

תכנות מונחה עצמים בשפת C++

אוהד ברזילי
אוניברסיטת תל אביב

על טיפוסים וירוושה (לא בהכרח ב C++)

ההרצאה מבוססת על מצגת של פרופ' עמירם יהודאי ע"פ הספר:

Object-Oriented Software Construction,
second edition, by Bertrand Meyer (Prentice Hall) .

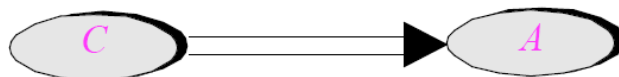
כל הזכויות שמורות למחברים

ירושה וטענות (assertions)

- תנאי קדם, תנאי בתר ושמורות שהוגדרו עבור מחלקה תקפים גם לגבי צאצאיה, ועשויים להשתנות
- עצם ממחלקה נגזרת המוצבע ע"י עצם (מצביע או הפנייה) מטיפוס מחלקת הבסיס צריך לקיים את שמורת מחלקה הבסיס
- מכאן ששמורה של כל מחלקה יכולה להיות שווה או חזקה יותר משמורת הוריה

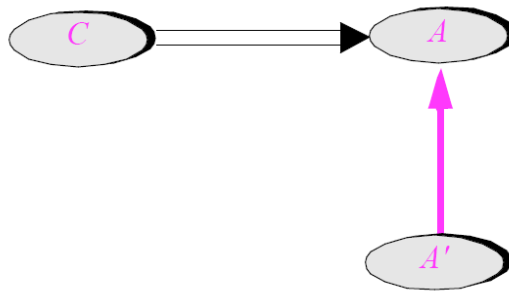
קבלנות משנה

- מחלקת C היא לקוחה של מחלקה A, כלומר:
 - יש ל-C מצביע A (אחד השדות)
 - או
 - אחת המתודות של C מקבלת פרמטר מטיפוס מצביע ל-A
- C מכירה את השמורה של A ומצפה מ A לקיים אותה



קבלנות משנה - השמורה

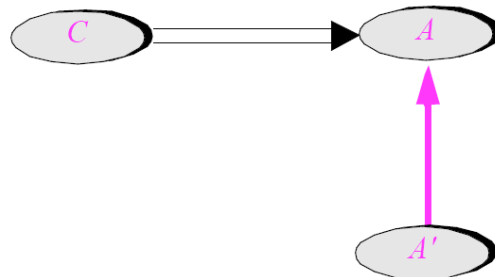
- בפועל, המצביע ל- A מצביע ל- A' , מחלקה הנורשת מ- A
- ברור שכדי לקיים יחסים פולימורפים תקינים על A' לקיים לפחות את שמורת A



5

קבלנות משנה – תנאי קדם ובתר

- המחלקה A' מסתירה (overrides) רוטינה של A
- מה יש לדרוש מתנאי הקדם והבתר של המתודה החדשה ביחס לאלו של הרוטינה המקורית?



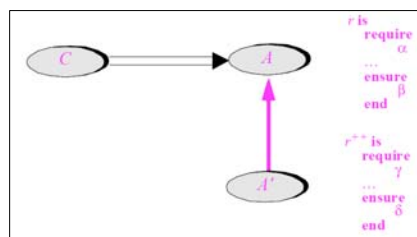
r is
require
 α
...
ensure
 β
end

r^{++} is
require
 γ
...
ensure
 δ
end

6

קבלנות משנה – תנאי קדם

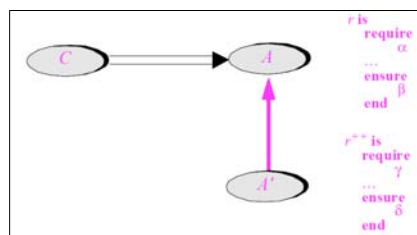
- נתבונן בקריאה $a.r()$ המופיעה במחלקה C
- על C לקיים את תנאי הקדם של $A: r$, היא כלל אינה מכירה את המחלקה A' ואינה יודעת על קיום $A': r$
- לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות שווה או חלש יותר מתנאי הקדם המקורי



7

קבלנות משנה – תנאי בתר

- משיקולים דומים על תנאי הבתר של המחלקה הנגזרת להיות שווה או חזק יותר מתנאי הבתר המקורי
- ללקוח C 'הובטח' β ע"י A ואסור שמאחורי הקלעים יסופק δ החלש ממנו
- מנגנון זה מכונה "קבלנות משנה" (subcontracting)



8

השמורה האפקטיבית

- השמורה ה'אמיתית' של מחלקה C מורכבת מ AND לוגי של כל הטענות המופיעות בשמורה של מחלקה C ובכל הוריה לאורך עץ הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדרה שמורה, ניתן להתייחס לשמורה שלה כ- TRUE
- כותב המחלקה C יכול להגדיר את שמורת C בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד

תנאי קדם אפקטיבי

- תנאי הקדם ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה C הוא ה OR הלוגי של כל תנאי הקדם של מתודה זו בכל הוריה של C לאורך עץ הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדר תנאי קדם למתודה זו, ניתן להתייחס לתנאי הקדם שם כ- FALSE
- כותב תנאי הקדם של המתודה שהוגדרה מחדש במחלקה C יכול להגדיר אותו בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד

תנאי בתר אפקטיבי

- תנאי הבתר ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה C הוא ה AND הלוגי של כל תנאי הבתר של מתודה זו בכל הוריה של C לאורך עץ הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדר תנאי קדם למתודה זו, ניתן להתייחס לתנאי הקדם שם כ- TRUE
- כותב תנאי הבתר של המתודה שהוגדרה מחדש במחלקה C יכול להגדיר אותו בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד

דוגמא

```
class MATRIX {  
...  
    /** inverse of current with precision epsilon  
    * @pre: epsilon >= 10 ^(-6)  
    * @post: (this * $prev(this) - ONE).norm <= epsilon  
    */  
    void invert(double epsilon);  
...  
};
```

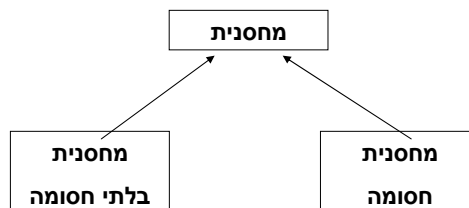
דוגמא

```
class ACCURATE_MATRIX : public MATRIX {
...
  /** inverse of current with precision epsilon
   * @pre: epsilon >= 10^(-20)
   * @post: (this * $prev(this) - ONE).norm <= epsilon/2
   */
  void invert(double epsilon);
...
};
```

- בשפת Eiffel כדי להדגיש שהחוזר של מתודה שהוגדרה מחדש אינו עומד בפני עצמו אלא תלוי בהיררכיה החליפו את התגיות require else- ו- ensure then בהתאמה

תנאי קדם מופשט

- מהי ההיררכיה בין 3 המחלקות: מחסנית, מחסנית חסומה, מחסנית בלתי חסומה?



- מה יהיה תנאי הקדם של המתודה של put במחלקה מחסנית?

תנאי קדם מופשט

- תנאי הקדם לא יכול להיות ריק (TRUE) כי אז הוא יחזק ע"י המחסנית החסומה
- תנאי הקדם צריך להיות `!full()` כאשר `full()` היא מתודה (וירטואלית) המחזירה תמיד `false`, שתוגדר מחדש במחלקה מחסנית חסומה להחזיר `count() == capacity()`
- תנאי קדם המכיל מתודות וירטואליות נקרא תנאי קדם מופשט
- למרות שתנאי הקדם הקונקרטי אכן מתחזק ע"י המחסנית החסומה תנאי הקדם המופשט נשאר ללא שינוי

תכונות שאינן קיימות ב C++

- בשפות התכנות Java ו-Eiffel כל המחלקות יורשות ממחלקת אב ראשית (GENERAL או Object) המכילה שרותי מחלקה בסיסיים (כגון: `clone`, `copy`, `equal` ואחרים)
- בשפות התכנות Java ו-Eiffel ניתן למנוע `overriding` של מתודות (ע"י הכרזתן כ `final` או `frozen`) במחלקות נגזרות
- הגרסה ה-C++-ית היא הגדרת מתודה ללא המציין `virtual` המשיגה חלק מתכונות ה `frozen`

אילוצים על טיפוסים התבנית

- בשפות Java ו- Eiffel ניתן לציין בהגדרת התבנית כי הטיפוס הפרמטרי בה שייך להיררכיה כלשהי
 - בשפת Eiffel: `class A [T -> B]` - אומר כי הטיפוס T חייב לרשת מטיפוס B
 - בשפת Java: `List<? extends Shape>` - אומר כי הטיפוס ? חייב להיות צאצא של Shape
- הדבר מאפשר להשתמש בתכונות (מתודות ושדות) של אותו טיפוס הגוף התבנית
- בשפת C++ לא צריך לציין זאת במפורש – יצירת מופעים של תבניות עם טיפוסים אשר אינם מכילים תכונות נדרשות גוררת טעות קומפילציה

אילוצים על טיפוסים התבנית

- ואולם, הבעיה האמיתית טמונה בשילוב של תבניות עם טיפוסים פולימורפים
- נניח כי המחלקה Derived יורשת מהמחלקה Base, וכי A היא תבנית המצפה לטיפוס T
- אזי יש הבדל מהותי בין שני קטעי הקוד הבאים:

```
void g(Base *a);  
int main()  
{  
    Derived d;  
    g(&d); // polymorphism  
}
```

```
void g(A<Base> *a);  
int main()  
{  
    A<Derived> a;  
    g(&a); // Error  
}
```

תבניות וירושה

אם D הוא יורש מ-B, ו-A היא תבנית כלשהי אזי:

$A < B >$ אינו תת טיפוס (יורש מ-) של $A < D >$

ניסיון השמה

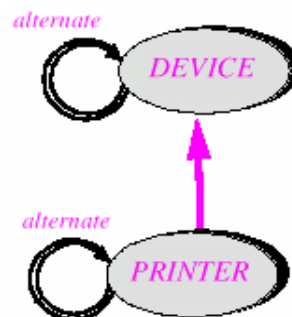
- הצגנו בתרגול את האופרטורים `dynamic_cast` ו-`typeid` התומכים הטיפוסי זמן ריצה
- המקבילות להן בשפות תכנות אחרות:
 - Eiffel: `x?=y` – ניסיון השמה
 - C#: `is` ו-`typeof`
 - Java: `instanceof` האופרטור

הגדרה מחדש של טיפוסים

- שדה, טיפוס מוחזר של מתודה וטיפוסי המשתנים של מתודות עשויים להיות מוגדרים מחדש ע"י טיפוס חדש במחלקות נגזרת (הערה: זו אינה הסתרה אלא העמסה)
- על הטיפוס החדש להיות מתאים לטיפוס הישן – תכונה זו מכונה "כלל השונות המשותפת" במדיניות טיפוסים (covariant typing policy)
- התאמה לטיפוס הישן עשויה להתממש ב-3 אופנים:
 - Covariance – על הטיפוס החדש לרשת מהישן
 - Contravariance – על הטיפוס הישן לרשת מהחדש
 - Invariance – על הטיפוסים להיות זהים ממש

covariance

```
class DEVICE {  
    DEVICE *alternate;  
public:  
    /** Designate an alternate */  
    virtual void set_alternate (DEVICE *a)  
    {  
        alternate = a;  
    }  
};  
  
class PRINTER : public DEVICE {  
    PRINTER *alternate;  
public:  
    /** Designate an alternate */  
    virtual void set_alternate (PRINTER *a)  
    {  
        alternate = a;  
    }  
};
```



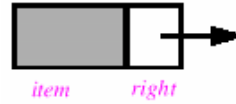
דוגמא נוספת

```

template<class G>
class LINKABLE {
    G *item;
    LINKABLE<G> *right;

public:
    /** put other to the right of current cell */
    void put_right(LINKABLE<G> *other)
    {
        right = other;
    }
    //...
};

```



תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

23

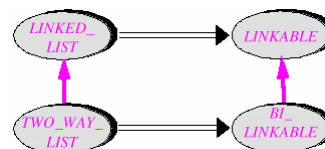
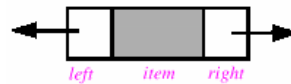
דוגמא נוספת - המשך

```

/**
 * @inv: right == NULL || right->left == this
 * @inv: left == NULL || left->right == this
 */
template<class G>
class BI_LINKABLE : public LINKABLE<G>
{
    BI_LINKABLE<G> *left, *right;

public:
    /** put other to the right of current cell */
    void put_right(BI_LINKABLE<G> *other)
    {
        right = other;
        if (other != NULL)
            other->put_left(this);
    }
    void put_left(BI_LINKABLE<G> *other)
    { /* ... */ }
};

```



תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

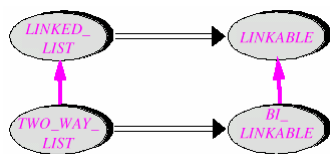
בעיית העוגן

```
template<class G>
class LINKED_LIST
{
    G *item;
    LINKABLE<G> *first_element;

public:
    void put_right (G *v)
    {
        LINKABLE<G> *new_node;
        new_node = new LINKABLE<G>(v);
        // ...
    }
};
```

בעיית העוגן

- המחלקה BI_LINKED_LIST יורשת מהמחלקה LINKED_LIST ומשתמשת במתודה put_right
- ואולם המשתנה המקומי new_node הוא מהטיפוס הלא נכון!
- כנ"ל לגבי רוב שאר התכונות של LINKED_LIST שהוגדרו במונחי LINKABLE
- יש להגדיר מחדש את כל הפעולות הבעיות כדי להשיג התנהגות נכונה



פתרון בעיית העוגן ב Eiffel

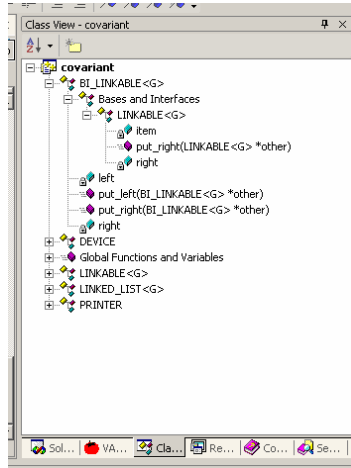
- בשפת Eiffel פתרו את הבעיה ע"י הוספת מנגנון like
- למשתנים המופעים במחלקה יוגדר במקום טיפוס שם של שדה במחלקה ("העוגן") שהוא מעוניין להצמיד את טיפוסו אליו
- המחלקה היורשת תצטרך עתה לעדכן רק את טיפוסו של העוגן וכל שאר הפעולות יעודכנו בהתאם
- ב Eiffel זהו מנגנון סטטי – זמן קומפילציה
- איך נממש מנגנון דומה ב C++ ?

שימוש ב like (לא ב C++)

```
template<class G>
class LINKED_LIST
{
    G *item;
    LINKABLE<G> *first_element;

public:
    void put_right (like item *v)
    {
        like first_element *new_node;
        new_node = new LINKABLE<G>(v);
        // ...
    }
};
```

שיטוח המחלקה



תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

29

- ברוב סביבות הפיתוח ניתן לייצר עבור מחלקה "שיטוח" שלה (flat view) – כלומר להציג את כל תכונות המחלקה הממשיות (concrete) בין אם הן הוגדרו במחלקה עצמה, נורשו ממחלקת בסיס כלשהי או הוגדרו מחדש
- לרעיון יש תמיכה בקוד, תיעוד והצגת מתאר

מתי לרשת?

- המחלקה CAR_OWNER עשויה לרשת מ PERSON אבל עדיף שתהיה לקוחה של CAR
- יחס is-a לעומת יחס is-part-of
- פרט למנגנון הרב-צורתיות (polymorphism) ירושה לעולם אינה הכרחית
- במקום ש B יירש מ-A, ל-B יכולה להיות התכונה A (שדה מוכל או מצביע)
- To be is also to have אבל לא להיפך (משאית היא מכונית)
- לפעמים נוח לשאול "האם יכולים להיות לו שניים?"
 - לדוגמא: למכונית יש מנוע

תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

30

שימוש חוזר בממשק ובמימוש

יחס ירושה	יחס שימוש (client-supplier)
שימוש חוזר דרך מימוש	שימוש חוזר דרך ממשק
הסתרת מידע מוגבלת	יש הסתרת מידע
הגנה מוגבלת מפני שינויים	הגנה מפני שינויים במימוש המקורי