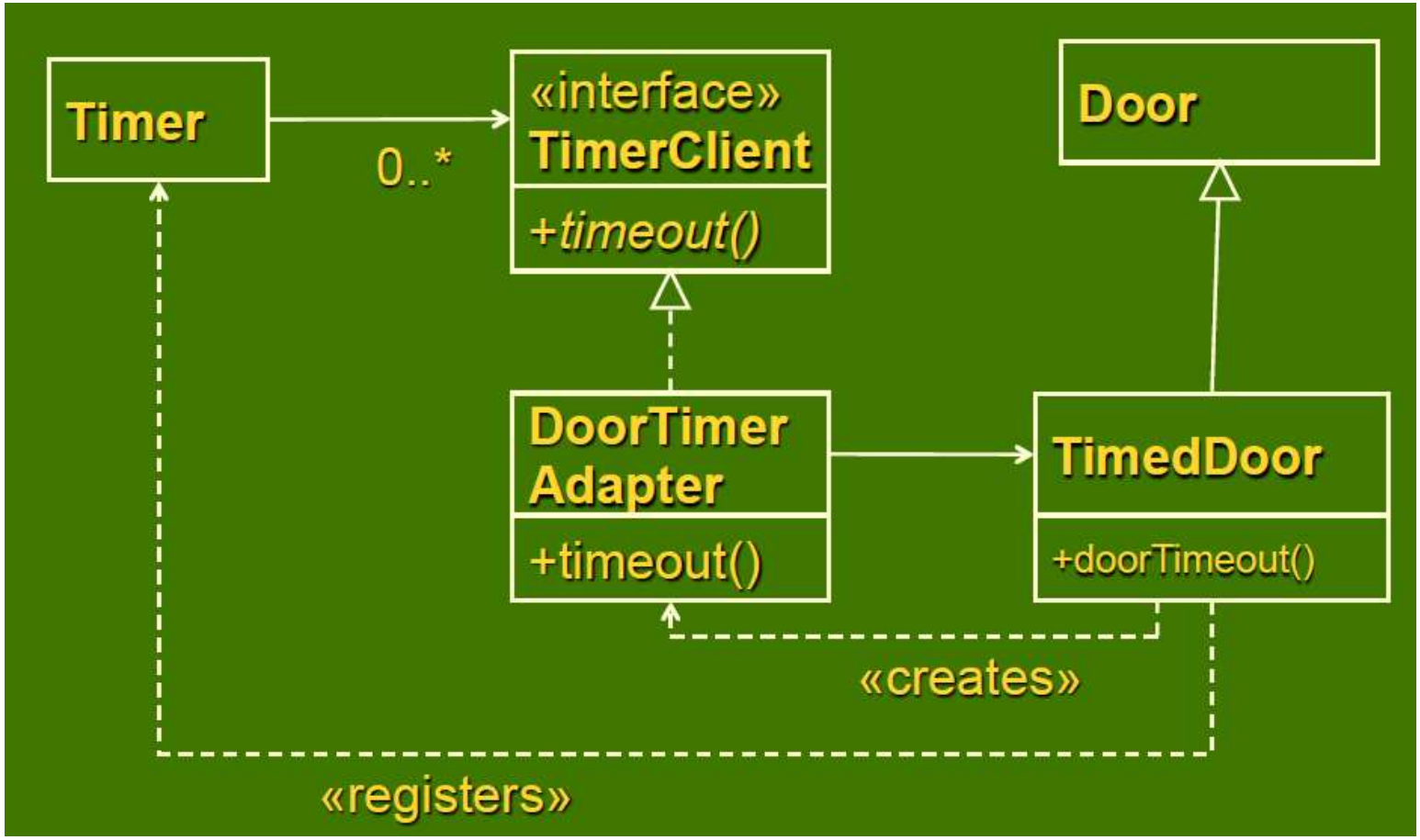
דלת מושהית: פתרון מבוסס על ירושה-מרובה (Multiple Inheritance)

- לא כל השפות תומכות בירושה מרובה



דלת מושהית: פתרון מבוסס על "שליח" (delegation)

- דפוס תכן "מתאם" (Adapter)

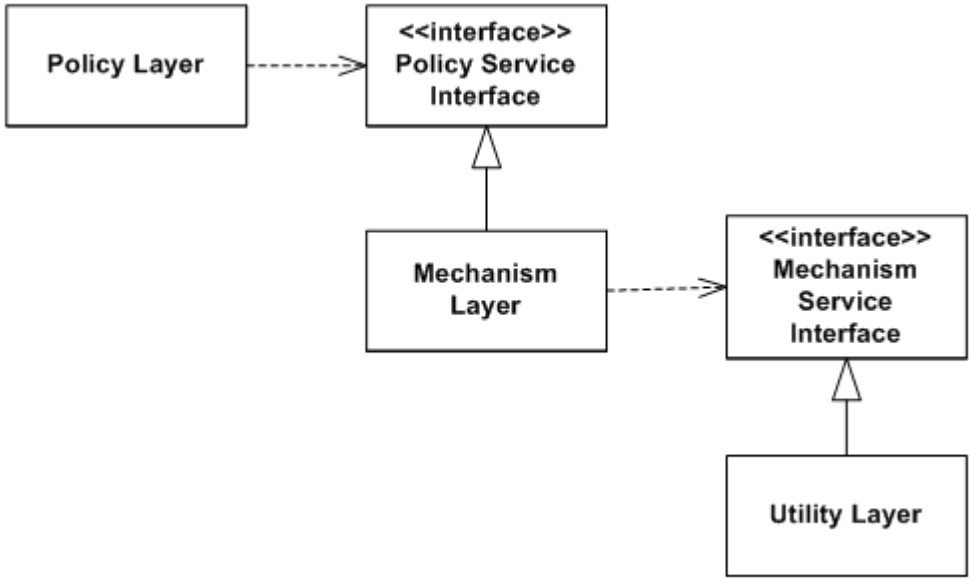


עיקרון היבדלות הממשקים: סיכום

Source: www. themoderndeveloper.com

עיקרון היפוך התלות – Dependency Inversion Principle (DIP)

1. מודולים ברמה גבוהה אינם צריכים להיות תלויים במודולים ברמה נמוכה
   - שניהם צריכים להיות תלויים באבסטרקציה
2. אבסטרקציות אינן צריכות להיות תלויות בפרטים
   - פרטים צריכים להיות תלויים באבסטרקציות



עיקרון היפוך התלות: סיכום

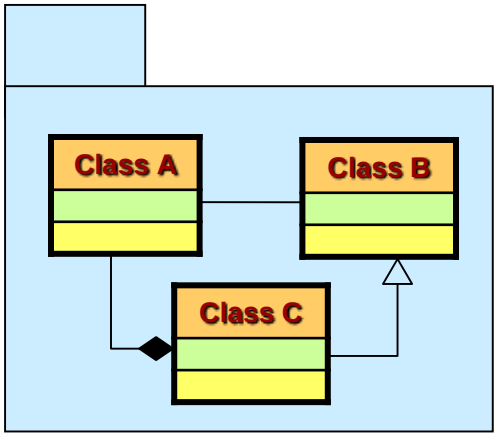
Source: www. themoderndeveloper.com

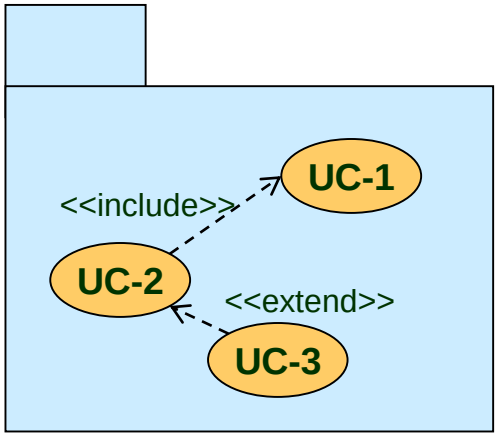
הנושאים בפרק זה

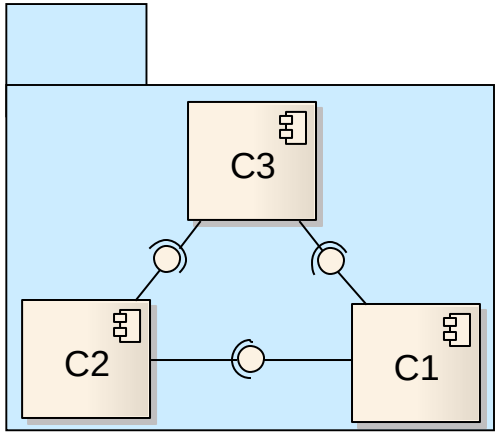
- דפוסי תכן
- עקרונות לתכן יציב (SOLID)
- חלוקה למארזים וארגונם הנכון

מודל המארזים

- מארז (Package) = אוסף של ישויות מאותו סוג
  - Use Cases, מחלקות, רכיבים, מארזים
- תלות (dependency) בין מארזים
  - אלמנטים ממארז אחד נדרשים "להכיר" אלמנטים ממארז אחר
- יש לוודא שהתלות בין המארזים איננה מעגלית
  - מפריע לייצוב הפיתוח
- תרשים מארזים (package diagram)
  - אוסף של מארזים והתלויות שביניהם

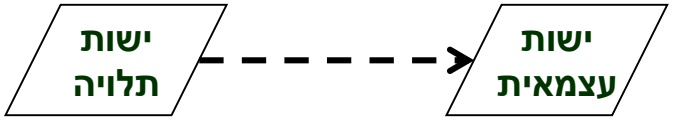




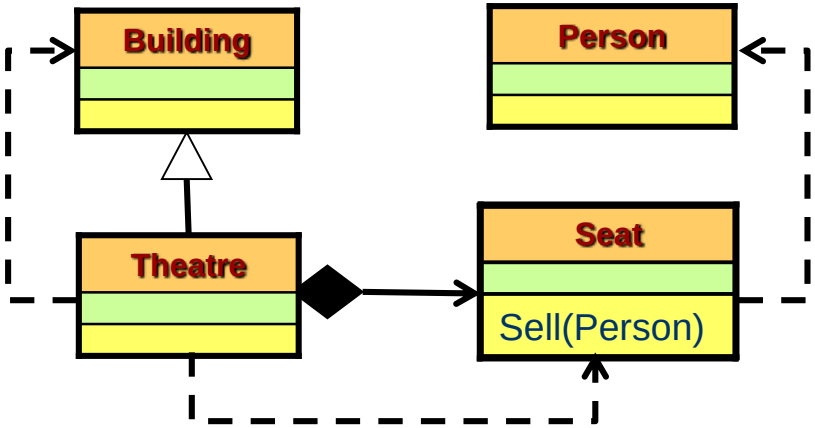


תלות - dependency

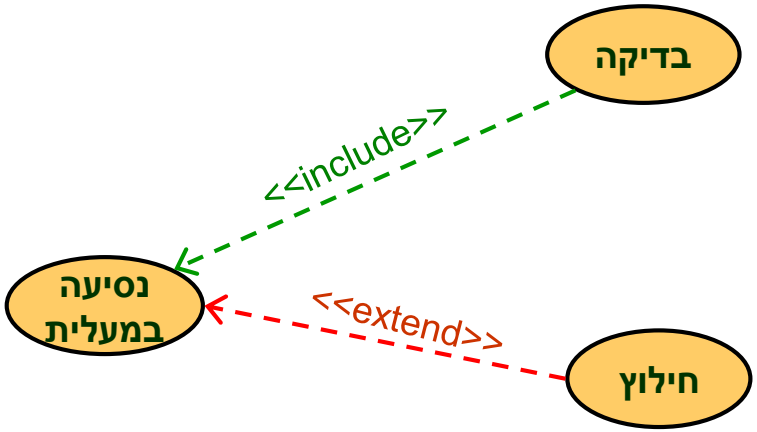
- יחס (relationship) בין שתי ישויות
  - ישות עצמאית
  - ישות תלויה
  - שינוי כלשהו בישות העצמאית עלול להשפיע על הישות התלויה
- תלות יכולה להתקיים גם בין ישויות שאין ביניהן זיקה (association) כלשהי
  - לדוגמה: עצם ממחלקה כלשהי מועבר כפרמטר לפונקציה של מחלקה אחרת



תלויות בין מחלקות

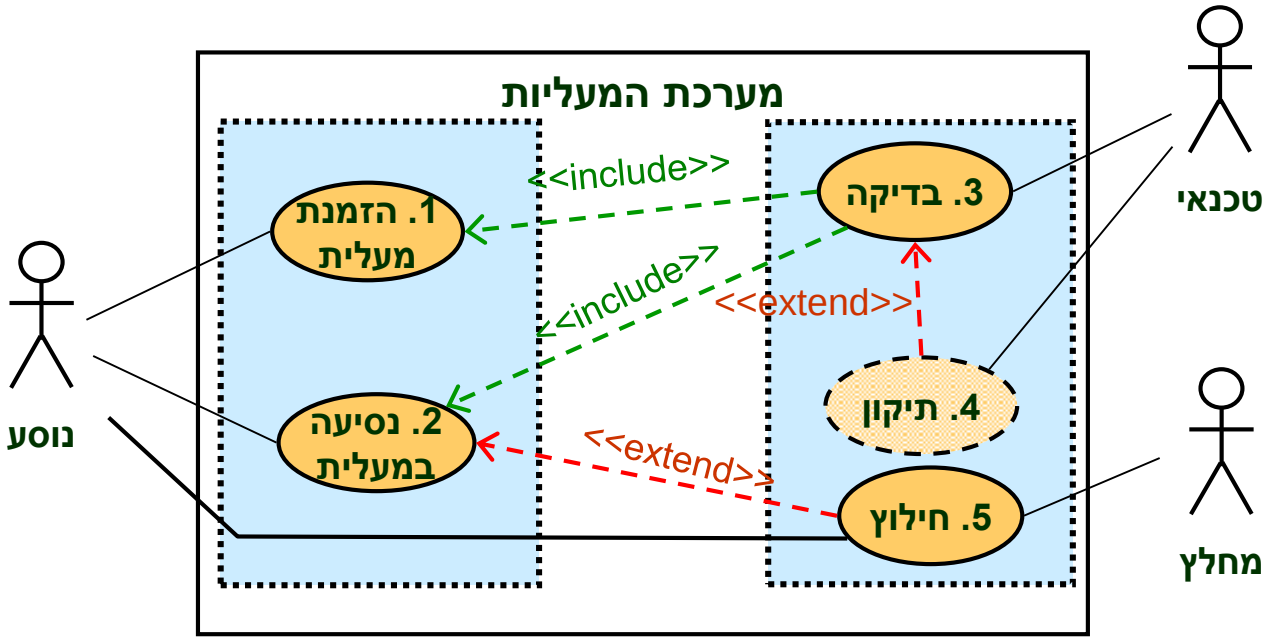


תלויות בין Use Cases

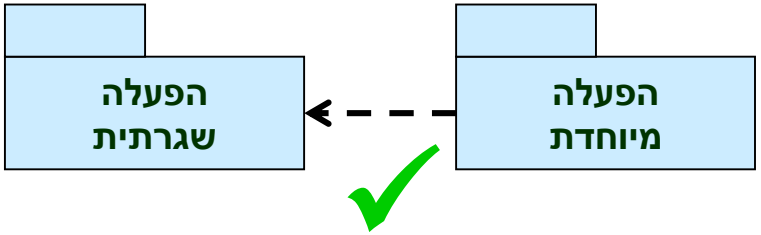


חלוקה למארזים על בסיס Use Cases (1)

- חלוקה לפי סוג תפעול

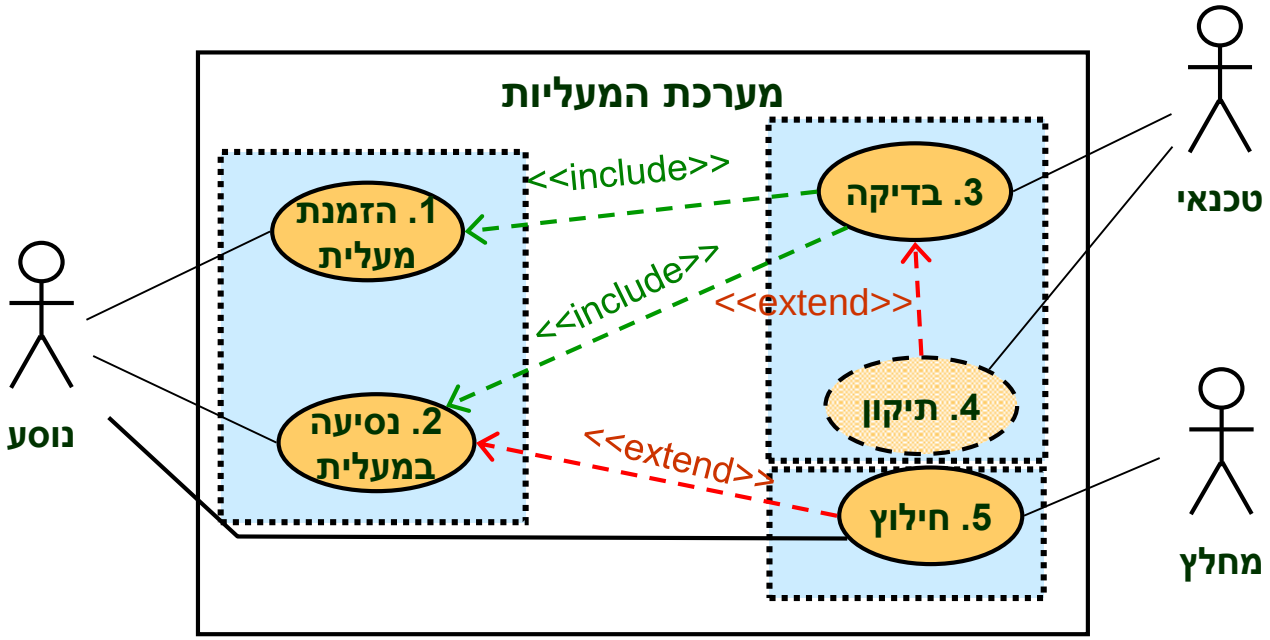


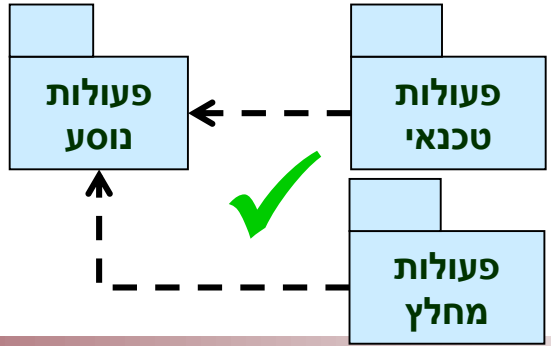
התלות הנוצרת בין המארזים:



חלוקה למארזים על בסיס Use Cases (2)

- חלוקה לפי שחקנים

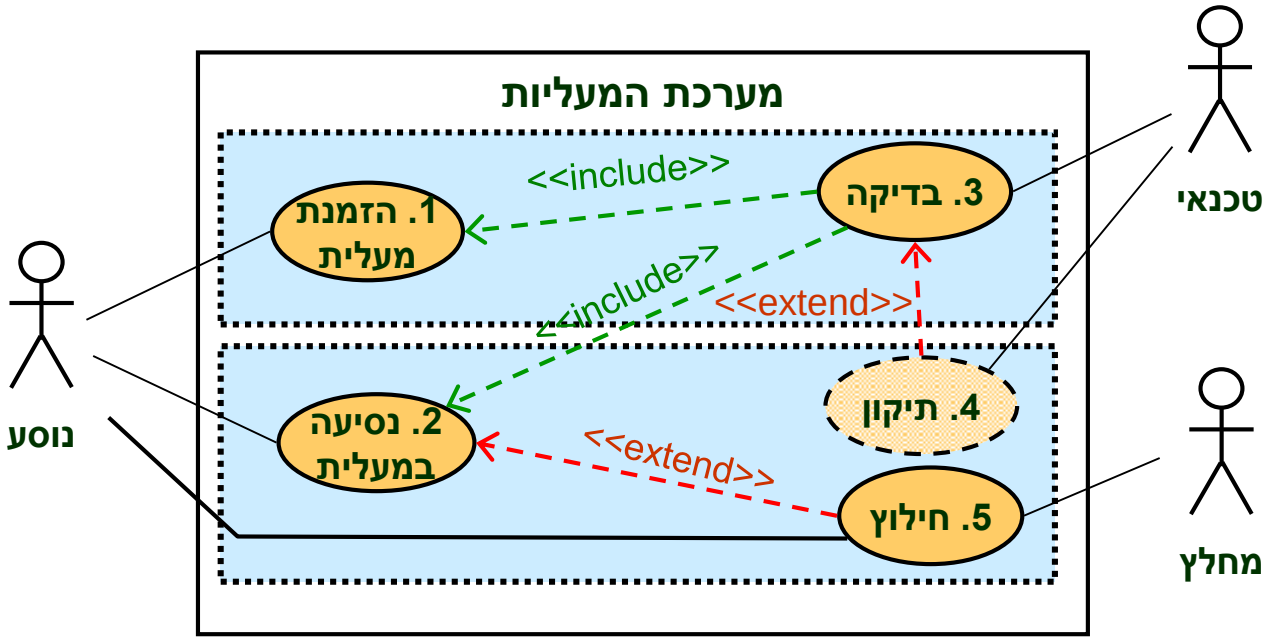


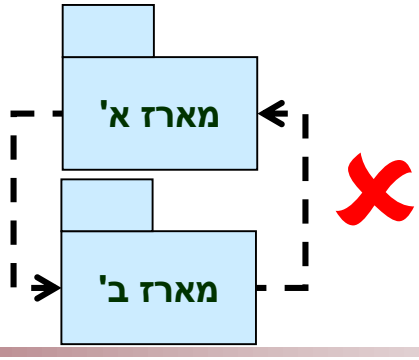


התלות הנוצרת בין המארזים:

חלוקה למארזים על בסיס Use Cases (3)

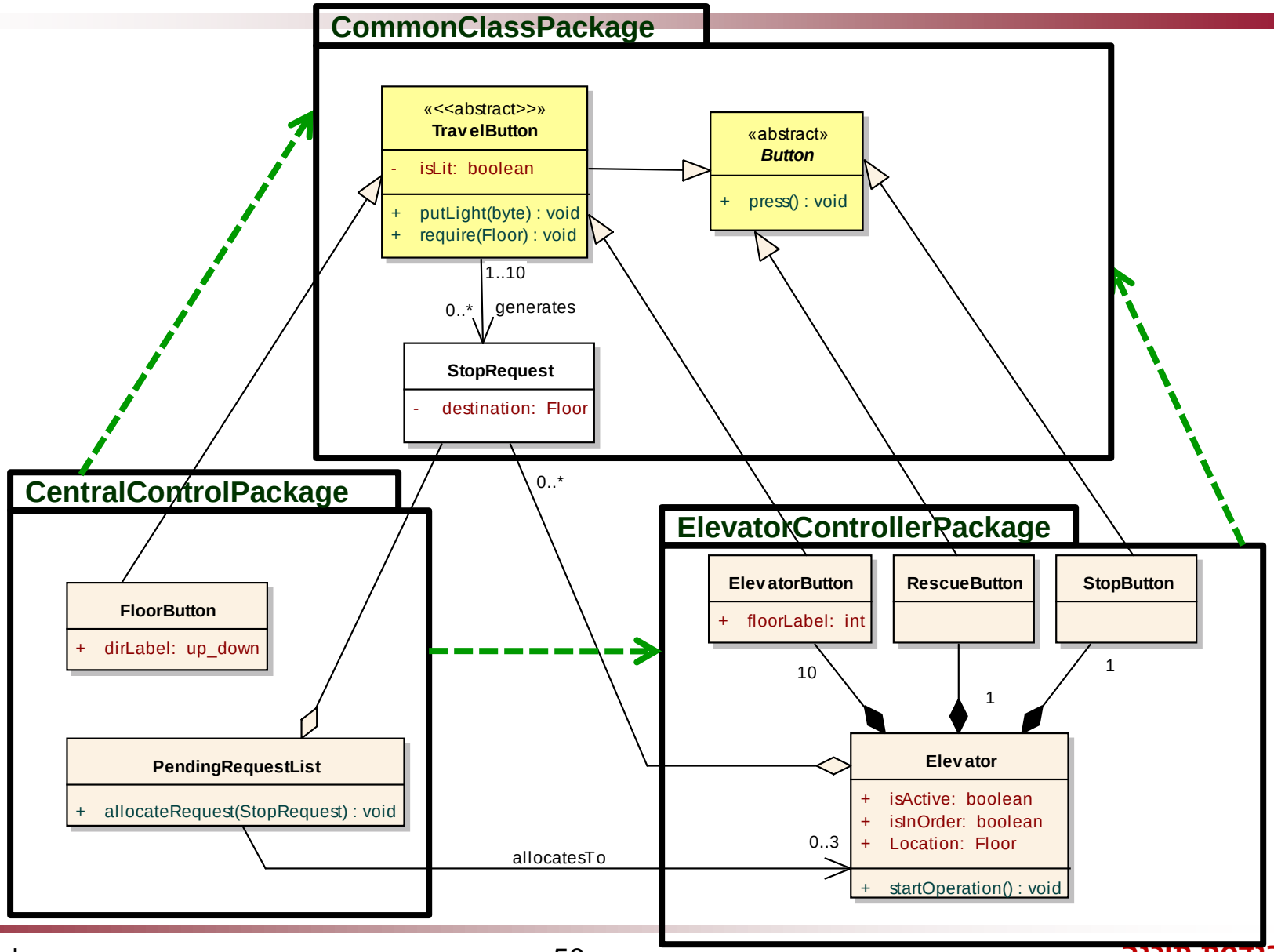
- חלוקה שרירותית





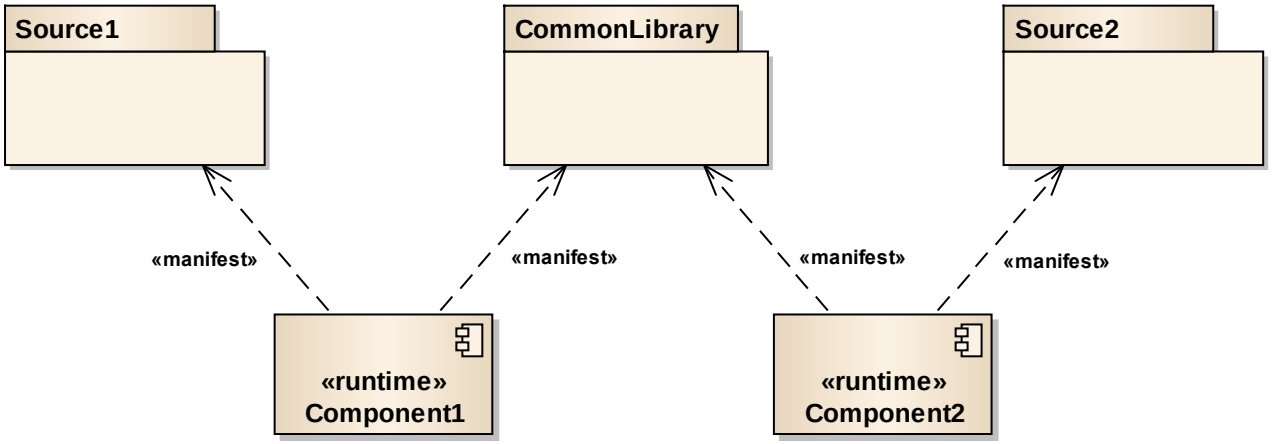
התלות הנוצרת בין המארזים:

מארזים של מחלקות – מחלקות "פרטיות" ו"משותפות"



ההבדל בין מארזים לרכיבים

- מארז = אוסף לכוד (cohesive) של **מחלקות**
  - למארז אין משמעות בזמן ריצה
  - מארז מוגדר ע"י תרשים מחלקות אחד או יותר (Class Diagrams)
- רכיב (זמן ריצה) = אוסף של **עצמים** המבצעים ביניהם אינטראקציה
  - משקף (manifests)את המארזים מהם נבנה (ע"י הידור וקישור)
  - פעולת הרכיב מוגדרת ע"י תרשים רצף אחד או יותר (Sequence Diagrams)



עקרונות חלוקה למארזים (של מחלקות)*

- עקרונות לשמירה על לכידות הדוקה (Tight Cohesion)
  - שקילות שחרור – שימוש-חוזר (Reuse-Release Equivalence)
  - שימוש-חוזר משותף (Common Reuse)
  - סגירות משותפת (Common Closure)
- עקרונות לשמירה על צימוד רופף (Loose Coupling)
  - תלויות לא-מעגליות (Acyclic Dependencies)
  - תלויות יציבות (Stable Dependencies)
  - אבסטרקציות יציבות (Stable Abstractions)

שקף זה, והבאים אחריו, מבוססים על
http://www.slideshare.net/JouniSmed/designing-object-oriented-software-lecture-slides-2013

עיקרון השקילות של שחרור / שימוש-חוזר
Reuse Release Equivalence Principle (REP)

The granule of reuse is the granule of release

- כל דבר שעושים בו שימוש-חוזר הוא גם משוחרר וגם נמצא במעקב
- כותב המארז חייב להבטיח
  - תחזוקה
  - הודעות על שינויים עתידיים
  - אופציה למשתמש לסרב לגרסאות חדשות
  - לתמוך בגרסאות ישנות לאורך תקופת זמן מסויימת
- עניין פוליטי מרכזי
  - התוכנה צריכה להיות מחולקת כך שתהיה נוחה לאנשים
- מארזים הניתנים לשימוש-חוזר חייבים להכיל מחלקות הניתנות לשימוש-חוזר
  - או שכל המחלקות במארז ניתנות לשימוש חוזר, או שאף אחת מהן
- ניתנות לשימוש חוזר – ע"י אותו קהל

עיקרון השימוש-החוזר המשותף – Common Reuse Principle (CRP)

The classes in a package are reused together

If you reuse one of the classes in a package, you reuse them all

- אם מחלקה אחת במארז משתמשת במחלקה במארז אחר, אזי יש תלות בין המארזים
  - בכל פעם שמשחררים את המחלקה שבשימוש חוזר, המארז המשתמש בה חייב לעבור תיקוף ויש לשחררו מחדש
  - תלות במארז פירושה תלות בכל המחלקות של אותו מארז!
- מחלקות בעלות זיקה הדוקה זו לזו צריכות להיות באותו מארז
  - בדרך כלל בין מחלקות אלה קיים צימוד הדוק
  - לדוגמה: מחלקה שהיא container וה-iterator
- המחלקות באותו מארז אמורות להיות בלתי-ניתנות להפרדה – לא ניתן לעשות שימוש-חוזר באחת ללא השניה

עיקרון הסגירות המשותפת – Common Closure Principle (CCP)

The classes in a package should be closed together against the same kind of changes

A change that affects a closed package affects all the classes in that package and no other packages

- חזרה על עיקרון האחריות היחידה (SRP) עבור מארזים
  - למארז לא תהיה יותר מסיבה אחת לשינוי
- תחזוקתיות חשובה לעיתים יותר מיכולת לשימוש-חוזר
  - כל השינויים אמורים להתרחש במארז אחד
  - מצמצם את היקף העבודה הקשורה בשחרור, תיקוף מחדש והפצה מחדש
- קשור לעיקרון הפתיחות-סגירות (OCP)
  - סגירות אסטרטגית: סגירות בפני סוגי שינויים צפויים
  - העיקרון מנחה לקיבוץ של מחלקות הפתוחות לאותו סוג של שינוי

עיקרון התלויות הלא-מעגליות – Acyclic Dependencies Principle (ADP)

Allow no cycles in the package dependency graph

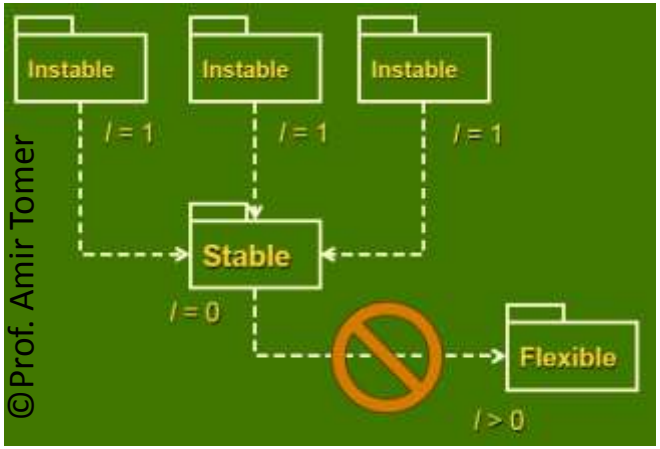
- ללא מעגלים קל יותר לקמפל, לבדוק ולשחרר bottom-up כאשר בונים את כל התוכנה
- המארזים במעגל יהפכו למארז אחד דה-פקטו
  - זמן הקומפילציה גדל
  - הבדיקות נעשות קשות כיוון שצריך לבנות build שלם כדי לבדוק מארז בודד
  - המפתחים "דורכים אחד לשני על הרגליים" כי הם חייבים להשתמש באותה גרסת מארזים של חבריהם

עיקרון התלויות היציבות – Stable Dependencies Principle (SDP)

Depend in the direction of stability

- התכן לא יכול להיות סטטי לחלוטין
  - נדרשת גמישות מסויימת כדי לתחזק את התכן
  - עיקרון הסגירות-המשותפת (CCP): חלק מהמארזים רגישים לסוגים מסויימים של שינויים
- מארז "גמיש" לא יכול להיות תלוי במארז שקשה לשנותו
  - מארז שתוכנן לעריכת שינויים בקלות עלול (בטעות) להפוך למארז בלתי ניתן לשינוי, כי מארזים אחרים תלויים בו!

- **מדד ליציבות של מארז (יציבות = קושי לשנות)**
  - C_a: צימודים פנימה (afferent)
    - מספר המחלקות מחוץ למארז שתלויות במחלקות בתוך המארז
  - C_e: צימודים החוצה (efferent)
    - מספר המחלקות בתוך מארז שתלויות במחלקות מחוץ למארז
  - $I = C_e / (C_e + C_a)$
    - $I = 0$: מארז יציב לחלוטין
    - $I = 1$: מארז בלתי-יציב (גמיש) לחלוטין
- **מדד היציבות של מארז חייב להיות גדול ממדד היציבות של מארז שהוא תלוי בו**

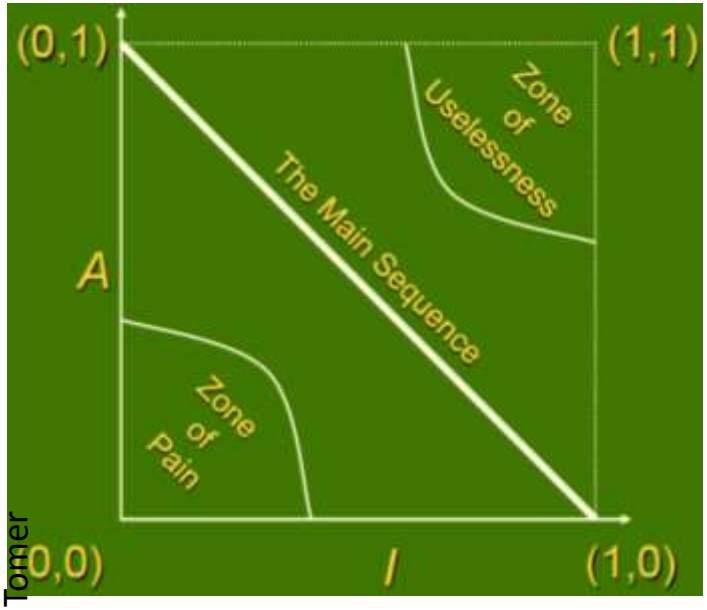


©Prof. Amir Tomer

עיקרון האבסטרקציות היציבות – Stable Abstractions Principle (SAP)

A package should be as abstract as it is stable

- מארז יציב צריך להיות אבסטרקטי, על מנת שלא למנוע את הרחבתו
- מארז לא יציב (גמיש) חייב להיות קונקרטי, כי הגמישות מאפשרת לקוד קונקרטי להשתנות בקלות
- SDP + SAP = DIP
  - כיוון התלויות הוא ככיוון האבסטרקציות



- **מדד לאבסטרקציה של מארז**
  - N_c : מספר המחלקות במארז
  - N_a : מספר המחלקות האבסטרקטיות במארז
  - $A = N_a / N_c$
    - $A = 0$: אין מחלקות אבסטרקטיות
    - $A = 1$: יש רק מחלקות אבסטרקטיות