מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 72
אוקטובר 2002

Design By Contract (תיכון עפ"י חוזה)

מבוא

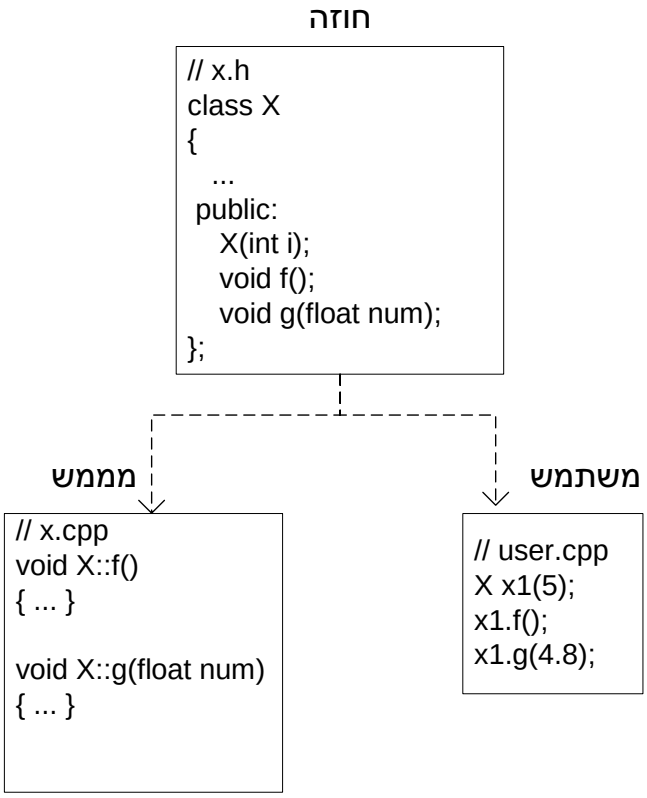
"**Design By Contract**" (**DBC**) הינה גישת תכנות הכוללת מספר עקרונות מנחים בפיתוח תוכנה מונחה עצמים. בגישה זו מספר מושגים - **Invariant**, **Pre-Condisions**, **Post-Conditions** ו- **Exceptions** - המתייחסים לעצמים ולשירותים שהם מספקים, ותפקידם בחלק הממשק (ה"חוזה") שבן מממש העצמים למשתמש בהם. מעט מאוד שפות תכנות תומכות באופן ישיר בכלל עקרונות DBC - הבולטת שבהן היא Eiffel, ומחבר מרבית עקרונות אלו הוא Bertrand Meyer (ראה/י סעיף References).

במאמר זה נסקור את עקרונות DBC. תוכן נושא זה מבוסס על סעיף התיכון "תיכון עפ"י חוזה" שבפרק 17 בספר "***C++ - מדריך מקצועי***", בהוצאת "**מרכז ההדרכה 2000**".

Design By Contract

הרעיון שעומד מאחרי הפרדת המחלקות לחלק הממשק ולחלק המימוש הוא **תיכון עפ"י חוזה**: ממשק המחלקה מהווה חוזה בין שניים:

- מממש המחלקה
- המשתמש במחלקה



מממש המחלקה מספק את קובץ ההכרזה (.h) הכולל את ממשקי הפונקציות ואת הכרזות המשתנים במחלקה. המהדר אחראי לקיום החוזה: הוא בודק את המממש מצד אחד - מימוש הפונקציה חייב להתאים להכרזתה (פרמטרים מועברים, ערך מוחזר, const וכו') - וכן את המשתמש מהצד השני, צד הפעלת הפונקציה.

ההנחה היא, שקובץ ההכרזה מספיק על מנת שהמשתמש ישתמש כראוי במחלקה - כלומר, יפעיל באופן נכון את פונקציות המחלקה. במידה ולא, המהדר אחראי להודיע לו על שגיאת הידור. אולם, קיימים לעתים תנאים "נסתרים" שאינם מיוצגים כהלכה בשפת התכנות.

לדוגמא, הפונקציה strcat() הכלולה בספרייה התקנית של C מוכרזת כך:

char *strcat(char ***dest, const char ***s*ource**);**

הפונקציה משרשרת את המחרוזת source לסופה של המחרוזת dest.

מספר שאלות:

- האם המשתמש חייב לדאוג למספיק מקום במחרוזת dest כך שתוכל להכיל גם את source, או שהפונקציה תקצה את הזיכרון בעצמה?
- האם מותר להעביר מחרוזת NULL כ- source? ואם לא, מהן ההשלכות של קריאה כזו?
- האם מותר להעביר שתי מחרוזות חופפות (Overlapped) בתוכנן, כלומר, שהן כוללות מספר תאי זיכרון משותפים?

ברור ששאלות אלו אינן באות לידי ביטוי ב C/C++, אם כי הן עדיין חשובות מאוד לצורך הפעלה תקינה של הפונקציה מצד המשתמש, ולכן יש צורך לתעד אותן באופן ברור.

DBC - מונחים

בתיכון מונחה עצמים עפ"י חוזה, קיימים מספר מושגים חשובים בהפעלת עצם מסוים. ראשית, עבור העצם מוגדר המושג הבא:

- **Invariant** - אוסף המצבים שבהם יכול להיות העצם במשך כל חייו: החל מרגע סיום ביצוע ה- constructor שלו ועד לתחילת ביצוע ה- destructor.

כמו כן, עבור כל פונקציה (שירות) של העצם מוגדרים 3 מושגים:

- **Pre-Conditions** - תנאים מוקדמים שצריכים להתקיים לפני הפעלת הפונקציה. אלו תנאים החלים על המשתמש בפונקציה.
- **Post-Conditions** - תנאים שצריכים להתקיים לאחר סיום הביצוע של הפונקציה. אלו תנאים החלים על מממש הפונקציה.
- **Exceptions** - חריגות שעלולות להיווצר בהפעלת הפונקציה.

יש לשים לב לכך שהחריגות, **Exceptions**, ניתנות לתיעוד במרבית שפות העצמים המודרניות.

Pre-Conditions

ה- Pre-Conditions הם תנאים החלים על המשתמש בפונקציה מסוימת של העצם. במצב האופטימלי אין תנאים מיוחדים - כלומר, ה- Pre-Conditions הם ה- Invariant של העצם.

לדוגמא, במחלקה הבאה,

```
class Worker
{
        char  m_name[20];
        long m_id;
public:
```

```
        void set(const char *name);
        void set(const long id);
};
```

מהם ה- Pre-Conditions של הפונקציה ()set בגרסה המקבלת מחרוזת? כפי שנאמר, לא ניתן להבין מהם תנאים אלו מתוך הכרזת הפונקציה בלבד. אולם, ניתן לשער שממש המחלקה מתכוון להעתיק את המחרוזת הנתונה כפרמטר למחרוזת החברה במחלקה, שארכה הקבוע הוא 20. מכאן, שעל מחרוזת הקלט להיות בעלת אורך מקסימלי של 19 תווים.

האם ניתן להעביר 0 (null) כפרמטר? גם תנאי זה אינו מתועד בכותרת הפונקציה, ויש לתאר האם שימוש מעין זה תקין.

ומהם ה- Pre-Conditions של הפונקציה ()set בגרסה המקבלת long? שוב, גם כאן ניתן לשער שמספר מזהה של עובד אינו יכול להיות שלילי, ולא 0.

לכן, הכרזה נכונה של המחלקה צריכה לכלול את ה- Pre-conditions של כל פונקציה:

```
class Worker
{
        char  m_name[20];
        long m_id;
public:
        // Pre-Conditions:  strlen(name) < 20   or name==NULL
        void set(const char *name);

        // Pre-Conditions:  id > 0
        void set(const long id);
};
```

Post-Conditions

ה- **Post-Conditions** הם התנאים החייבים להתקיים לאחר סיומה התקין של פונקציה נתונה. קיומם של תנאים אלו הם באחריות מממש המחלקה. לדוגמא, במחלקה הנ"ל ניתן להגדיר את ה- Post-Conditions הבאים:

- עבור set(const char*):
  1. לאחר סיומה של הפונקציה, ערכו של מספר הזהות m_id לא שונה.
  2. כמו כן, ללא תלות בפרמטר שהועבר, ערכה של m_name תקף (כלומר, אינו זיבלי).
- עבור set(const long):
  3. לאחר סיומה של הפונקציה, ערכה של המחרוזת m_name לא שונה.
  4. כמו כן, ללא תלות בפרמטר שהועבר, ערכו של m_id אינו שלילי.

בכדי להבין את הדרישות הנ"ל, נתבונן בקוד הלקוח הבא:

```
void user()
{
        Worker        w1;
```

מהו ערכם של w1.m_name ו- w1.m_id ? מכיוון שלא הוגדר constructor מחדל כלשהו, ערכם לא מוגדר, כלומר, זיבלי.

נניח שכעת המשתמש מבצע את הקריאה:

```
        w1.set("John");
```

עפ"י תנאי 1, w1.m_id נשאר בעל ערך זיבלי. כמו כן, w1.m_name בעל הערך התקף "John", וברור שגם תנאי 2 מתקיים.

המשתמש ממשיך ומבצע את הקריאה הבאה:

```
w1.set(12345678);
```

למספר הזהות מוצב ערך תקף (תנאי 4 מתקיים). עפ"י תנאי 3, ערכו של w1.m_name נשאר "Jhon".

כעת, נניח שהמשתמש מבצע את הקריאה הבאה:

```
char *p = 0;
w1.set(p);
```

העברת מצביע null חוקית, עפ"י המתועד ב- Pre-Conditions של הפונקציה. אולם מהי תוצאת הפעלה כזו? עפ"י תנאי 1, ערכו של מספר הזהות, *w1.m_id* לא השתנה (12345678).

עפ"י תנאי 2, ערכו של *w1.m_name* כעת אינו זיבלי - ניתן להניח שערכו הוא "", כלומר, מחרוזת ריקה (אם רוצים, ניתן גם לרשום זאת במפורש כ- Post-Conditions במקרה שמועבר null כפרמטר).

כך יוגדרו ה- Post-Conditions הנ"ל במחלקה:

```
class Worker
{
        char  m_name[20];
        long m_id;
public:
        // Pre-Conditions:  strlen(name) < 20   or name==NULL
        // Post-Conditions: no change to m_id,  m_name is valid
        void set(const char *name);

        // Pre-Conditions:  id > 0
        // Post-Conditions: no change to m_name,  m_id is valid
        void set(const long id);
};
```

Exceptions

Exceptions הן חריגות העלולות להיזרק מתוך פונקציה נתונה, אם כתוצאה מהפעלה שגויה (שלא עפ"י ה- Pre-Conditions) של המשתמש, ואם כתוצאה מכישלון בזמן ריצה מסיבה אחרת כלשהי.

C++ מאפשרת להכריז בכותרת הפונקציה על החריגות העלולות להיזרק ממנה. אולם ב- C++ מנגנון זה אינו כפוי מצד אחד על מממש המחלקה, וכן אינו נבדק ע"י המהדר מצד שני (זוהי שגיאת ריצה לזרוק חריגה בניגוד למוגדר בהכרזת הפונקציה, ונקראת הפונקציה ()unexpected).

ב- Java לעומת זאת מנגנון ה- Exceptions נבדק בזמן קומפילציה, וכל פונקציה חייבת לבצע אחד מהשניים: לטפל בחריגה העלולה להיזרק ממנה, או להכריז אל אפשרות זריקתה ("Declare or Handle").

לדוגמא, במחלקה Worker הנ"ל ניתן להכריז על חריגה הנזרקת כתוצאה מהעברת שם ארוך מדי לפונקציה ()set (הפרת ה- Pre-Conditions):

```
class Worker
{
        char  m_name[20];
        long m_id;
public:
        // Pre-Conditions:  strlen(name) < 20   or name==NULL
        // Post-Conditions: no change to m_id,  m_name is valid
        void set(const char *name) throw (overflow_error);

        // Pre-Conditions:  id > 0
        // Post-Conditions: no change to m_name,  m_id is valid
```

```
        void set(const long id);
};
```

מצב העולם/עצם לאחר זריקת חריגה

מובן שזריקת חריגה הינה פעולה יוצאת דופן שאינה מבטיחה קיומם של ה- Post-Conditions. כלומר, רק סיום תקין של הפונקציה מבטיח את קיומם.

אולם, עדיין ניתן לשאול מספר שאלות בנוגע למצב ה"עולם" לאחר זריקת החריגה:

- האם זריקת חריגה מבטיחה את שמירתו של ה- Invariant ?
- במידה וה- Invariant נשמר, מהו מצב העצם?
- האם יש צורך בביצוע פעולות איפוס מסויימות עקב זריקת החריגה?

Stroustrap דן בהרחבה בנקודות אלו ובנושא Exception Safety במאמר:

***B. Stroustrup**: **Programming with Exceptions**, InformIt.com. April 2001.*
http://www.research.att.com/~bs/ eh_brief.pdf

באופן כללי, הוא מציין 3 גישות ביכולת להבטיח את מצב העולם/העצם לאחר זריקת חריגה:

1. לא מובטח דבר.
2. מובטחת שמירת ה- Invariant.
3. מצב העצם משוחזר למקורו (Roll-Back), למצב שקדם לקריאה לפונקציה שגרמה לחריגה.

ככלל, יש להשתדל לשמור על מצב ה- Invariant במידת האפשר. כלומר, יש להבטיח את אפשרות 2 בכל מקרה, במידת האפשר. הבטחת אפשרות 3 בד"כ כרוכה בתשלום בביצועים, עקב הצורך לשמור מידע שיסייע בשיחזור מצב העצם.

Invariant

Invariant אינו קשור להפעלת שירות מסוים של העצם: זוהי הגדרה של אוסף המצבים שבהם יכול להימצא עצם מהמחלקה. יש לשים לב לכך שמצב העצם מוגדר על פי ערכי הנתונים בלבד, ולכן הגבלת העצם למצבים מסוימים היא הגבלת המשתנים שבו לערכים מסוימים.

ה- Invariant מתקיים החל מנקודת סיום ביצוע ה- constructor ועד לתחילת ביצוע ה- destructor.

במקרים רבים, אין הגבלה כלשהי על ערכי המשתנים של העצם, ואז ה- Invariant כולל את כל המצבים האפשריים. לדוגמא, במחלקה Worker שלעיל אין הגבלה כלשהי על ערכי התכונות של העצם.

נתבונן בגירסה מעט שונה של המחלקה Worker - המחרוזת m_name מוקצית כעת באופן דינמי:

```
class Worker
{
        char  *m_name;
        long    m_id;
public:
        Worker()  {}                                    // default constructor
        Worker(const char *name, const long id);        // constructor
        Worker(const Worker&);                          // copy constructor
        ~Worker();                                      // destructor
        Worker& operator=(const Worker&);               // operator "="

        // Post-Conditions: no change to m_id,  m_name is valid
        void set(const char *name);
```

```
    // Pre-Conditions:  id > 0
    // Post-Conditions: no change to m_name,  m_id is valid
    void set(const long id);
};
```

הערה: set() בגירסה הראשונה כעת אינה כוללת Pre-Conditions כלשהם.

שאלה: המחלקה בגרסתה הנוכחית כוללת באג חמור - מבלי לקרוא הלאה, האם הנך רואה אותו?

הבאג נובע מהגדרה לא נכונה של ה- Invariant של עצם מהמחלקה Worker: constructor המחדל של המחלקה אינו מבצע דבר, מה שמתפרש כ- Invariant לא מוגבל.

הסתכלות מעמיקה יותר על מבנה המחלקה מבהירה צורך בהגדרת Invariant: אסור למשתנה המחלקה m_name להיות זיבלי!

במספר פונקציות במחלקה עלול להיווצר צורך בשחרור הזיכרון הקודם ((destructor, operator=(), set()), ואם m_name בעל ערך זיבלי, התכנית "תעוף".

תיקון הבאג: ראשית יש להכריז על ה- Invariant:

- **ה- Invariant** של Worker: המצביע m_name בעל ערך תקף. כלומר, הוא בעל ערך 0, או מצביע לזיכרון שהוקצה באופן תקין.

כעת יש לדאוג לקיומו של ה- Invariant - זה יבוצע ע"י תיקון constructor המחדל, שיציב ערך 0 למצביע.

המחלקה המתוקנת:

```
class Worker
// Inariant: m_name is valid
{
        char  *m_name;
        long    m_id;
public:
        Worker()  : m_name(0) {}                        // default constructor
        Worker(const char *name, const long id);        // constructor
        Worker(const Worker&);                          // copy constructor
        ~Worker();                                      // destructor
        Worker& operator=(const Worker&);               // operator "="

        // Post-Conditions: no change to m_id
        void set(const char *name);

        // Pre-Conditions:  id > 0
        // Post-Conditions: no change to m_name,  m_id is valid
        void set(const long id);
};
```

שמירת החוזה בירושה: Liskov Sustitutability Principle

עצם ממחלקה נגזרת עלול להיות מופעל בהקשר של מחלקת הבסיס שלו באופן פולימורפי. לכן, ובכדי לקיים את עקרונות התיכון עפ"י חוזה, על המחלקה הנגזרת חלות הדרישות הבאות:

- היא אינה רשאית להוסיף תנאים ל- Pre-Conditions של הבסיס
- היא אינה רשאית להחסיר תנאים מה- Post-Conditions של הבסיס
- היא אינה רשאית להוסיף על ה- Exceptions של הבסיס

דרישות אלו מוכרות כ- LSP (Liskov Sustitutability Principle):

"If for each object o1 of type S there is an object o2 of type T, such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T."

***Barbara Liskov**, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23,5 (May, 1988)*

ניסוח פשוט יותר: אם S טיפוס נגזר של T, עקרון ההחלפה של Liskov דורש שבהתייחסות פולימורפית ניתן יהיה להחליף עצמים של T בתכנית בעצמים של S מבלי שהקוד המשתמש בהם יצטרך לעבור שינוי.

References

- Bertrand Meyer: **Error! Bookmark not defined.**, Prentice Hall, 1988.
- Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23,5 (May, 1988)
- Bertrand Meyer: *Applying "Design by Contract*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992, pages 40-51.
- B. Stroustrup: Programming with Exceptions, InformIt.com. April 2001.

כל הזכויות שמורות © מאיר סלע

מרכז ההדרכה 2000