על טיפוסים וירושה
תרגול

אוהד ברזילי

אוניברסיטת תל אביב

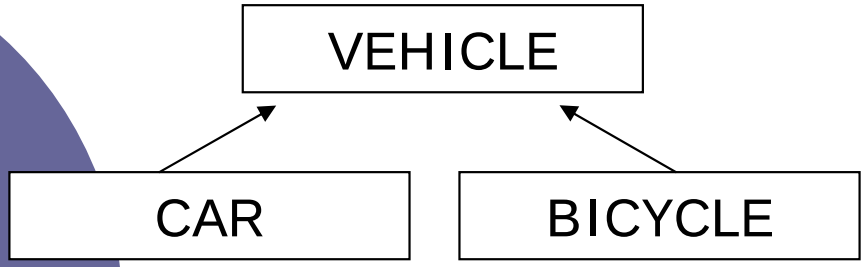
תנאי קדם מופשט

תנאי קדם מופשט

- ראינו שבקבלנות משנה החוזה של המחלקה היורשת חייב לקיים 3 עקרונות:
  - תנאי קדם יכולים להיות חלשים
  - תנאי בתר יכולים להיות חזקים יותר
  - שמורת המחלקה יכולה להיות חזקה יותר
- כאשר מחלקת הבסיס מופשטת, תנאי קדם טריויאלים מחייבים לפעמים ראייה לעתיד, כדי שלא יחוזקו במחלקות נגזרת (בהרצאה ראינו דוגמא של מחסנית מופשטת ומחסנית חסומה)

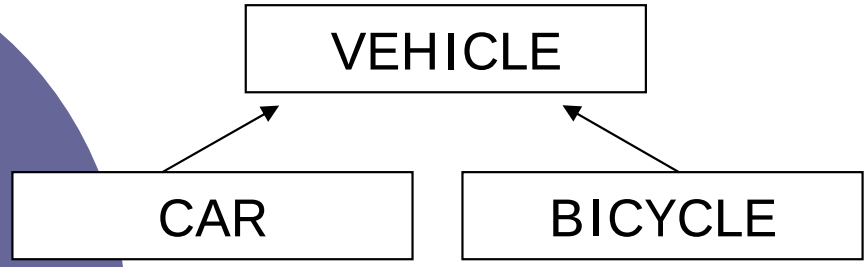
ראייה לטווח רחוק

- ראייה לעתיד אינה דבר מופרך במחלקות מופשטות
- האבולוציה של היררכית מחלקות כלי הרכב לא התחילה בגזירת מחלקות קונקרטיות שיירשו מ VEHICLE
- הגיוני יותר שבמהלך מימוש \או עיצוב המחלקות CAR ו- AIRPLANE נגלה שיש להן הרבה מן המשותף, וכדי למנוע שכפול קוד ניצור מחלקה שלישית - VEHICLE שתכיל את החיתוך של שתיהן
- אף כלי רכב אינו רק VEHICLE
- בראייה זו, לא מוגזם לדרוש ממחלקה מופשטת ניסוח תנאי קדם מופשט



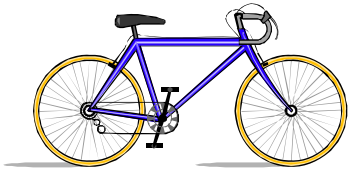
דוגמא

- מהו תנאי הקדם של המתודה go() של המחלקה VEHICLE ?
- על פניו – אין כל תנאי קדם לפעולה מופשטת
- מה עם המחלקה CAR ? – לה בטח יש דרישות כגון hasFuel()
- מה עם המחלקה BICYCLE ? – לה בטח יש דרישות כגון hasAir()
- איך VEHICLE תגדיר תנאי קדם ל go() גם כללי מספיק וגם שלא יחוזק ע"י אף אחד מיורשותיה?



פתרון

- מתודה בולאנית וירטואלית כגון canGo() תעשה את העבודה
- המתודה תוגדר כמחזירה TRUE עבור VEHICLE (או שתוגדר כטהורה), ועבור כל אחת מיורשותיה תוגדר לפי המחלקה האמורה
- בעצם המתודה go() היתה צריכה להיקרא "go_if_you_can()" וכך לא היתה כל הפתעה בתנאי הקדם "המוזר"



מניעת ירושה

מימוש הרעיון ב C++

מניעת ירושה

- מניעת ירושה משיקולי יעילות בדרך כלל אינה במקומה:
  - המנגנון הוירטואלי יוצר תקורה קבועה, זניחה וללא תלות בעומק הירושה
  - המנגנון הוירטואלי פועל רק במקרה של מצביעים, את שאר המקרים ממיר ה optimizer לקריאות רגילות
  - קיטום ההררכיה משיקולי מניעת פונקציות וירטואליות צריך לברר האם הפונקציות צרכיות להיות וירטואליות מלכתחילה

דוגמא

- נדגים איך ניתן למנוע ממחלקות אחרות לרשת מהמחלקה Usable

```
class Usable;

class Usable_lock
{
   friend class Usable;
private:
   Usable_lock() {}
   Usable_lock(const  Usable_lock&) {}
};
```

דוגמא

```
class Usable : public virtual Usable_lock
{
   // ...
public:
   Usable();
   Usable(char*);
   // ...
};

Usable a;

class DD : public Usable { };

DD dd;  // error: DD::DD() cannot access
        // Usable_lock::Usable_lock(): private  member
```

אילוצים על טיפוסי התבנית

מימוש הרעיון ב C++

אילוצים על טיפוסים התבנית

- נרצה לממש מנגנון כפי שקיים אשר אוכף הגבלות על הטיפוס הפורמלי של התבנית
- כבר עכשיו נקבל טעיות קומפילציה עבור טיפוסים לא חוקיים (שלא מכילים מתודה מסויימת) ואולם נרצה שההערות הקומפיילר יהיו אינפורמטיביות יותר
- למשל:

```
template<class Container>
void draw_all(Container& c)
{
   for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

קרוב ראשון

```
template<class Container>
void draw_all(Container& c)
{
    Shape* p = c.front();   // accept only containers
                            // of Shape*s

    for_each(c.begin(),c.end(),mem_fun(&Shape::draw));

}
```

- טעות הקומפילציה, תפנה להשמה ולא לתבנית mem_fun

הכללה

- נכליל את הרעיון:

```
template<class Container>
void draw_all(Container& c)
{
   typedef typename Container::value_type T;
   Can_copy<T,Shape*>(); // accept containers of
                         // only Shape*s
   for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

- נבטא את הטענה מפורשות בקוד (ללא ביצוע ההשמה בפועל):

```
template<class T1, class T2>
struct Can_copy {
   static void constraints(T1 a, T2 b)
   { T2 c = a; b = a; }
   Can_copy() { void(*p)(T1,T2) = constraints; }
};
```

עוד אילוצים על הטיפוס

```
template<class T, class B> struct Derived_from {
   static void constraints(T* p) { B* pb = p; }
   Derived_from() { void(*p)(T*) = constraints; }
};

template<class T1, class T2> struct Can_copy {
   static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
   Can_copy() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2 = T1> struct Can_compare {
   static void constraints(T1 a, T2 b) { a==b; a!=b; a<b; }
   Can_compare() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2, class T3 = T1> struct Can_multiply {
   static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
   Can_multiply() { void(*p)(T1,T2,T3) = constraints; }
};
```

דוגמא

- העצמים במיכל צריכים לרשת מ MyBase*:

```
template<class T>
class Container : Derived_from<T,Mybase>
{ // ... };
```

- Derived_from לא באמת בודק ירושה כי אם השמה פולימורפית, אבל במקרה זה. זו כוונתנו

מקרים נוגדי פולימורפיזם

Covariance והסתרת תכונות ב- C++

הסתרת תכונות

- במחלקה POLYGON קיימת התכונה addVertex() אשר מוסיפה קודקוד למצולע קיים
- במחלקה RECTANGLE היורשת ממנה אין משמעות לוגית לכזו מתודה
- ניתן להסתיר אותה בתהליך הירושה ע"י הצהרתה מחדש כ private ואולם הדבר סותר את עקרון הפולימורפיזם

מלבן ומצולע

```
class POLYGON {
  //...
public:
  void addVertex(POINT &p) { /*...*/ }
  //...
};

class RECTANGLE : public POLYGON {
  POLYGON::addVertex;
  //...
};
```

שבירת הפולימורפיזם

```
void f(POLYGON *pp, POINT& x)
{
   RECTANGLE r;
   POLYGON p;

   p.addVertex(x);
   pp->addVertex(x); // The problem started already here !
   r.addVertex(x); // ERROR! hidden in RECTANGLE

   pp = &r;
   pp->addVertex(x); // TROUBLE! Adding a vertex
                     // to a RECTANGLE
}
```

הסתרת תכונות ופולימורפיזם

- קיימות מספר גישות ליציאה מהסבך – כולן כרוכות ב"פשרות כואבות":
  - להחליט שמלבן אינו תת מחלקה של מצולע אלא מצב כלשהו של מצולע (עצם ולא מחלקה)
  - להחליט שהתכונה addVertex() אינה צריכה להיות במצולע, מכיוון שמצולע הוא בסיס משותף של כל נגזרותיו
  - להחליט שמלבן אינו יורש ממצולע – למשל שניהם יכולים להיות אחים היורשים ממחלקה שלישית

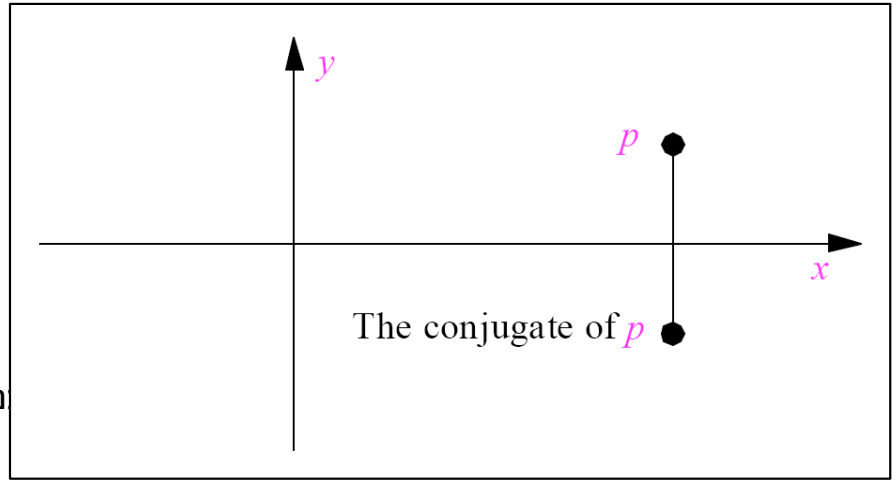
שונות משותפת

- ראינו בהרצאה כי כדי לשמור על בטיחות טיפוסים (ובמקרים מסוימים כדי לשמור על נכונות התוכנית) יש להגדיר מחדש את הטיפוס של שדות ומתודות של מחלקת הבסיס
- על אף שככול שהטיפוס כללי יותר האלגוריתם עצמו כללי יותר (contra-variance) דבר זה אינו בטוח (ולפעמים אף אינו נכון) מבחינות לוגיות

דוגמא

- המחלקה POINT מממשת את פעולת ה conjugate המחזירה נקודה סימטרית לנקודה הנוכחית

```
/** Conjugate of this */
POINT POINT::conjugate()
{
  POINT rslt = *this;
  rslt.move(0,2*y);
  return rslt;
}
```



חלקיק ונקודה

- כדי לממש חלקיק (PARTICLE) נירש מהמחלקה POINT (ובטח נוסיף תכונות כגון מהירות, מסה ואחרות)
- ואולם המתודה conjugate אשר באופן הגיוני אמורה לפעול גם על חלקיק לא תעבוד כמצופה:

```
PARTICLE p1, p2;
p2 = p1.conjugate(); // ERROR
```

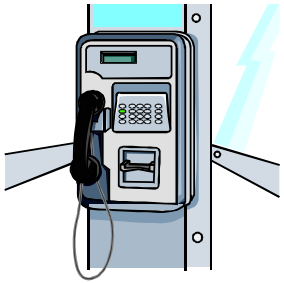
פתרון ע"י הגדרת עוגן

- ראינו פתרון בשפת Eiffel אשר מגדיר עוגן לטיפוס ומצמיד את שאר הטיפוסים לעוגן:

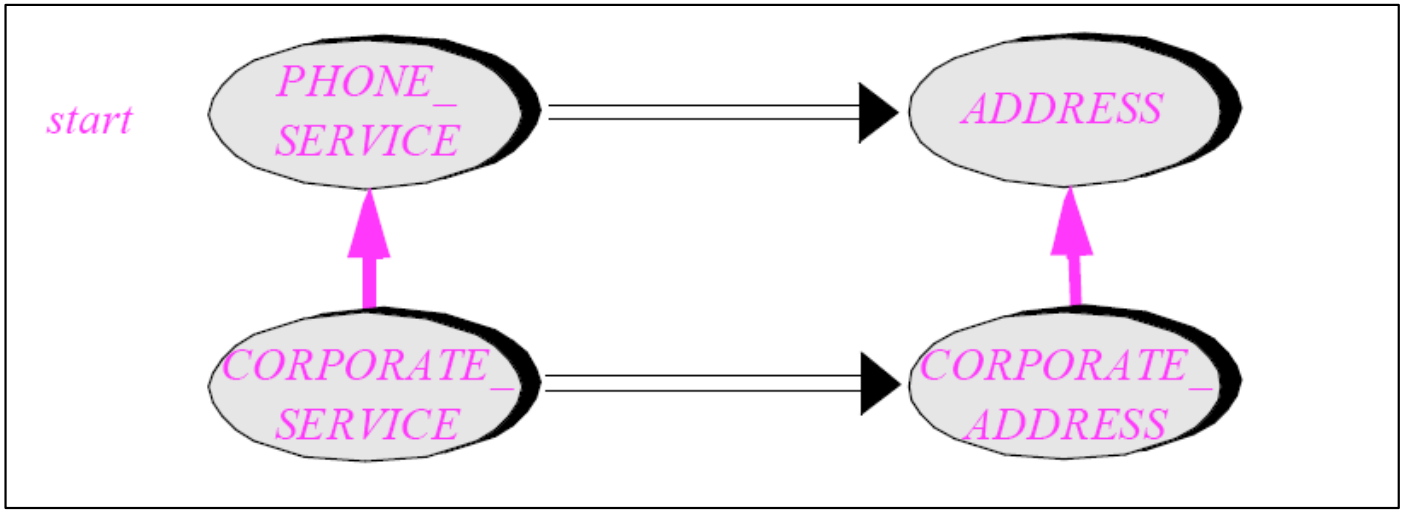
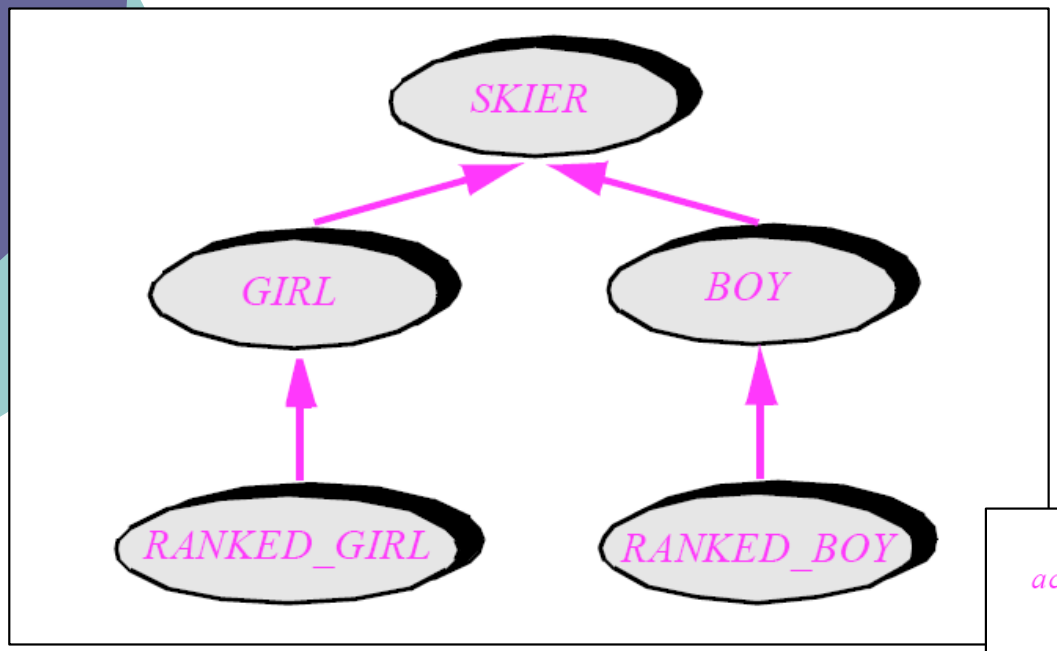
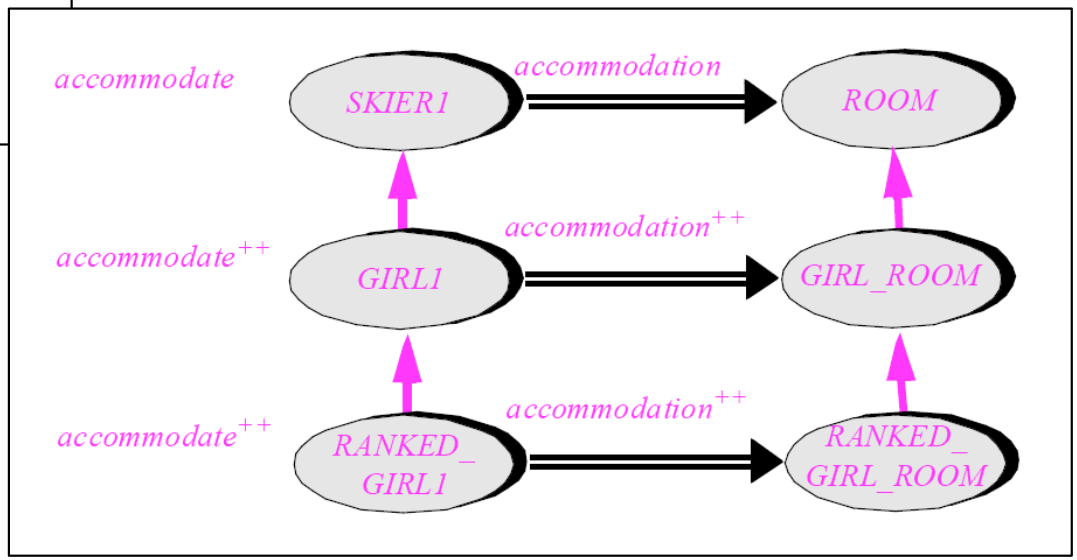
```
/** Conjugate of this */
like *this POINT::conjugate()
{
   like *this rslt = *this;
   rslt.move(0,2*y);
   return rslt;
}
```

- ב C++ ניתן לממש רעיון זה בחלקו ע"י הפיכת העוגן לטיפוס פורמלי של תבנית. בדוגמא לעיל לא ניתן להפוך את העוגן (this) לטיפוס פורמלי מכיוון שהדבר יוצר הגדרת תבנית רקורסיבית

דוגמאות נוספות



- מקרים שבהם ניתן לממש את הרעיון בעזרת תבניות הם מקרים של היררכיה מקבילה:



דוגמאות נוספות