

סיגלִיזַט חלק 5

נושא נפרד: שדות מחלקה ושרותי מחלקה

- השדות והשירותים שדברנו עליהם עד כה נקראים שדות מופע ושרותי מופע (instance fields/methods)
- העצמים הם מופעים של המחלקה
- לכל עצם עותק משלו של כל שדות המופע
- שרות מופע פועל על עצם מסוים (ש-`this` מתייחס אליו) ושדות המופע נגישים לו
- יש גם סוג אחר של שדות ושירותים, שמוגדרים עם המילה `static` לפניהם (כ-`modifier` בלי קשר אם הם `private` או `public` וכו'); הם נקראים שדות מחלקה ושרותי מחלקה

שדות מחלקה ושרותי מחלקה (המשך)

- לשדה מחלקה יש עותק אחד, ולא עותק לכל עצם.

- שימוש לדוגמא: מספר העצמים ממחלקה זאת שנוצרו

- שימוש נפוץ נוסף: קבועים גלובליים (`final`, השמה אחת)

```
public static final double PI = 3.14159
```

- שרותי מחלקה אינם פועלים על עצם מסוים, ולכן אינם יכולים לגשת לשדות מופע, אלא רק לשדות מחלקה

- שימושים נוספים: כאשר אין התייחסות לשדות בכלל, כגון פונקציות מתמטיות, וכמובן ה-`main`

- הגישה היא באמצעות שם המחלקה, לא שם העצם. דוגמא

```
Math.sin(...)
```

שדות מחלקה ושרותי מחלקה (דוגמה)

```
public class SomeClass {  
    private static int count = 0;  
    private SomeType x ..... // can be public etc.  
    public SomeClass ( ) {  
        count++; // additional code ....  
    }  
    public static int getCount ( ) {  
        return count; // cannot access x here!  
    }  
}
```

הוראות לניקוי אקווריום

How to clean your aquarium safely, by Sarah Davies

...

Turn off and remove all heaters and filters. These can be put in the sink and left until they are cleaned. Fill one of the new, clean buckets half full of water from the aquarium. Using the fish net, transfer **the fish**, one by one, to this bucket until all the fish are out of the aquarium. Next, remove all plants and ornaments. If the plants are living put them in the bucket with the fish. Put all **the ornaments** on the counter or in **the bucket** where the rocks go....

איזה דגים, קישוטים, ודליים בדיוק?

- כמובן שזה לא משנה; אותן ההוראות תקפות לאקוואריום עם דגי זהב ולאקוואריום עם ברקודות וכרישים, לדלי פח כחול ולדלי פלסטיק אדום
- ההוראות מתייחסות לעצמים באופן כללי שמאפשר שימוש בהוראות בהרבה מצבים שונים
- החוזה של הדלי: נוזלים ומוצקים שמכניסים לתוכו נשארים שם אם לא ממלאים אותו יותר מדי, מה שנכנס אפשר להוציא בדיוק באותו מצב (בפרט אסור שיהיו בדלי שיירי סבון)
- המימוש יכול להיות מפלסטיק, מפח, ואפילו מימושים כמו אגרטל חרסינה או סיר נירוסטה ממלאים את הדררשות

הקוד שלנו, עד עתה, לא היה כל כך כללי

• הלקוח שהשתמש בעצם מהמחלקה `VersionedString`

עשה זאת דרך ייחוס מטיפוס `VersionedString`

• עצם מהמחלקה הזו מקיים כמובן את החוזה שהלקוח

מסתמך עליו, אבל אולי יש עוד הרבה מחלקות שמקיימות

את החוזה הזה

• למה שקוד הלקוח לא יוכל לפעול על כל מחלקה כזו:

```
int Find(VersionedString vs, String s) {  
    for (int i=0; i<vs.length(); i++)  
        if (s.equals( vs.getVersion(i) ))  
            return i;  
}
```

זה פועל, אבל

- זה לא מאפשר להשתמש בשגרה הזו, המימוש הזה של אלגוריתם חיפוש, במצבים אחרים
- זה מקביל ל- "קח את דלי הפלסטיק האדום שמתחת לכיור (לא את דלי הספונג'ה הכחול), והעבר אליו את המים ואת דג הזהב"
- הפתרון: להפריד את הספק, המחלקה שמממשת את השירותים, מהחזוזה, שאינו תלוי במימוש

מִנְטָקִים (interfaces)

המונשק מגדיר את המנשקים של השירותים ואת החזרה

```
interface VersionedString {  
    public void add(String s) ;  
    requires ... ; ensures: ...  
    public int length() ;  
    requires ... ; ensures: ...  
    public String getLastVersion() ;  
    requires ... ; ensures: ...  
    public String getVersion(int i) ;  
    requires ... ; ensures: ...
```

מִנְשָׁקִים (המטרך)

- מנשק אינו מכיל מימוש כלשהו; השירותים מופיעים ללא גוף, רק כותרת שלאחריה נקודה פסיק (;)
- כל השירותים שמופיעים מיוצאים; למעשה ניתן להשמיט את ה public לפניהם (כתובת private למשל היא שגיאה)
- במנשק לא יכולים להופיע שדות (כי הם חלק ממימוש)
- במנשק לא יכולים להופיע שירותי מחלקה
- לא ניתן ליצור עצמים מן המנשק, ולכן לא יכולים להופיע בנאים

ספק מממש מנטר

ספק שמממש מנטר מקיים את החוזה שלו; אין לו חוזה משלו

```
class LinkedVersionedString
    implements VersionedString {
    protected int n;
    protected Version last;
    public void add(String s) {...}
    public int length() {...}
    public String getLastVersion() {...}
    public String getVersion(int i) {...}
}
```

הספק והחזרה

- הספק חייב לקיים את החזרה של המנשק שהוא מממש. ככלל, אין לו חזרה משלו
- הספק חייב לספק שירות אם המצב התוכנית מקיים את תנאי הקדם של השירות
- אם מצב התוכנית מקיים את תנאי הקדם של שירות, אזי מצבה לאחר סיום השירות חייב לקיים את תנאי האחר
- אולי השירות יפעל גם כשלא מקיימים את תנאי הקדם, ואולי השירות מביא למצב טוב מזה הנדרש בתנאי האחר
- אבל שירות טוב יותר מזה המובטח בחזרה אינו נדרש, וגם אם הספק מכריז על החזרה המשופר שלו, עדיף אולי ללקוח להימנע מלהסתמך עליו, כי אחרת לא יוכל להחליף ספק

דוגמה: משחקים

```
public interface GameMove {  
}  
  
public interface GameBoard {  
    public GameMove move(GameMove m,  
                           boolean isWhite) ;  
  
    public GameMove[] legalMoves() ;  
    public double isGameOver() ;  
    public double score() ;  
}
```

כעת ניתן להגדיר חיפוש כללי בעץ משחק

```
public interface GameSearch {  
    public GameMove optimalMove(GameBoard b,  
        boolean isWhite);  
}
```

- המעלה שבשימוש במנשקים עבור הלוח והמהלך הוא שכעת אנו יכולים להגדיר אלגוריתם חיפוש בעץ משחק באופן כללי, ולא עבור משחק מסוים
- אותו אלגוריתם יעבוד עבור Connect Four, עבור שחמט, עבור דמקה, גו, וכן הלאה; כל מה שצריך עבור כל אחד מהם הוא מימוש מתאים של GameBoard ושל GameMove

ג'אווה לא מהפשת דליים מתחת לכיור

- אני כן, ואם הייתי מנסה לעקוב אחרי הוראות ניקוי האקווריום, והייתי קורא שצריך דלי נקי, הייתי מחפש ומוצא אבל ג'אווה לא, ולכן, הקוד הבא יכשל ולא יעבור קומפילציה,

```
GameMove m = new GameMove(); // won't compile
```

- ג'אווה רוצה שהלקוח יגיד איזה סוג עצם (מאיזו מחלקה) הוא רוצה לבנות; ג'אווה לא תנחש עבורו, אפילו אם רק מחלקה אחת מממשת את טיפוס המנשק של המשתנה, או רק מחלקה אחת שמממשת את המנשק ויש לה בנאי מתאים

- מה שצריך בג'אווה הוא

```
GameMove m = new ConenctFourMove();
```

אסימטריה

- שירות כמו `optimalMove` ב-`GameSearch` שהגדרנו יכול להשתמש רק בטיפוס המנשק, ולכן לעבוד עם כל מימוש שיועבר לה כארגומנט
- אבל קוד שצריך ליצור עצמים לא יכול להשתמש בהם רק דרך טיפוס המנשק המתאימים, מכיוון שלאופרטור `new` חייבים להעביר שם של מחלקה, לא של מנשק

דוגמא לשימוש במנטק

```
public void playInteractive() {
    GameBoard b = new ConnectFourBoard();
    GameMove m;
    ...
    do {
        ...
        m = new ConnectFourMove(column);
        b.move(m, true);
    } while (b.isGameOver == -1.0);
}
```

אם רוצים להשתמש במימוש אחר,

- יש להחליף את

```
GameBoard b = new ConnectFourBoard();
```

- ב

```
GameBoard b = new ChessBoard();
```

- ובאופן דומה להחליף את הקריאה לבנאי של
ConnectFourMove
- יתר הקוד בלתי תלוי במימוש שנבחר
- במקרה של GameMove הבנאים כנראה יהיו שונים
בפרמטרים שלהם
- בכל מקרה, המנשק אינו קובע כיצד יראו הבנאים

הפותרון: בתי חרושת (factories)

עצם שמממש מושק יצירת עצמים מטיפוס מושק מסוים

```
interface GameMoveFactory {  
    public GameMove construct(String ms);  
}  
  
class ConnectFourMoveFactory  
    implements GameMoveFactory {  
    public GameMove construct(String ms) {  
        int column = Integer.parseInt(ms);  
        return new ConnectFourMove(column);  
    }  
}
```

שימוש בבית הרושת (1)

```
public void playInteractive(
    GameBoardFactory f, ...) {
    ...
    do {
        ...
        GameMove m = f.construct(ms);
        b.move(m, true);
    } while (b.isGameOver == -1.0);
}
```

• צריך גם בית הרושת עבור `GameBoard`

שימוש בבית חרושת (2)

- בזמן יצירת העצם ששירותיו צריכים GameMove חדשים, מעבירים לבנאי שלו בית חרושת; אפשר להעביר לו בית חרושת שייצר ConnectFourMove או בית חרושת שייצר עצמים אחרים שמקיימים את המנסק
- ניתן כמובן גם להעביר את בית החרושת ישירות לשירות שהקוד שלו צריך עצמים חדשים
- בקוד של המחלקה הלקוחה אין אזכור לספקים ספציפיים, לכן היא יכולה לעבוד עם כל ספק, גם ספק שייכתב בעתיד
- המבנה הרצוי תלוי בעיקר בשאלה מי הקוד שמחליט באיזה ספק להשתמש, ולכן באיזה בית חרושת להשתמש; בדרך כלל, זהו קוד בקרת תצורה שרחוק למדי מהקוד המשתמש

בתי הרושת משוכללים

בית הרושת יכול לייצר עצמים תוך שימוש בארגומנטים

```
interface VersionedStringFactory {
```

```
    public VersionedString construct();
```

Ensures: returns a reference to a new VersionedString

```
    public VersionedString construct(int m);
```

Ensures: returns a reference to a new VersionedString

that keeps at least the m most recent versions

```
}
```

אם החזרה מרשה, מותר להתבטל

```
class LinkedVersionedStringFactory
    implements VersionedStringFactory {
    public VersionedString construct() {
        return new LinkedVersionString();
    }
}
```

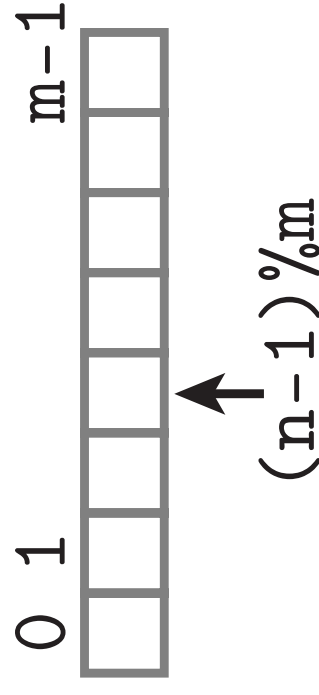
keeps an unlimited number, so satisfies the contract

```
public VersionedString construct(int m) {
    return new LinkedVersionString();
}
```

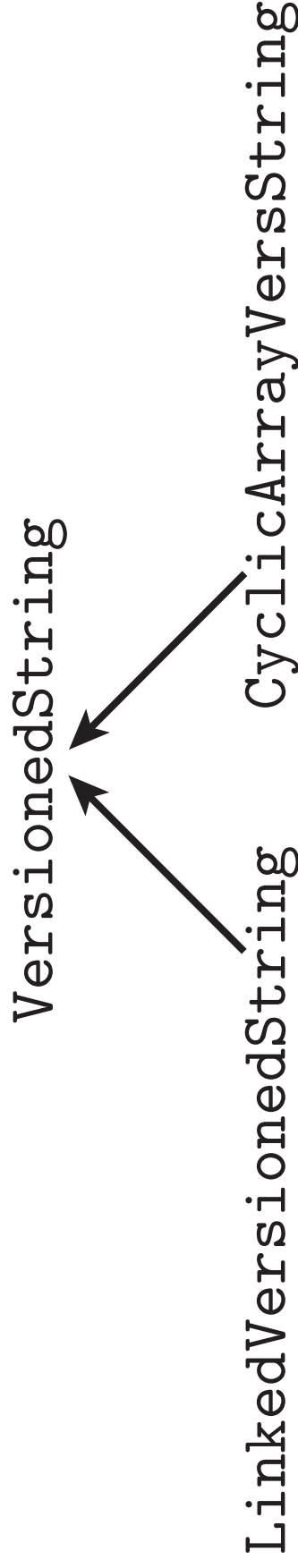
מאפשר ללקוח לתת עצה לבית החרושת;
בית החרושת לא חייב לנצל את העצה

אבל אפשר להתאמץ

```
class SmartVersionedStringFactory
    implements VersionedStringFactory {
    public VersionedString construct() {
        return new LinkedVersionString();
    }
    public VersionedString construct(int m) {
        return new CyclicArrayVersionString(m);
    }
}
```



ניצני ארגון עבוד טיפוסים



- בין שלושת הטיפוסים יש יחסים: שתי המחלקות ממשות את המנשק
- המנשק **יותר כללי** מהמחלקות שממשות אותו
- הכלל הבסיסי: משתנה או שדה מטיפוס כללי יותר (כאן מנשק) יכול להתייחס לעצם מטיפוס יותר ספציפי
- אבל לא להיפך

עוד דוגמה: מיון מערך

requires: a != null && a's elements != null

ensures: a becomes sorted

```
void insertionSort(Comparable[] a) {  
    int i, j;  
    for (j=1; j<a.length; j++) {  
        Comparable key = a[j];  
        for (i=j-1;  
             i>=0 && a[i].compareTo(a[j])>0;  
             i--) a[i+1] = a[i];  
        a[i+1] = key;  
    }  
}
```

מנשק להשוואות

```
interface Comparable {  
    requires: other != null  
    ensures: return == 0 iff this = other  
        return == 1 iff this > other  
        return == -1 iff this < other  
    int compareTo(Comparable other);  
}
```

בספריות של השפה מוגדר מנשק `Comparable` מ `java.lang`. הוא דומה ברוחו ובמהותו למנשק שהגדרנו כאן, אבל לא זהה לו לגמרי; בהמשך נבין למה

מימוש המנטק ב-`VersionedString`

```
class LinkedVersionedString
    implements VersionedString,
        Comparable {
    protected int    n;
    protected Version last;
    int compareTo(Comparable other) {
        if (this.n > other.n) return 1;
        if (this.n < other.n) return -1;
        return 0;
    }
}
```

לפעמים אני מרצה ולפעמים הורה

- וזה נורמלי

- אני מספק שירותים מסוימים בתור מרצה ואחרים בתור הורה

- אבל אני תמיד נשאר אני (וממש שני מנשקים שונים) יש מכונת פקס שלפעמים היא פקס, לפעמים מכונת צילום, לפעמים טלפון, ולפעמים משיבון

- הגדרה של מחלקה יכולה להצהיר שהיא ממשתת מספר מנשקים

אבל המימוש לא נכון!

```
int compareTo(Comparable other) {  
    if (this.n > other.n) return 1;  
    if (this.n < other.n) return -1;  
    return 0;  
}
```

- השירות למעשה מניח ש-`other` הוא מטיפוס `LinkedVersionString`
- אבל `other` הוא מטיפוס `Comparable`, לא מטיפוס `LinkedVersionString`, ולכן הקומפילר לא ירשה להתייחס לשדה `n` דרכו

ניסיון לפתרון הבעיה

נדרוש בתנאי הקדם של `insertionSort` שכל העצמים במערך יהיו מאותה מחלקה, ונתייחס ל-`other` בהתאם:

```
int compareTo(Comparable other) {  
    LinkedVersiondString other_lvs  
        = (LinkedVersiondString) other;  
    if (this.n > other_lvs.n) return 1;  
    if (this.n < other_lvs.n) return -1;  
    return 0;  
}
```

האופרטור שבסוגריים נקרא יציקה (`cast`), והוא מייצר ייחוס מטיפוס נתון לעצם נתון, אם העצם מתאים לטיפוס

המרת טיפוס ייחוס

- עצמים יכולים להיות מומרים בין טיפוס ייחוס שונים.
- המרה מרחיבה (widening) מטיפוס ספציפי לטיפוס כללי יותר, תמיד מותרת באופן אוטומטי, למשל בהשמה

```
LinkedVersionedString lvs;
```

```
VersionedString vs;
```

```
lvs = vs; // vs is widened
```

- המרה מצרה (narrowing) מטיפוס כללי לטיפוס ספציפי יותר, דורשת פעולה מפורשת של יציקה (cast), ולא תמיד מצליחה (עלולה להיכשל בזמן ריצה אם העצם שמתייחסים אליו אינו מהטיפוס שדרשנו ב-cast)

יציקות (casts)

- תחביר:

(`<TypeToBeConvertedTo>` `<Expression>`)

- הביטוי `<Expression>` מחושב, ונעשה ניסיון להמיר את ערכו לטיפוס `<TypeToBeConvertedTo>`

- כאשר הביטוי הוא ייחוס, היציקה מצליחה אם הייחוס מתייחס לעצם מתאים לטיפוס שיוצקים לתוכו

- יציקה למטה (`downcast`): יציקה של ייחוס לטיפוס פחות כללי; כרגע הכוונה ליציקה של ייחוס למנשק לייחוס למחלקה שמממשת את המנשק; בהמשך נראה שיש עוד מקרים

יציקות (casts) - המשך

- יציקה למעלה (upcast): יציקה של ייחוס לטיפוס יותר כללי, למשל יציקה של ייחוס למחלקה לייחוס למנשק שהמחלקה מממשת; שוב, בהמשך נראה עוד מקרים
- יציקה למעלה תמיד מצליחה, (כאמור לא נדרש אופרטור יציקה מפורש); היא פשוט גורמת לקומפילר לאבד מידע
- יציקה למטה עלולה להיכשל; בהמשך נראה מה קורה אז
- אפשר לצקת גם ערכים פרימיטיביים. לדוגמא, ערך הביטוי הוא מספר בנקודה צפה והוא מומר לשלם,

```
double x = 4.0, y = 5.5;
```

```
int i = (int) (x + 2.5 * y);
```

אבל היציקה לא פתרה את הבעיה

- כי הלקוח לא בהכרח יודע אם כל העצמים שהוא רוצה למיין שייכים לאותה מחלקה או לא
- למשל, אם `SmartVersionedStringFactory` ייצר את העצמים, יתכן שהלקוח קיבל עצמים מכמה מחלקות
- אם בעיני הלקוח אפשר למיין עצמים שמממשים `VersionedString`, אז סימן שהמיון צריך לבטא תכונות של העצמים שהלקוח מודע להם, לא את המימוש הנסתר
- אי לכך, צריך להצהיר שכל עצם שמממש `Comparable VersionedString`

מנשק מבטיח לממש מנשק אחר

```
interface VersionedString
    extends Comparable {
    public void add(String s) ;
    public int length() ;
    public String getLastVersion() ;
    public String getVersion(int i) ;
}
class LinkedVersionedString
    implements VersionedString { ... }
```

- המנשק לא צריך להצהיר על `compareTo`, קיום השירות `Comparable` מעצם העובדה שהמנשק מרחיב את `Comparable`

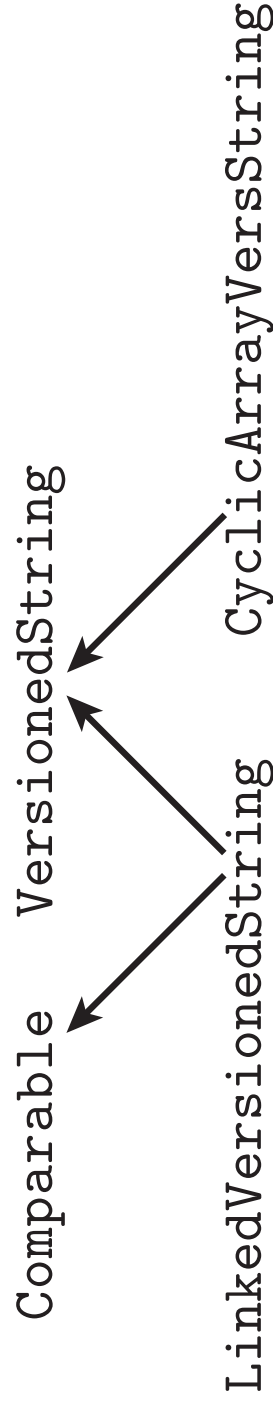
והמימוש הנכון...

```
int compareTo(Comparable other) {  
    VersiondString other_vs  
        = (VersiondString) other;  
    if (this.length() > other_vs.length())  
        return 1;  
    if (this.length() < other_vs.length())  
        return -1;  
    return 0;  
}
```

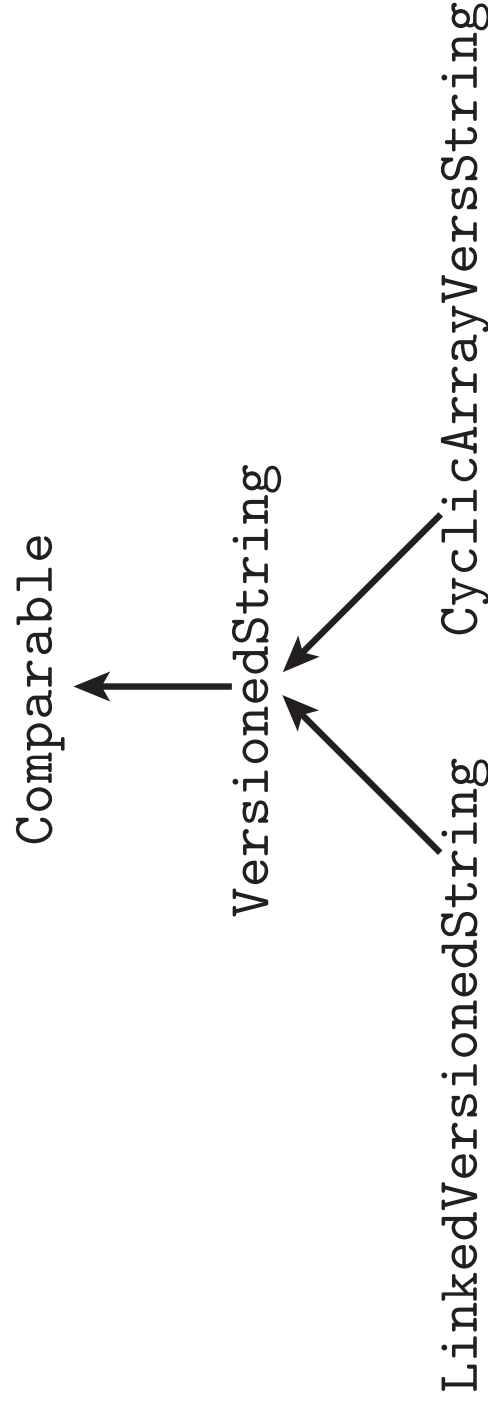
קצת פגום: למרות שהמימוש לא ספציפי למחלקה צריך לשכפול בכל מימוש של `VersiondString`; בהמשך נתקן

היררכיית הטיפוסים גדלה

זה לא היה
מבנה מוצלח
במקרה זה,
אבל הוא
הראה



שהיחסים בין טיפוסים הם גרף א-ציקלי מכוון
המבנה הזה היה יותר מוצלח, והוא מראה שגרף היחסים עשוי
להיות עמוק



סיכום מנשקים

- שימוש בטיפוס מנשק מאפשר ללקוח להצהיר שייחוס (משתנה או שדה) מתייחס לעצם שמספק שירותים מסוימים, בלי לציין מאיזו מחלקה העצם; זה מאפשר כלליות בלקוח
- לקוח כזה נקרא רב-צורת (polymorphic)
- שימוש בבית חרושת (factory) מאפשר ללקוח לבנות עצמים עם מנשק מסוים בלי לציין בעצמו מאיזו מחלקה הם מחלקה יכולה לממש מספר מנשקים
- מנשק יכול להרחיב מנשק אחר או מספר מנשקים אחרים
- **רב צורתיות מאפשרת ניצול של קוד קיים במקרים נוספים ומונעת שכפול של קוד, שכפול שהוא יקר לפיתוח ותחזוקה**