עצמים תחילה? עצמים אחר כך? אולי ... עצמים לעולם לא!

פרופ' מוטי בן-ארי

המחלקה להוראת המדעים
מכון ויצמן למדע

הקדמה

במשך שנים רבות לימדתי שפות תכנות וכתבתי ספרי לימוד עליהן. המבנים לתמיכה בתכנות מונחה עצמים (תמ"ע) היו נושא מרכזי, וידעתי להסביר איך משתמשים בהם ולמה הם טובים. היתה רק בעיה אחת: אני אישית לא השתמשתי בתמ"ע! במהלך חיי, למדתי מספר רב של שפות תכנות, ובהרבה מקרים תגובתי המיידית היתה: איזה כיף, שפה זו פותרת לי בעיה שנתקלתי בה פעמים רבות. משם קצרה הדרך לניצול האפשרויות ששפת התכנות החדשה סיפקה לי. לא כך הרגשתי כאשר למדתי תמ"ע. בעשור האחרון כתבתי לא מעט תכניות לא קטנות בג'אווה — שהיא שפה **המחייבת** שימוש במבנים של תמ"ע כגון מחלקה, עצם, פעולה בונה, וכדומה — אבל זכורה לי רק פעם אחת שהרגשתי שהשימוש בתמ"ע שיפר את מבנה התוכנה שכתבתי. מנגד, היו מקרים לא מעטים בהם תמ"ע הקשה עלי.

אחת הטענות לטובת תמ"ע היא שהגישה מקלה על תחזוקת תוכנה ושימוש חוזר בה. בשני מקרים ניסיתי זאת, וכפי שאומרים בשפה עממית "אכלתי הרבה מאוד קש", עד כדי כך שבמקרה אחד ויתרתי על הניסיון לעשות שימוש חוזר בתוכנה. במאמר זה, אציג את ההתנסויות שלי ואת המסקנות מהן שהביאו אותי למצב בו אני נמנע ככל האפשר משימוש בתמ"ע. בסוף המאמר אדון בלקחים שראוי להפיק.

מהו תמ"ע?

כדי להגדיר מהו תמ"ע אשתמש בהגדרתו של Peter Wegner, אחד מהחלוצים בנושא (Wegner, 1987):

תמ"ע = עצמים + מחלקות + ירושה.

המאפיין של תמ"ע בעיני הוא העיקרון שכל "דבר" הוא "עצם" ושהמחלקה היא יחידת התכנון והמימוש היחידה בה ניתן להשתמש. מאפיין נוסף הוא ההכמסה המלאה של תכונות העצמים והפעולות עליהם בתוך הגדרת המחלקה. לפי מומחים כגון Wegner השימוש בירושה נחשב כמאפיין המבדיל בין תמ"ע "לייט" לבין תמ"ע אמיתי.[2]

שימו לב שאינני טוען נגד הרעיונות הבסיסיים של חלוקת התוכנה והכמסה, אלא בקיצוניות בהפעלתן שהיא לב לבו של תמ"ע כפי שמלמדים אותו. לשם השוואה, חשבו לדוגמה על Turbo Pascal (ז"ל, באמת זיכרונו לברכה) שאיפשר חלוקת תוכנה ליחידות וחלוקת כל יחידה לממשק ולמימוש. כאן מדובר באמצעי הניתן למתכנת כדי לקבוע את מבנה התוכנה, ולא ב"גישה" או "פרדיגמה" המכתיבה את אופן החלוקה. היום יש דרכים מתקדמות יותר וגמישות יותר לחלוקת תוכנה, אבל כל אחד יכול לחוש בחוסר הגמישות בשפת תכנות התומכת רק במבנה אחד לחלוקת תוכנה (המחלקה) וכך מעודדת רק פיתוח תוכנה לפי עקרונות תמ"ע.

---

1. מבוסס על המאמר Objects, never? Well, hardly ever! שיתפרסם ב- Communications of the ACM.

2. מומחים שהתכתבתי איתם לאחרונה מודים שירושה היא בעייתית ושברוב המקרים עדיף להשתמש בממשק (interface). בגלל המקום המרכזי של ירושה במשך כעשרים שנה, אני עדיין ממשיך ליחס חשיבות לירושה כעיקרון מרכזי בתמ"ע. יש גם מעט מאוד חומרי למידה שמדגישים את מבנה הממשק במקום מבנה הירושה.

התנסות א: שינוי בתוכנה קיימת

לפני שנים רבות פיתחתי מדמה מקביליות המשמש להוראת חישוב מקבילי. מדמה מקביליות מאפשר לתלמיד לחוות כי הרצת תכנית מקבילית מספר פעמים איננה מניבה תוצאות זהות. כל הרצה שוזרת באופן מקבילי הוראות אחרות על ציר הזמן ולכן יכולות להתקבל תוצאות שונות. הסביבה שפיתחתי מאפשרת לתלמיד לשלוט על ביצוע של תכנית מקבילית ברמה של משפט בודד (כי ניתן לבחור מאיזה תהליך ייבחר "המשפט הבא לביצוע").

התוכנה התבססה על מהדר ומפרש (interpreter) לגרסה פשוטה של פסקל שנכתבו על ידי Niklaus Wirth הממציא של פסקל. עם השנים עשיתי שינויים רבים בתוכנה (כגון מעבר ל-Turbo Pascal וחלוקה ליחידות, הוספת ממשק למשתמש, הרחבה של המבנים הנתמכים בשפה), אבל שימוש בסביבה הראה שתלמידים – הרגילים לממשקים גרפים מודרניים – אינם שבעי רצון מהתוכנה.

הרעיון של מדמה מקביליות הכה שורשים ומספר גרסאות אחרות פותחו. מצאתי מפרש שנכתב על ידי סטודנט לתואר שני בארה"ב. המפרש צויד בממשק גרפי יפהפה שאיפשר מעקב נוח וברור אחר ביצוע התכנית. עיון בקוד המקור גילה שלסטודנט יכולת מדהימה בכתיבה תוכנה בג'אווה. התוכנה יכולה לשמש כדוגמה ומופת של תמ"ע. רציתי להרחיב את הממשק לסביבת פיתוח המאפשרת עריכה והידור, ולהוסיף תמיכה במספר מבנים בשפה שהיו נחוצים לנו.

אין ספק שמאחר שהתוכנה פותחה לפי הגישה של תמ"ע יקל לערוך בה את השינויים.

אז לא.

המהדר של המדמה מוציא "קוד בתים" (byte code) הדומה, למשל, לקודים שמוציא המהדר של ג'אווה (JVM). הקודים אמורים להתבצע על מחשב ווירטואלי, ותפקיד המפרש הוא לבצע אלגוריתמים "כאילו" שמחשב זה אכן קיים. בניית מפרש אינה משימה קשה במיוחד: כל שעלינו לעשות הוא להגדיר מבני נתונים עבור הזיכרון והאוגרים של המחשב הווירטואלי, ואז לכתוב משפטים המדמים את הקודים השונים.

להלן הסבר קצר — שניתן לדלג עליו — עבור קוראים שמתעניינים.

***** הסבר קצר *****

במקום אוגרים הקיימים במחשבים אמיתיים, מחשבים וירטואליים משתמשים במחסנית לשמירת ערכים תוך כדי חישוב. משפט כגון x := y + 2 יתורגם לקודים:

```
load address y
load 2
add
store address x
```

מבנה האלגוריתם העיקרי של המפרש נראה בערך כך:

```
while (more instructions to execute)
   code := read next byte code
   op := the operand of this code
   case code of
      load: push the value of "op"
            onto the stack
      load address: read the value at address
            "op" in the memory and push it
            onto the stack
      store address: pop a value from the top
            of the stack and write it to the
            memory at address "op"
      add:  pop the top two values from the
            stack, add them, and push the
            result back onto the stack
```

עבור הקוד לחיבור, דרושים רק שני משפטי השמה:

```
begin t:=t-1; s[t] := s[t] + s[t+1] end;
```

כאשר s הוא מערך השומר את המחסנית ו- t שומר את ראש המחסנית.

***** סוף ההסבר *****

מספר הקודים במחשב הווירטואלי הוא כ-80, ולכן מדובר במבנה בקרה case עם 80 ברירות, רובן של שורה אחת או מספר קטן של שורות[3]. זה בכלל לא קשה לדפדף בתכנית זו ולהוסיף תמיכה לקודים חדשים.

מה עם תמ"ע? כל אחד מהקודים המרכיבים את התכנית הוא כמובן "עצם", ולכן באופן טבעי כל סוג של קוד (כגון "חיבור", "חיסור") יוגדר כמחלקה. לקוד עבור חיבור יש 13 שורות (לא כולל כ-20 שורות של תיעוד):

```
package baci.program;
import baci.interpreter.*;
public class InstructionDoAdd extends
        InstructionDo {
    public InstructionDoAdd(
            Program program,  int x, int y) {
        super(program, x, y);
    }
    protected int doOp(int arg1, int arg2) {
        return arg1+arg2;
    }
    protected String getDescription() {
        return "…";
    }
}
```

מסביב לשורה אחת של ביצוע return arg1+arg2, יש לנו הגדרת package, import, הגדרת המחלקה ופעולה הבונה עם מספר פרמטרים![4]

באמת יפה. אבל כדי לקרוא ולהבין את תכנת המפרש, הייתי חייב לדפדף בין כ-80 **קבצים** (אחד לכל קוד). זו משימה מסורבלת גם בעזרת סביבת פיתוח מתקדמת כגון Eclipse.

אני מסיק שהגישה של תמ"ע מונעת שימוש בשיקול דעת: אינני "רשאי" להחליט שהטיפוס "קוד" הוא פשוט **רשומה** עם מספר שדות שאין למעשה שום סיבה להסתיר את קיומם:

```
type byte_code =
        record
                code: byte;
                operand1: integer;
                operand2: integer;
        end;
```

בתמ"ע, עלי לקבוע שמדובר במחלקה עם שלוש תכונות פרטיות ושלושה זוגות של פעולות set ו-get כדי לקרוא ולכתוב אותן.

המעבר ממשפט בקרה אחד לאוסף של עצמים היה טרדה קטנה. הבעיה העיקרית נבעה מההכמסה הקיצונית הנובעת משימוש בתמ"ע. כדי לקרוא או לכתוב נתון מסוים, יש לקבל גישה לעצם שמכיל את הנתון. נניח שאנו כותבים תכנית כלשהי ואנו רוצים לבדוק אם החישוב מתבצע במצב "ניפוי שגיאות":

```
public class MyClass {
    public doSomething() {
        if (debugMode) {
            System.out.println("print some
                        debug information");
        }
    }
}
```

לפי תמ"ע, הדגל המצביע על מצב "ניפוי שגיאות" עשוי להיות תכונה בעצם ממחלקה Debug:

```
public class Debug {
    private boolean inDebuggingMode;
    public Boolean getDebugMode() {
        return inDebuggingMode;
    }
}
```

3. הקוד המסובך ביותר הוא עבור זימון פונקציה המחייב 17 שורות, באמת לא הרבה.

4. השימוש בתמ"ע בתוכנה זו היה מתוחכם הרבה יותר והתבסס על מחלקות abstract וירושה.

המחלקה שלנו חייבת לקבל פרמטר מטיפוס Debug בפעולה הבונה, לשמור את ההפניה לעצם זה במשתנה מקומי, ודרכו לבקש את ערך הדגל:

```
public class MyClass {
    private Debug debug;
    public MyClass(Debug debug, …) {
        this.debug = debug;
                …
    }
    public doSomething() {
        if (debug.getDebugMode()) {
        System.out.println("print some
                debug information");
        }
    }
}
```

זה מסורבל — אבל לא נורא קשה — אם כך תוכנן ותוכנת מראש. אבל מה קורה אם תוך כדי שימוש חוזר בתוכנה עלינו לבצע שינוי זה **ולהוסיף** קריאה של הדגל למחלקה MyClass? כעת עלינו לחפש את **כל העצמים** מטיפוס MyClass ולהוסיף לזימון של הפעולה הבונה פרמטר שהוא עצם מטיפוס Debug:

```
class YourClass {
    MyClass myclass354 =
        new MyClass(debug, …);
}
```

ואם העצם debug לא מוגדר במחלקה YourClass, נצטרך להוסיף אותו לפעולה הבונה שלה, ולשנות את כל הזימונים לפעולה הבונה בכל המחלקות האחרות, ואם העצם debug ... . חד גדיא, חד גדיא.

בתכנית זו עם אלפים רבים של שורות, קרה לא פעם ולא פעמיים שעצם שרציתי לא היה בנמצא, וזה גרר סדרה ארוכה של שינויים כדי להבטיח שהעצם יגיע דרך שורה של פעולות בונות.

הנה דוגמה נוספת לסרבול שיצר תמ"ע. כאשר לוחצים על הכפתור בממשק למשתמש כדי להציג את חלון ההיסטוריה (של קודים או משפטים שהתבצעו), הפעולה המטפלת באירוע כוללת את המשפט הבא:

```
getDebugger().
    getDebuggerFrame().
        getWindowManager().
            showHistoryWindow(
                getDebugger(),
                getDebbuger().getInterpreter());
```

אני מבין את הצורך להגדיר מחלקה לחלונות כי יש כמה מופעים של המחלקה Window עם ערכים שונים לתכונות שלה. אבל יש רק מנפה שגיאות (Debugger) *אחד*, מסגרת (DebuggerFrame) *אחת*, מנהל חלונות (WindowManager) *אחד* ומפרש (Interpreter) *אחד*. **רכיבים אלה הם תת-מערכות ולא עצמים**. האם באמת נחוץ להגדיר אותם כמחלקה עם הכמסה? ההכמסה מחייבת שימוש באינסוף זימוני get, וגורמת להסתבכות בשינויים אינסופיים בפעולות הבונות כדי להעביר הפניות של העצמים האלה שהם המופעים היחידים של המחלקות שלהם. שוב, במקום להפעיל שיקול דעת בהגדרת יחידות ובמידת ההכמסה, תמ"ע גרם לכתיבת קוד שהוא מסורבל מאוד לקריאה ולשינוי.

התנסות ב: שימוש חוזר בתוכנה

הטיעון המוחץ בעד תמ"ע הוא שתמ"ע מאפשר שימוש חוזר בתוכנה. במסגרת כלי שפיתחתי להוראת המושג אי-דטרמיניזם (nondeterminism), נזקקתי לתיאור של מכונות סופיות לא-דטרמיניסטיות (nondeterministic finite automata), והשתמשתי בקבצים שנוצרו על ידי חבילת התוכנה ללימוד מודלים חישוביים ושפות פורמאליות הנקראת JFLAP (jflap.org). בהמשך רציתי לשלב בתוכנה שלי את העורך הגרפי של JFLAP. אין טבעי מזה כמקרה מייצג של שימוש חוזר בתוכנה שפותחה בגישה של תמ"ע.

אז ניסיתי.

לא הלך.

הבעיה הייתה שיחידת התוכנה של תמ"ע היא המחלקה, **והמחלקה היא יחידה קטנה מדי** המתאימה אולי לשימוש חוזר של יחידת ספריה פשוטה. אבל מה שנחוץ כאן היה שימוש חוזר של **תת-מערכת**. העורך הגרפי מורכב מכ-40 מחלקות מתוך 400 המחלקות של JFLAP, אבל לא הצלחתי "לשלוף" את אוסף המחלקות הזה. הסיבה היא שכאשר עובדים עם מחלקות, אין מניעה להשתמש בתוך מחלקה אחת בכל מחלקה אחרת. בג'אווה, למשל, פשוט רושמים משפט import בראש המחלקה, או, גרוע מזה, מספקים את מלוא השם של המחלקה. אם השימוש נעשה "עמוק" בתוך המחלקה אין אפילו רמז לתלויות בין המחלקות, והתלות מתגלה רק כאשר מנסים להדר את המחלקה.

נניח, למשל, שבאמצע התכנות של העורך הגראפי העוסק (כך קיוויתי) בצמתים וקשתות באופן כללי ללא התייחסות לעובדה שהגרפים מתארים מכונות, יש זימון לפעולה הנמצאת במחלקה שהיא חלק התכנית המטפלת במכונות סופיות:

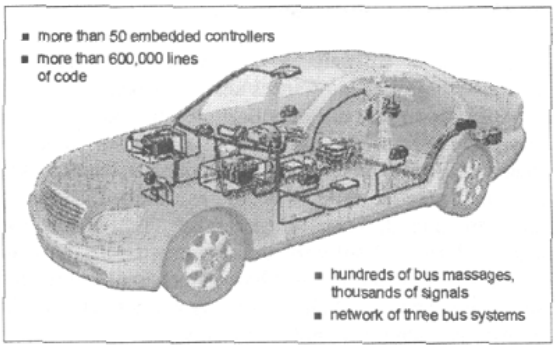
```
int max =
    jflap.finiteAutomata.NDFA.getMaxStates();
```

לאחר שאשלוף את המחלקות של העורך, אני אגלה את התלות רק כאשר אנסה להדר את העורך. בהעדר המחלקה NDFA (שלא שלפתי כי היא בכלל ב-package אחר שאינו קשור לעורך), המהדר לא ימצא את הפעולה ויודיע על הודעת שגיאה. ואכן, כל ניסיון לשלוף את העורך הגראפי נתקל בשגיאות הידור שהצביעו על שימוש במחלקות אחרות, שהן בתורן השתמשו במחלקות נוספים, ... חד גדיא, חד גדיא. כדי להבין את 40 המחלקות של העורך, הסתבר שאצטרך ללמוד חלק ניכר מ-400 המחלקות של התוכנה. נשברתי ומצאתי פתרונות אחרים[5].

5. תחילה צירפתי את קובץ ה-jar של JFLAP **בשלמותו** לתוכנה שלי, למרות שהוא היה גדול פי כמה מהתוכנה שאני פיתחתי. החיסרון בפתרון זה הוא שהמשתמש מקבל ממשק לכל JFLAP ולא ממשק מינימאלי לעורך גרפי. בהמשך מצאתי תוכנה פשוטה יותר שיכולתי לשלב בתוכנה שלי.

תמ"ע והעולם האמיתי

לפני מספר שנים נתקלתי בתרשים שלהלן (Grimm, 2003):



(התרשים באדיבות Klaus Grimm מחברת Daimler)

התרשים מראה מכונית מרצדס חדשה (S class, למי שמתעניין) המכילה מעל 50 בקרים מתוכנתים ומעל 600,000 שורות תכנות! לא הצלחתי לקבל מידע מפורט על מערכת מחשוב זו, אבל אין לי ספק מה **לא עשו**. אף אחד לא ישב וכתב הגדרות לתת המערכות השונות כפי שאנו מצפים מתלמיד הלומד תמ"ע. למשל:

```
public class Brake {
        private int oilPressure;
        private int diameter;
        public void applyBrakes(int force);
}
```

כמעט וודאי הוא שהמערכות השונות נרכשו מחברות בנות או מקבלני משנה, ואין ספק שהמערכות התבססו על מערכות שפותחו קודם עבור מכוניות אחרות, אפילו מכוניות של המתחרים. כפי שמראה התרשים, הממשק ביניהן הוא לא ברמה של **תוכנה**, אלא ברמה של **פרוטוקולי תקשורת** בין הבקרים השונים[6]. הממשקים מוגדרים כהתנהגויות כגון:

אם האות מקו 3 דלוק למשך 50 מילישניות לפחות
אז
הפעל את הבלמים תוך הפעלת לחץ המחושב
לפי הנוסחה: ... .

6. טענה זו ברורה מהשימוש במלים bus messages ו-signals.

תיאורים כגון אלה חורגים ממה שאפשר לכתוב בתמ"ע[7]. התנהגויות ואלגוריתמים נכתבים בתרשימים ונוסחאות, ומתורגמים למשפטי השמה ומשפטי בקרה. נראה לי שהוראת אלגוריתמים וחישובים איבדה מחשיבותה מרגע שתמ"ע הפך לנושא לימוד מרכזי. חבל.

אני משער שתהליך הפיתוח של המכונית התנהל בערך כך: חברת Daimler המייצרת את מכונית המרצדס פנתה למפעלים שונים כדי לחפש קבלני משנה לפיתוח תת מערכות. הארכיטקט הראשי של המכונית הגדיר את פרוטוקולי התקשורת לכל תת המערכות, וכל יצרן **התאים את המערכת הקיימת שלו לממשק זה**. תהליך זה **מנוגד לחלוטין** לעיקרון בסיסי של תמ"ע שהוא: הגדר ממשק והחלף מימוש (ללא צורך לשנות את הממשק). כאן, המימוש של תת המערכות משתנה רק במעט הדרוש למכונית החדש, אבל **הממשק מוחלף במלואו** כדי להתאים לדרישות המיוחדות של כל יצרן של מכוניות. העקרונות המכאניים לבלימה הם בערך אותו דבר ללא קשר לדגם המכונית (מרצדס או הונדה או אלפא רומיאו או וולבו), אבל אני משוכנע שמערכות האלקטרוניקה המבקרות את הבלמים שונות לחלוטין בין יצרן אחד לשני. בעידן של סחר גלובלי, בנייה מוצלחת של מוצר מחייבת חיפוש תת מערכות קיימות בכל מקום אפשרי ושילובם ביחד כאשר **עיקר ההתאמות הוא בממשק ולא במימוש**.

החידה: למה "כולם" משתמשים בתמ"ע?

הרהורי הכפירה המתוארים לעיל התחילו להתגבש אצלי כאשר שמעתי דיון בין תומכי הוראה בגישה של "עצמים תחילה" לבין המתנגדים לגישה הזו (Astrachan et al., 2005). אחד התומכים, Kim Bruce (שהוא מומחה לתמ"ע), טען לטובת "עצמים תחילה" שמדובר **בגישה המובילה בפיתוח תוכנה**, ולכן כדאי להתחיל איתה מוקדם ככל האפשר. מאוחר יותר, ביקשתי לדעת מאיפה הקביעה שתמ"ע הוא הגישה המובילה, אבל הוא לא היה מסוגל לספק לי ראיות כלשהן לטענתו. פניתי למספר מומחים נוספים וגם הם לא הצליחו למצוא ראיות למקומו המרכזי של תמ"ע. לא הצלחתי גם לקבל הפנייה למחקר כלשהו שיוכיח מה באמת היתרונות בתמ"ע (אם בכלל), ולאיזה סוג של תוכנה גישה זו מתאימה. לכל שיטת תכנות ולכל שפת תכנות יש תכונות העוזרות בפיתוח סוג מסוים של תוכנה ותכונות המקשות על פיתוח סוג אחר, וכל תומך בשימוש בגישה מסוימת או בשפה מסוימת יכול לאפיין בקלות את סוג התוכנה "שירוויח" ממנה. מתי תומכי תמ"ע ינסחו את היתרונות והחסרונות של הגישה לסוגים שונים של תוכנה?

אני משער שיש שני מקורות לרושם ש-"כולם" משתמשים בתמ"ע. הראשון הוא שיש שימוש רחב **בשפות תכנות** התומכות בתמ"ע. אולם שימוש בשפה לא מוכיח שהתכנות נעשה לפי גישה כלשהי, וגם סביר מאוד שחלק ניכר מהטוענים שהם משתמשים בתמ"ע, לא עושים את זה נכון או עושים את זה נכון ולא מפיקים תועלת ממשמעותית ממנו. לכן, שימוש בשפה התומכת בתמ"ע לא מהווה ראיה לשימוש רחב בתמ"ע עצמו או ליתרונות שלו.

המקור השני, והחשוב יותר, לרושם המוטעה הוא שרוב העיסוק בפיתוח תוכנה הוא בלתי נראה. הספרייה של ג'אווה לפיתוח ממשק למשתמש – המתבססת על תמ"ע – מוכרת ל-"כולם", אבל האם בכלל עלה בדעתך שיש 600,000 שורות קוד במכונית החולפת לידך? האם חשבת על התוכנה בנגני מוסיקה, טלפונים סלולאריים ותחנות המיתוג שלהם, במטוסים, ובמחשבי הבנקים וחברות הביטוח? אפילו אם קיימת מודעות לתוכנה זו, האם יש לך מושג איך מפתחים אותה? באיזו שפות תכנות? סביר שלא, כי פיתוח זה נעשה תחת מעטה כבד של סודיות מסחרית. עולם התכנות עשיר הרבה יותר ממה שרוב האנשים חושבים, וחבל שנוצר רושם שיש אחידות בתהליכי הפיתוח או בכלי הפיתוח.

האם תמ"ע הוא "טבעי"?

הלומד תמ"ע מרגיש לעיתים תסכול, כי תומכי הגישה טוענים שתכנון לפי תמ"ע הוא "טבעי" ו-"אינטואיטיבי" כי כל "דבר" הוא "עצם". יכול להיות שכל דבר הוא עצם, אבל פיתוח תוכנה עוסק לא ב-"דברים" אלא בבנייה של מודל פורמאלי למציאות.

7. לעתים, פיתוח תוכנה לפי תמ"ע נעשה במסגרת כלי תכנון התומכים בסימונים הנקראים UML. ב-UML אפשר לבטא ניסוחים כגון אלה, אולם, עדיין נשארת השאלה איך לממש תנאי מסוג זה בתמ"ע.

"כפתור" בממשק למשתמש הוא לא באמת כפתור אלא שטח מוגדר על המסך שגורם להתנהגות מסוימת של התוכנה אם מביאים אליו את העכבר (שהוא, כמובן, לא באמת "עכבר") ולוחצים על אחד מהלחצנים. אין סיבה להאמין שמה שטוב ל-"דברים" הוא טוב לאפיון ופיתוח תוכנה. סביר שיש מקרים שכן, ובמקרים (רבים) אחרים, לא.

במחקר מעניין, עירית הדר ואורי לירון מהטכניון חקרו את קליטת המושגים של תמ"ע על ידי **מפתחי תוכנה מנוסים**. הם מצאו ש"כאשר יש דרישה להפשטה, לפורמאליות, ולצורך בביצוע על מחשב, הפרדיגמה הפורמאלית של עצמים מתנגשת לעיתים דווקא עם האינטואיציות שהולידו אותה" (Hadar & Leron, 2008, p. 45). אני מפרש את הממצאים שלהם בדרך אחרת. לדעתי, אין שום בעיה עם האינטואיציה של מהנדסי התוכנה המנוסים, ושהצד הלא אינטואיטיבי בסיפור הוא דווקא תמ"ע. לכן, אין סיבה להרגיש תסכול אם אתם מתקשים לראות בתמ"ע משהו טבעי.

אז מה לעשות?

גיליתי שאני לא לבד במחשבות כופרות בכל הקשור לתמ"ע ויש רבים החושבים כמוני[8]. אבל, אי אפשר להכחיש שלתמ"ע הגמוניה כמעט מוחלטת בתכניות הלימודים למדעי המחשב בעולם כולו. אני מרשה לעצמי להשיא מספר עצות למחנכים במדעי המחשב:

- אם עליכם ללמד תמ"ע, למדו אותו כמיטב יכולתכם בלי להרגיש תסכול. ייתכן שתרגישו שמשהו בגישה לא טבעי או מסורבל או קשה להבנה. המושגים באמת קשים להבנה ועוד יותר מזה מסובכים להפעלה נכונה, והיכולת לגבש דעה עצמאית על כל נושא מורכב מחייבת תהליך ארוך של "אכילת קש".
- במידת האפשר, אל תדכאו תלמידים המפעילים שיקול דעת ומציעים פתרונות אחרים. אמנם, ייתכן שהבחינה תדרוש מבנה מסוים, אבל כבדו את דעתם של התלמידים, כי ייתכן שפתרונותיהם באמת טובים יותר, גם אם הם לא בדיוק תואמים לתמ"ע.
- החטא הקדמון בהוראת מדעי המחשב הוא התמחות יתר בשפת תכנות אחת ובפיתוח סוג מסוים של תוכנה. אמנם קשה במסגרת הלימודים הפורמאליים למצוא זמן ללמד מגוון רחב של שפות, אבל במסגרת פרויקטים או פעילות חוץ-כיתתית, עודדו את התלמידים להכיר שפות תכנות ושיטות תכנות אחרות.
- אל תפחדו לדרוש הסברים והנמקות מפורשים מהתומכים בתמ"ע. במיוחד בקשו תיאור מפורט של התנסויות (אישיות או מהספרות) היכולות לתמוך בדעותיהם. בדרך זו תוכלו לגבש לעצמכם דעה אישית על תמ"ע: בעד, נגד, או במקרים מסוימים בעד ובמקרים אחרים נגד.

הבעת תודה: ברצוני להודות למיכל ארמוני ולנוע רגוניס על הערותיהן על המאמר. כמובן שהדעות שהבעתי כאן הן שלי ולא בהכרח שלהן.

מקורות

Astrachan, O., Bruce, K., Koffman, E., Kölling, M., and Reges, S. 2005. Resolved: Objects early has failed. *SIGCSE Bull.* 37, 1 (Feb. 2005), 451-452.

Grimm, K. 2003. Software technology in an automotive company: Major challenges. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 498-503.

Hadar, I. and Leron, U. 2008. How intuitive is object-oriented design?. *Commun. ACM* 51, 5 (May. 2008), 41-46.

Wegner, P. 1987. Dimensions of object-based language design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. N. Meyrowitz, Ed. OOPSLA '87. ACM, NY, 168-182.

8. אתר שמביא טיעונים טכניים רבים נגד תמ"ע
http://www.geocities.com/tablizer/oopbad.htm