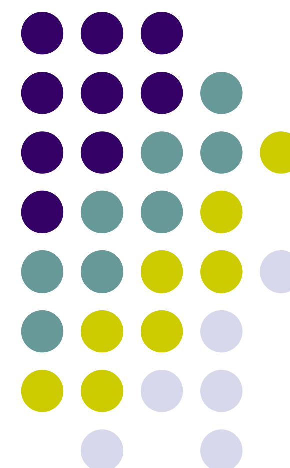
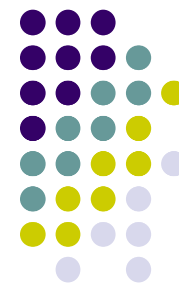
תכנות אדפטיבי דינאמי

הפרדת מטרות על ידי שימוש בתורת הגרפים

ניר אדר

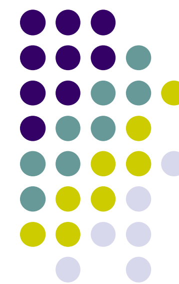


תקציר (1)



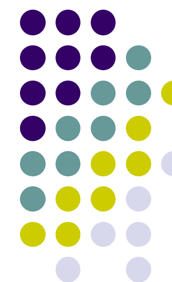
- **תכנות מונחה עצמים** יוצר תלות חזקה בין עץ המחלקות למימוש התוכנית.
- **דוגמא**: אוטובוס לו מספר תחנות, בכל תחנה מספר נוסעים ממתינים.
  - כמה נוסעים ממתינים ישנם? נממש מתודה העוברת על התחנות וסופרת את הנוסעים.
  - אם נציע ממשק שונה (למשל, רשימה אחת של נוסעים) כל הפונקציות המניחות שהנוסעים מסודרים לפי התחנות יצטרכו לעבור שינוי.

תקציר (2)



- פרדיגמת תכנות המכונה **תכנות אדפטיבי** מציעה פתרון לבעיה.
- ניתן להתייחס לפרדיגמה זו כאל מקרה פרטי של **תכנות מונחה אספקטים**.
- בהרצאה:
  - החוק של Demeter – כלל אצבע המאפשר תכנות OOP בעל תלות מופחתת בעץ המחלקות.
  - עקרונות התכנות האדפטיבי.
  - השוואה בין תכנות אדפטיבי לתכנות מונחה אספקטים.

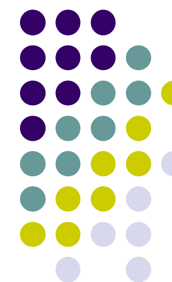
החוק של Demeter (1)



- **החוק הכללי של Demeter:**

רכיבי התוכנית השונים צריכים להכיר רק את רכיבי התוכנית האחרים הקשורים אליהם באופן קרוב ("closely related").

- המטרה: הורדת התלות של מחלקות במימוש ובממשק של מחלקות אחרות.
- **מערכות Loosely coupled** - ניתן לקחת חלק אחד מהמערכת ולהחליפו באחר באופן שקוף למשתמש.

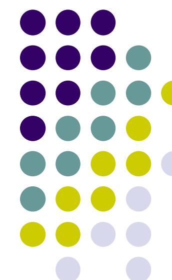


החוק של Demeter (2)

- **החוק של Demeter עבור תכנות מונחה עצמים:**

מתודה M השייכת למחלקה C צריכה להשתמש אך ורק במתודות השייכות לארגומנטים שלה (כולל this), במתודות השייכות לאובייקטים הנוצרים על ידי M וברכיבים הישירים של המחלקה C.

החוק של Demeter (3)



- מחלק העיתונים, הקונה והארנק

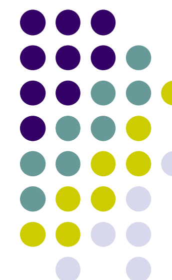
```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;

    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public Wallet getWallet() {
        return myWallet;
    }
}
```

```
public class Wallet {
    private float value;

    public float getTotalMoney() {
        return value;
    }
    public void setTotalMoney(float newValue) {
        value = newValue;
    }
    public void addMoney(float deposit) {
        value += deposit;
    }
    public void subtractMoney(float debit) {
        value -= debit;
    }
}
```

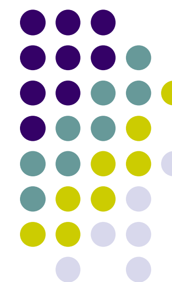
החוק של Demeter (4)



```
payment = 200; // "I want my two dollars!"
Wallet theWallet = myCustomer.getWallet();
if (theWallet.getTotalMoney() > payment)  {
    theWallet.subtractMoney(payment);
}
else {
    // come back later and get my money
}
```

- האם אתה היית נותן למחלק העיתונים את הארנק שלך?
- <u>מחלק העיתונים חשוף ליותר מידע מהדרוש לו.</u>
- תלות בין מחלקת הארנק למחלקת מחלק העיתונים.
- ואם גנבו לקחת את הארנק? (getWallet() == null)

(5) Demeter החוק של



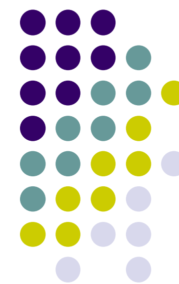
```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public float getPayment(float bill) {
        if (myWallet != null) {
            if (myWallet.getTotalMoney() > bill) {

theWallet.subtractMoney(payment);
                return payment;
            }
        }
        else {
            // come back later and get my money
        }
    }
}
```

- החלפת המתודה getWallet() במתודה getPayment().
- מודל הקרוב יותר למציאות.
- ביטול הקשר בין מחלק העיתונים לארנק.
- תכנות מונחה עצמים: המימוש של getPayment() יכול להשתנות.

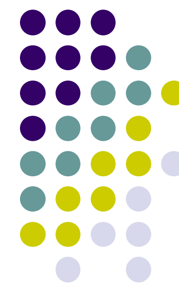
```
payment = 200; // "I want my two dollars!"
paidAmount =
myCustomer.getPayment(payment);
if (paidAmount == payment) {
    // say thank you and give customer a
receipt
} else {
    // come back later and get my money
}
```

החוק של Demeter (6)



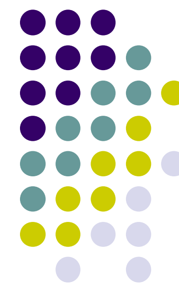
- החוק של Demeter אכן מאפשר תחזוקה קלה יותר של תוכנית על ידי הפחתת התלות בין חלקי התוכנית השונים.
- אנחנו משלמים בכך שקוד המימוש של concern כלשהו בתוכנית יתפרש על פני מחלקות רבות (*code scattering*).

(1) Adaptive Programming



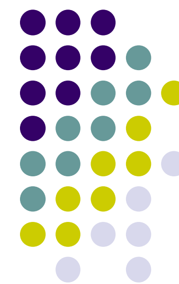
- <u>הפתרון לבעיה</u>: שימוש בפרדיגמה הנקראת "תכנות אדפטיבי" – **Adaptive Programming** או בשם נוסף **Shy Programming**.
- <u>אבחנה</u>: האבסטרקציה של מודול כ"קופסה שחורה" איננה מספיק טובה!
- "קופסה שחורה" מפרידה בין הממשק למימוש, אולם היא אינה מפרידה בין הממשק לאלו המשתמשים בו.
- תכנות אדפטיבי מאפשר את שינוי הממשק ללא שינוי מימוש התוכנית.

(2) Adaptive Programming



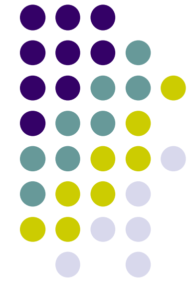
- **מצב התוכנית** בזמן ריצה, בייחוד מצב של תוכניות מונחות עצמים, יכול להיות מיוצג על ידי גרף מכוון.
- הצמתים הינם אובייקטים. הקשתות בין האובייקטים מבטאות קשרי has-a וקשרי is-a של המחלקות המתאימות לאובייקטים.
- נתייחס למהלך ריצת כל התוכנית **כטיול על הגרף**.

(3) Adaptive Programming



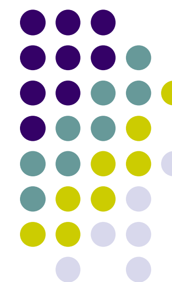
- **הגדרות אפשרויות לטיול על הגרף:**
  1. קבוצת האובייקטים אליהם ניתן להגיע מאובייקט נתון, שלהם מאפיין מסוים, נצפו במהלך הטיול.
  2. בוצעה פעולה כלשהי על תת קבוצה של קבוצת האובייקטים שניתן להגיע אליהם מאובייקט נתון.
- concern אשר מבצע טיול בין אובייקטים לצורך מימוש התנהגות מסוימת על אובייקטים יכונה **traversal-related concern**.
- תוכנית טיפוסית הפועלת על קבוצות גדולות של אובייקטים מכילה concerns רבים מסוג זה.

(4) Adaptive Programming



- עבור דוגמת האוטובוס והנוסעים:
  - שורש הגרף שלנו מתחיל באובייקט מסוג *אוטובוס*.
  - אנו מעוניינים בכל הצמתים הישיגים שהם מסוג *נוסע*.
  - ניתן להגדיר את האלגוריתם הכללי לפתרון הבעיה כך: "ספור את כל האובייקטים מסוג *נוסע* שניתן להגיע אליהם דרך האובייקט B (שהוא מסוג *אוטובוס*)".
  - ניתן לנסח אלגוריתם כזה בשפות תכנות!

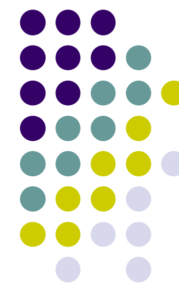
(5) Adaptive Programming



- תכנות אדפטיבי מסתמך על **החוק של Demeter עבור Concerns**:

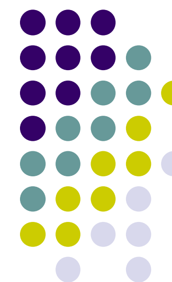
בעת מימוש של concern, אובייקט צריך לתקשר רק עם אובייקטים החולקים איתו באופן מובהק את אותו ה-concern.

(6) Adaptive Programming



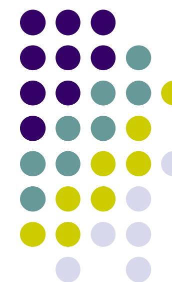
- **הספרייה DJ** היא ספריית מחלקות המיועדות להפוך את הגדרת הטיול בגרף לקלה להגדרה, להבנה ולתחזוקה.
- נכתבה על ידי צוות Demeter.
- הדוגמאות בהמשך - יעשו שימוש בספרייה זו.
- מטרות העל של ספריית DJ:
  - טיפול ב-concerns מסוג traversal-related concerns.
  - ביטול התלות בין מטרות הטיול לבין עץ המחלקות.
  - אפשור תכנות אדפטיבי בצורה ידידותית למתכנת.

(7) Adaptive Programming

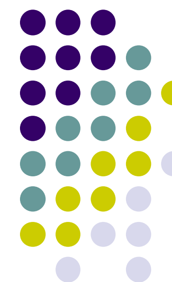


- **תכנות אדפטיבי דינאמי** הוא תכנות בו ההחלטה על מסלול הטיול בגרף המחלקות נעשית בזמן ריצה.
- ספריית DJ מממשת תכנות אדפטיבי דינאמי.
- הספרייה משתמשת ב-**Reflections** - טכנולוגיה המאפשרת, בהינתן מחלקה, לזהות בזמן ריצה את המתודות והמאפיינים שלה, ובאופן דינמי להריץ מתודות השייכות לה.
- בעזרת השימוש ב-Reflections הספריה אינה דורשת התערבות בפעולת המהדר של Java.

(8) Adaptive Programming

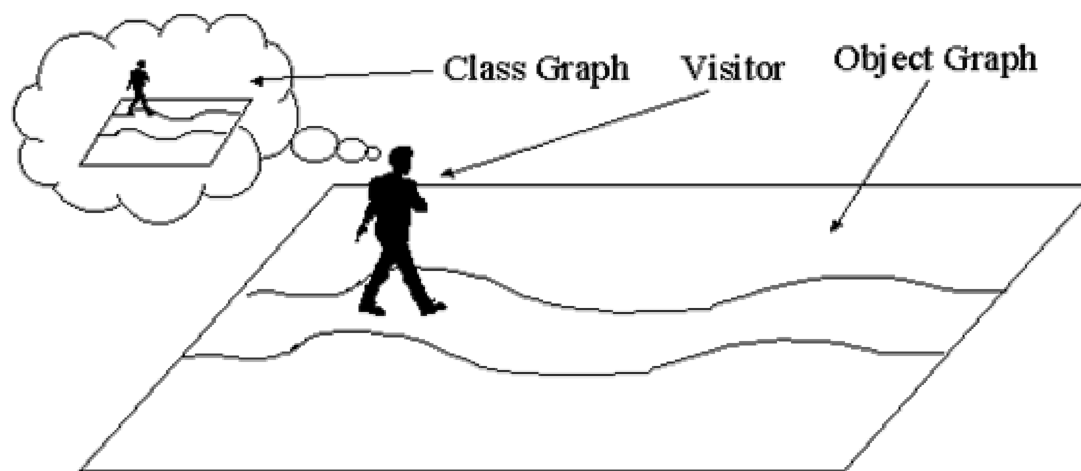


- האלמנטים עליהם מושתתת החבילה DJ הן המחלקות ClassGraph, ObjectGraph, Strategy ו-Visitor.
- אובייקט מסוג **ClassGraph** הוא גרף מכוון בו הצמתים הם <u>מחלקות</u>, והקשתות מבטאות קשרי is-a וקשרי has-a בין המחלקות המתאימות.
- אובייקט מסוג **ObjectGraph** הוא גרף מכוון בו הצמתים הם <u>האובייקטים</u> של התוכנית. הגרף מתאים לאובייקטים הקיימים בזמן ריצה.

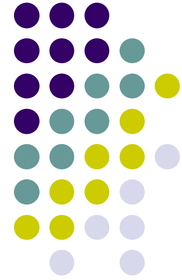


(9) Adaptive Programming

- **Strategy** מגדירה את הטיול שאנו רוצים לבצע. מחלקה זו מסוגלת להיות מתוארת גם כמחרוזת.
- אובייקטים מסוג **Visitor** שומרים את המידע הנדרש ליישום הטיול.



(10) Adaptive Programming

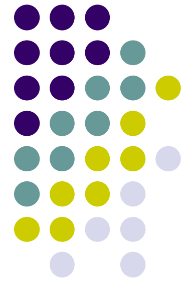


```
class Company {
    Double sumSalaries(edu.neu.ccs.demeter.dj.ClassGraph cg) {
        String s = "from Company to Salary";
        edu.neu.ccs.demeter.dj.Visitor v = new edu.neu.ccs.demeter.dj.Visitor() {
            private double sum;

            public void start() {
                sum = 0.0;
            }
            public void before(Salary host) {
                sum += host.getValue();
            }
            public Object getReturnValue() {
                return new Double(sum);
            }
        }

        return (Double) cg.traverse(this, s, v);
    }
}
```

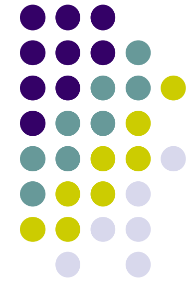
(11) Adaptive Programming



- מתודות בהם תומך Visitor:

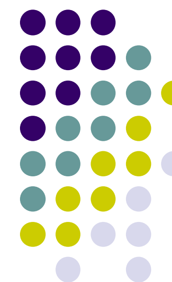
1. public void start()
2. public void finish()
3. public void before(<Class>)
4. public void after(<Class>)
5. public Object getReturnValue()

(12) Adaptive Programming



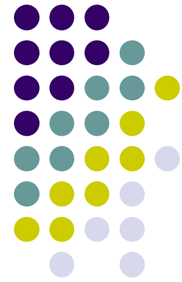
- המחלקה Strategy מגדירה את נקודת המוצא, נקודת היעד, ואופציונאלית גם תחנות ביניים.
- ניתן להשתמש במחרוזת בכל מקום שמשתמשים במחלקה זו.
  - ניתן להתייחס למחרוזת כאל ביטוי רגולארי המתאים לאסטרטגיה.
  - ניתן לבנות ולהגדיר אסטרטגיות בזמן ריצה.

(13) Adaptive Programming



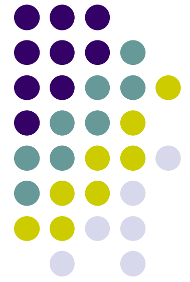
- **דוגמא**: שמירת קובץ XML המייצג מחלקה.
- **דוגמא**: caching של ערכי פונקציה מוחזרים.
  - לכאורה, caching של ערכי פונקציה מוחזרים נראה פעולה שהגיוני לבנות עבורה תבנית גנרית.
  - בפועל – בשיטות התכנות המקובלות קשה לממש זאת, מכיוון שקיימים מקרי קצה רבים – מתי צריך לקרוא לפונקציה שוב ולא להשתמש בערך שנשמר.
  - AP מאפשר לנו לבצע caching בקלות – נגדיר עצמים בגרף שכאשר נעבור דרכם – נחשב את הערך מחדש. עבור העצמים האחרים – נשתמש בערך שחושב ונשמר.

(14) Adaptive Programming



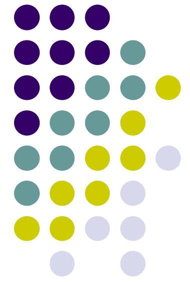
- כלים נוספים המאפשרים תכנות אדפטיבי:
- **DemeterJ** - סביבת עבודה המרחיבה את Java לצורך תמיכה בתכנות אדפטיבי. הטיול מוגדר בזמן הידור הקוד.
- **DAJ** – תוסף ל-AspectJ המאפשר תכנות אדפטיבי. הטיול מוגדר בזמן הידור הקוד. התחביר מאוד דומה לזה של DJ.

תכנות מונחה אספקטים ושיטות אדפטיביות (1)



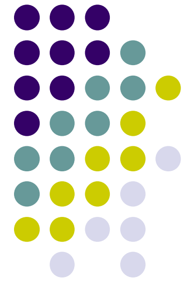
- תכנות אדפטיבי הופיע מספר שנים לפני תכנות מונחה אספקטים.
- עם זאת, מתייחסים אל תכנות אדפטיבי כאל מקרה פרטי של תכנות מונחה אספקטים.
- תכנות אדפטיבי הוא תכנות מונחה אספקטים בו concern אחד לפחות מבוטא על ידי מושגים מתורת הגרפים.

תכנות מונחה אספקטים ושיטות אדפטיביות (2)



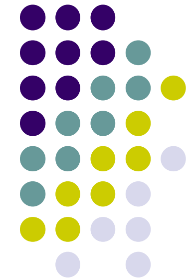
- גם תכנות מונחה האספקטים וגם תכנות אדפטיבי באים לטפל בבעיה זו על ידי הצגת יחידות מודלריות להפרדת ה-crosscutting concerns.
- תכנות מונחה אספקטים - "**מודולריות טובה יותר עבור concerns**".
- תכנות אדפטיבי - "**concern-shyness**" – הסתרת פעולת ה-concerns מ-concerns אחרים.
- תכנות אדפטיבי מתרכז ב- traversal-related concern וב- structural concern.

הרחבות ומידע נוסף



http://underwar.livedns.co.il

מקורות



1. Baris Aktemur (2004), "**Aspect Oriented Programming**"
2. Bradford Appleton (1996), "**Introducing Demeter and its Laws**"
3. Bradford Appleton (1995), "**The Law of Demeter for Propagation Patterns**"
4. David Bock (2000), "**The Paperboy, The Wallet, and The Law Of Demeter**"
5. Doug Kaye (2002), "**Web Services Strategies**"
6. Karl Lieberherr, Doug Orleans (2000), "**A Short Introduction to Adaptive Programming**"
7. Karl Lieberherr (2003), "**AOSD and the Law of Demeter: Shyness in Programming**"
8. Karl Lieberherr, and Holland, I. (1989), "**Assuring good style for object-oriented programs**", IEEE Software, pp 38-48
9. Karl Lieberherr, Doug Orleans, Johan ovlinger (2001), "**Aspect-Orient Programming with Adaptive Methods**"
10. Karl Lieberherr's (1997), "**Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP)**"
11. Karl Lieberherr (2001), "**Coupling Mechanisms in Aspect-Oriented Software**"
12. Karl Lieberherr (2001), "**Computer Science Adapts Techniques from Engineering and Mathematics**"
13. Karl Lieberherr (2004), "**Controlling the Complexity of Software Designs**"
14. Karl Lieberherr, Doug Orleans (2001), "**DJ: Dynamic Adaptive Programming in Java**"
15. Karl Lieberherr, Ian Holland (2002), "**Preventive Maintenance of Object-Oriented Software**"
16. Karl Lieberherr, Boaz Patt-Shamir, Doug Orleans (2004), "**Traversals of Object Structures: Specification and Efficient Implementation**"
17. wordIQ.com (2004), "**Definition of Tree traversal**"
18. www.soaprpc.com (2004), "**Web Service FAQ**"