



מיכאל שטאל

ארכיטקט בדיקות תוכנה באינטל, ישראל, עוסק בעיקר בבדיקות מערכות משובצות מחשב. במסגרת תפקידו, מיכאל מגדיר שיטות בדיקה ומתודולוגיות עבודה, עוסק בהדרכה ולפעמים אפילו מרשים לו לבדוק משהו (שזה הכי כיף). מיכאל מציג תכופות בכנסים בארץ ובח"ל ומלמד בדיקות תוכנה בפקולטה למדעי המחשב באוניברסיטה העברית. ניתן לראות חלק מהמצגות והמאמרים שלו באתר www.testprincipia.com



מגבלות (לכאורה)

אפשר להעלות לא מעט טענות למה הטכניקה של מחלקות שקילות אינה "חזקה" (כלומר, לא באמת נותנת ביטחון גבוה בנכונות הקוד). מספיק להסתכל על דוגמה ששונה במעט מהקודמת:

```
float calc_inverse (int x) {
    if (x < -10) OR (x > 10) OR (x not an Integer)
        print "Invalid input!!!"
        print "Valid input is Integer, between -10 and 10"
    Else
        print "Calculating inverse(x)"
        return (1/x)
}
```

מבנה התוכנה זהה למקרה הקודם ולכן גם הגדרת מחלקות השקילות זהה. אבל מיד רואים שהטכניקה נכשלת לגמרי: בחירה בנציג ממחלקה 2 שאינו 0, לא תזוהה את הבאג של חלוקה ב-0.

כמעט כל דבר מתגלה כמורכב יותר ממה שנראה לנו ברגע הראשון



תוכנית החלוקה

כבר מזמן הגעתי למסקנה שכמעט כל דבר מתגלה כמורכב יותר ממה שנראה לנו ברגע הראשון. הסתכלו סביבכם. קחו חפץ פשוט ויומיומי כמו לדוגמה, שקית ניילון לסנדוויץ' של הילדים. חשבתם פעם איך מצליחים לייצר את זה בלי שהצדדים ידבקו אחד לשני? או למשל איך אופים לחם מיוחד כזה שיש בו בועות אוויר גדולות? (אני עדיין לא הצלחתי!); או איך בטריה עובדת? כל דבר דורש פרטים רבים וידע רב. מצד שני, אם אנחנו מנסים ללמוד משהו, עדיף לא לרדת מיד לפרטי פרטים אלא קודם לתאר את הרעיון בקווים כללים ולתת דוגמאות פשוטות, ורק בהמשך לצלול יותר לעומק.



זו גם הגישה בקורסי בדיקות כשמלמדים את טכניקות הבדיקה הבסיסיות.

חלוקה למחלקות שקילות? פשוט! יש לנו שדה שמקבל ערכים בין 1 ל-10. מחלקת השקילות התקפה (valid) היא {1..10}, והמחלקות הלא-תקפות הן המספרים הגדולים מ-10 או הקטנים מ-1. הלאה: בדיקות ערכי גבול לאותה דוגמה: ערכי גבול תקפים הם 1 ו-10, והלא-תקפים הם 0 ו-11.

קל, פשוט... בואו נתקדם לטכניקה הבאה!

טוב – אני קצת מגזים... משקיעים יותר זמן בלימוד כל טכניקה, אבל בסופו של דבר נשארים בשלב מאוד בסיסי. התוצאה רעה משתי סיבות: קודם כל, זה מעמיד את מקצוע הבדיקות כמשהו פשוט לגמרי שכל אדם עם דופק יכול ללמוד במהירות ולבצע. הבעיה השנייה היא שהעולם האמיתי לרוב הרבה יותר מסובך מהמקרים הפשוטים המוצגים בכיתה, וכשמנסים לממש את הטכניקות על מוצר אמיתי מתקבלת ההרגשה שהתאוריה היתה נהדרת אבל במציאות היא לא ממש עובדת. בטור הפעם אתמקד בטכניקה הראשונה שמלמדים: חלוקה למחלקות שקילות, ואכנס קצת יותר לעומק – מעבר למקרים הטריוויאליים.

תזכורת: מחלקות שקילות



טכניקת הבדיקה של "חלוקה למחלקות שקילות" מבוססת על כך שניתן לחלק את מרחב הקלטות שתוכנה מקבלת לקבוצות. כל קבוצה מכילה ערכי קלט שעבורם התוכנה מריצה בדיוק את אותן שורות קוד.

בואו נכנס טיפה "לתוך הקוד" כדי להבין את העיקרון עליו מבוססת הטכניקה.

כל תוכנה ניתן לחלק לבלוקים של שורות קוד הרצות אחת אחרי השנייה (כלומר, ללא הסתעפויות כתוצאה ממשפטי תנאי). מרגע שהתוכנה נכנסה לתוך בלוק כזה, כל שורות הקוד שבבלוק יורצו אחת אחרי השנייה. למשל (פסודו-קוד):

```
float calc_square (int x) {
    if (x < -10) OR (x > 10) OR (x not an Integer)
        print "Invalid input!!!"
    else
        print "Calculating square(x)"
        return (x*x)
}
```

במקרה זה, מחלקות השקילות הן:

1. מספרים שלמים מתחת ל-10 או מעל ל-10 (מריצים את בלוק 1)
2. מספרים שלמים בין -10 ל-10 (מריצים את בלוק 2)

לאחר ביצוע החלוקה, ניקח מכל מחלקת שקילות נציג אחד כלשהו ונריץ את התוכנה עליו. התיאוריה שמאחורי הטכניקה היא שאם התוכנה עובדת נכון עבור נציג אחד של קלט מהקבוצה, יש סבירות גבוהה שהקוד ירוץ נכון גם עבור כל הקלטות האחרים שבאותה קבוצה.

האמנם כישלון? בואו נחשוב על הדרישות עבור הפונקציה calc_inverse(). הם משהו בסגנון הזה:

- עבור ערכי קלט ממשיים בין 10 ל-10, הפונקציה תחזיר את ההופכי של הקלט (1/x)
- עבור ערכי קלט ממשיים מעל 10 או מתחת ל-(-10), הפונקציה תחזיר הודעת שאינה "הכנס ערכים בין 10 ל-10"

בסקירה של הדרישות, יש סיכוי טוב שמישהו היה עולה על הבעיה וממליץ על דרישה שלישית:

- עבור קלט בערך 0, הפונקציה תחזיר הודעת שאינה "חלוקה ב-0!"

עכשיו ברור שמחלקות השקילות הן:

1. מספרים שלמים מתחת ל-10 או מעל ל-10 (מריצים את בלוק 1)
2. מספרים שלמים בין -10 ל-10 (מריצים את בלוק 2)
3. אפס (0)

מה המסקנה? נראה שהחלוקה למחלקות שקילות צריכה להתבסס על **הדרישות**, לפחות כנקודת התחלה. אבל יש מקרים שהתבססות על הדרישות בלבד אינה מספיקה. יתכן שמסיבות שונות המפתחים מממשים את הקוד בצורה שונה ממה שחשבנו, ואז החלוקה הנכונה נובעת (גם) מהקוד. למשל: פונקציה שמיינת רשימה של מספרים. על פניו, כל רשימה - ארוכה או קצרה - שייכת לאותה מחלקת שקילות. בפועל, מסיבות של מהירות ביצוע, יתכן שהמימוש ישתמש שאלגוריתם bubble sort לרשימות קצרות וב-quick sort לרשימות ארוכות.

מסתבר אם כן שהדרך הטובה להחליט אם החלוקה שלכם נכונה, היא להגדיר את המחלקות לפי הדרישות, ואז לעבור על ההגדרות והשיקולים שלכם עם המפתחים. הם כבר יאידו לכם אם הקוד עושה משהו לא צפוי.



אפשרות ג': המיקום במחרוזת אי שבו נמצאת מחרוזת ב':

- כל זוגות המחרוזות שבהן מתקיים: ב' לא נמצאת ב-א' {
 - כל זוגות המחרוזות שבהן מתקיים: ב' נמצאת ב-א', מהתו הראשון של א' {
 - כל זוגות המחרוזות שבהן מתקיים: ב' נמצאת ב-א', מתו שאינו הראשון ב-א' {
- אפשר לחלק לפי כמות הפעמים ש-ב' נמצאת ב-א'; על פי אורך המחרוזות; על פי תכולת המחרוזות (קולא רוחים, סימני פיסוק, סימנים מיוחדים); האם יש חפיפה בין הופעות חוזרות של ב' (כגון: א' = aaaaaa ו-ב' = aaa) ... וכו' וכו'...

לעיתים, מחלקות השקילות תלויות בקריטריון החלוקה שבחורים

עכשיו החלוקה נראית כמו משימה סזיפית... אבל שימו לב איך כל חלוקה מעלה רעיונות חדשים לבדיקות!

סיכום

חלוקה למחלקות שקילות אינה פשוטה כמו שנדמה בהתחלה. מה שכתבתי כאן גם הוא רק חלק מהסיפור, ואפשר להתעמק עוד? במקרים רבים תהיה יותר מחלוקה נכונה אחת. הטכניקה מאלצת אותנו לעשות ניתוח של מרחב הקלט, דבר שעוזר לקבל הבנה טובה יותר של צירופי הקלטים השונים שאיתם התוכנה צריכה להתמודד, כשכל חלוקה מייצרת מקרי בדיקה שונים ובעלי ערך. אפשר להתחיל את הניתוח תוך שימוש בדרישות, אבל חשוב לא להזניח סקירה של המסכנות עם המפתחים, על מנת לגלות מקרים שבהם המימוש יותר מסובך ממה שאפשר היה להסיק מקריאת הדרישות.

חלוקה למחלקות שקילות צריכה להתבסס על הדרישות... ואז על הקוד

כנגד הטענות שהטכניקה אינה תמיד נכונה, ראיתי מאמר¹ שטוען כי באותם מקרים בהם נראה שהטכניקה כשלה זה לא בגלל שהטכניקה אינה נכונה, אלא שהחלוקה לא היתה מעודנת מספיק. למעשה, טוענים המחברים, כל באג לוגי שנמצא – בכל טכניקה שהיא – אפשר "לתרגם" למחלקת השקילות שהייתה מוצאת אותו. כלומר, כל באג מלמד אותנו איך לחלק יותר נכון את מרחב הקלט למחלקות שקילות. חוסר ידע שלנו על המערכת והקוד גרם לכך שהחלוקה לא היתה מדוייקת מספיק ולכן לא עלתה על כל הבאגים. ועוד אנחנו מעיזים להאשים את הטכניקה שאינה חזקה במיוחד...

אם כבר הזכרנו את המאמר: עוד טענה של המחברים היא שטכניקת חלוקה למחלקות שקילות, ברוב המקרים, אינה עדיפה על בחירה רנדומלית של קלט. בנוסף, כיוון שבמקרים מסובכים יש סבירות מסוימת שנפספס את החלוקה הנכונה ונמזג כמה מחלקות שקילות יחד, המחברים ממליצים להריץ מספר דוגמאות מכל מחלקת שקילות – דוגמאות שנבחרות באקראי או בחלוקה אחידה על פני מרחב המחלקה.

העלילה מסתבכת

כשנבוא לממש את הטכניקה על מערכת אמיתית יתכן ונעמוד לפני מצבים שבהם גם ידע מלא של הדרישות והקוד לא מספיק. יש מקרים שבהם מסובך להגדיר את החלוקה, ולמעשה ישנן חלוקות שונות שכל אחת מהן ניתנת להצדקה. דוגמה: נתונה תוכנה שמוצאת אם מחרוזת אחת (מחרוזת א') מכילה בתוכה מחרוזת אחרת (מחרוזת ב').

```
findstring stringA stringB
```

על כל הימצאות של ב' ב-א', הפונקציה מדפיסה "נמצאה המחרוזת!". למשל:

```
findstring ababa ab
```

```
String found!
```

```
String found!
```

מה החלוקה למחלקות שקילות? הניחו לרגע את הגליון, קחו דף ועט ונסו להגדיר את מחלקות השקילות.

חזרתם?

ובכן, מסתבר שאין תשובה אחת נכונה והמחלקות תלויות בקריטריון החלוקה שבחורים! הנה כמה אפשרויות:

אפשרות א': חלוקה לפי קלט חוקי \ לא חוקי:

מחלקות תקפות:

- { שתי מחרוזות שבכל אחת יש תו אחד או יותר }
- מחלקות לא תקפות:

- { קלט של מחרוזת אחת }
- { קלט של יותר משתי מחרוזות }

אפשרות ב': חלוקה לפי אורך המחרוזות

- { אורך מחרוזת א' קטן מאורך מחרוזת ב' }
- { אורך המחרוזות שווה }
- { אורך מחרוזת א' גדול מאורך מחרוזת ב' }

