

חלק 4 בדיקות (Testing)

איך יודעים שמודול או תוכנית נכונים?

- אימות: תהליך שמיועד לוודא באופן פורמאלי או לא פורמאלי נכונות של מודול או תוכנית ביחס לחוזה
- אימות פורמאלי אוטומאטי אינו אפשרי במקרה הכללי (לא כריע). למרות זאת קיימים כלים פורמליים שלעיתים אינם מצליחים.
- אימות פורמאלי ידני יקר מדי לרוב המערכות פרט אולי למערכות שחיי אדם תלויים בהן ישירות (רפואיות, מוטסות, וכולי, אבל גם שם יש פחות אימות ממה שהיה ראוי)
- בדיקות (testing): ביצוע סדרת הרצות של התוכנה שמיועדות למצוא פגמים, אם יש, ולהגדיל את בטחוננו בנכונותה
- לא מבטיח נכונות, אבל יותר טוב מכלום, ומועיל מאוד באופן מעשי להקטנת מספר הפגמים

מינוח שמסקף גישה בריאה לחיים

- כאשר המכונית לא עוברת טסט, זה כמובן מעצבן, אבל זה בדרך כלל לא כישלון של מכון הרישוי שביצע את הטסט
- כישלון והצלחה של בדיקה הם נפרדים לחלוטין מאלה של הקוד הנבדק!
- בדיקה מצליחה אם היא מגלה פגם
- בדיקה נכשלת אם היא לא מגלה פגם או מדווחת על פגם לא קיים
- אם בדיקה מדווחת על פגם נאמר שהקוד לא עבר את הבדיקה, ולא נאמר שהבדיקה נכשלה
- דווח על פגם הוא אירוע חיובי (לא משמח אולי, אבל חיובי) כי הוא מספק אפשרות לתיקון פגם לפני שהוא גורם עוד נזק

שלושה סוגי בדיקות

- בדיקות יחידה (unit tests) בודקות מודול בודד (בדרך כלל מחלקה אחת או מספר מחלקות קשורות)
- בדיקות אינטגרציה בודקות את התוכנית כולה, או קבוצה של מודולים ביחד; מתבצעת תמיד לאחר בדיקות היחידה של המודולים הבודדים (כלומר על מודולים שעברו את בדיקות היחידה שלהם)
- בדיקות קבלה (acceptance tests) מתבצעות על ידי הלקוח או על ידי צוות שמתפקד בתור לקוח, לא על ידי צוות הפיתוח
- גם לאחר כניסה לשימוש, התוכנה ממשיכה למעשה להיבדק, אבל אצל משתמשים אמיתיים; רצוי שיהיה מנגנון דיווח לתקלות ופגמים שמתגלים בשלב הזה, ורצוי לתקן את הפגמים הללו

קופסאות שחורות וקופסאות פתוחות

- על כל מודול תוכנה צריך לבצע שני סוגים של בדיקות יחידה
- בדיקות קופסה שחורה (black-box tests) בודקים את הקוד מול החוזה שהוא מבטיח לקיים, והן אינן תלויות במימוש
- בדיקות כיסוי (coverage tests או glass-box tests) דואגות שבזמן הבדיקות, כל פיסת קוד תרוץ, ובמקרים מסוימים, תרוץ יותר בכמה צורות
- בדיקות קופסה שחורה לא תלויות במימוש ולכן אותו סט בדיקות יישאר תקף גם אם נשנה בעתיד את המימוש, או נוסיף מימוש חלופי.
- בדיקות כיסוי צריך לעדכן כאשר מעדכנים את הקוד

מה בודקים בקופסה שחורה?

- את החוזה
- עבור כל שירות, מביאים את תוכנית הבדיקה למצב שבו היא מקיימת את תנאי הקדם, קוראים לשירות, ובודקים שתנאי האחר מתקיים
- לפעמים יש יותר מדרך אחת לקיים את תנאי הקדם; אז צריך לבדוק דרכים שונות
- ברור שלפעמים יש מספר עצום של דרכים לקיים את תנאי הקדם ואי אפשר לבדוק את כולן; צריך לבדוק דרכי קיום שונות של פסוקי "או" וצריך לבדוק מקרי קצה

קופסה שחורה (המשך)

- למשל, עבור תנאי הקדם $0 \leq i \leq \text{length}()$ צריך לבדוק את המקרה $i = \text{length}() = 0$, $i = 0 < \text{length}()$, וגם מקרה אחד לפחות שבו $i = 2$, $0 < i = \text{length}()$
- מקרי הקצה ($i = 0$ ו- $i = \text{length}()$ בדוגמה) מסייעים למצוא מקרים שבהם שכחנו לממש טיפול במקרי קצה (למשל שכחנו לטפל באופן נפרד במקרה שבו שדה מכיל null וכו')
- מקרים נוספים: מחרוזות ריקות וקצרות, מערכים ריקים, שני ארגומנטים או יותר שמתייחסים לאותו עצם או מערך
- אם השירות יכול לקיים אחד מתוך כמה תנאי אחר (למשל, "יוחזר מספר הגרסאות או שנודיע על חריג קלט/פלט") אז דרושות בדיקות שונות שגורמות לו לקיים כל אחד מהם

אבל האם החוזה "נכון"?

- אמרנו שהבדיקות בודקות את התנהגות הקוד מול החוזה
- אם בדיקה נכשלת, יתכן שהקוד פגום, אבל יתכן גם שהדרישה בחוזה חזקה מדי
- איך יודעים האם לתקן את הקוד או את החוזה?
- כמובן שאם עבדנו בצורה מסודרת וחשבנו על החוזה היטב (ואולי גם השתמשנו בו להוכחת נכונות של לקוחות), רוב הסיכויים שהבעיה היא במימוש
- אבל ככל שמטפסים ממחלקות בודדות לתתי מערכות שלמות ובסוף לתוכנית השלמה (מבדיקות יחידה לבדיקות אינטגרציה), הסיכוי שהחוזה פגום עולה, מכיוון שמחלקות ברמות נמוכות הן יותר סטנדרטיות ומכיוון שקשה יותר לאפיין נכון את ההתנהגות הנכונה של מערכות מורכבות

למה בדיקות קופסה שחורה לא מספיקות?

- כי אי אפשר לבדוק באופן ממצה את כל הדרכים לקיים את תנאי הקדם; מספר הדרכים עצום או אינסופי ברוב המקרים

```
public void someMethod(int depth, ...) {  
    if (depth == 23478)  
        System.out.println("xxx");  
}
```

- ברור שבדיקת קופסה שחורה לא תמצא את ההתנהגות הזו
- נראה דמיוני אבל זה לא; תוכניתן השתמש בקטע הקוד הזה כדי לקבוע במנפה (debugger) נקודת עצירה שלא מופעלת בכל הפעלה של השירות
- דוגמאות אחרות: מימוש מסוים של מיון למערכים קטנים, מימוש אחר לגדולים; פסוק `if` בתוך השירות בוחר את המימוש

בדיקות כיסוי

- בדיקות שמיועדות לגרום לכל פיסת קוד לרוץ
- בכל פסוק תנאי צריך לקבל את שתי התוצאות האפשריות (then/else)
- לולאות צריך לבצע אף פעם, פעם אחת, ושתי פעמים (אף פעם למקרה שהכנסנו לגוף הלולאה קוד שצריך להתבצע בכל מקרה, שתי פעמים למקרה שהמעבר בין איטרציות פגום)
- אם יש מספר דרכים לצאת מלולאה, צריך להשתמש בכולן
- כנ"ל לתנאים בוליאניים עם "או" (לבדוק את כל האפשרויות)
- רקורסיה דינה כדין לולאה: אף פעם, פעם אחת, שתי פעמים
- יש כלים אוטומטיים שמודדים כיסוי ומצביעים על קוד לא מכוסה

תוצאי לוואי רצויים ולא רצויים

- בדיקות הקופסה השחורה צריכות גם לבדוק תוצאי לוואי רצויים שמצוינים בתנאי האחר של השרות
- בדרך כלל בתנאי האחר יש גם פסוק סתום שאיננו מופיע מפורשות: "ופרט לכך אין לשירות תוצאי לוואי"
- זה לא תמיד פשוט לתחם את תוצאי הלוואי המותרים, כי ככלל אמרנו שמותר לשירות לקיים יותר ממה שהוא מבטיח
- (יש סגנון לכתובת חוזים שבו משתמשים בפסוק modifies שמתאר איזה עצמים מותר לשירות לשנות; את השאר אסור לו; זהו תיחום יותר מדויק ויותר מפורש של תוצאי הלוואי)
- קשה לבדוק שאין תוצאי לוואי פרט למותרים; היכן לחפש?
- ובכל זאת, לפעמים רצוי לחשוד ולבדוק

היכן לחפש דליפה של תוצאי לוואי

- יש חשודים רגילים: עצמים ומחלקות שיש נטייה לשנות אותם בהמון מקרים, למשל מנגנונים של הקצאת זיכרון ומשאבים אחרים (קבצים פתוחים, חלונות על המסך); (בהמשך נראה מנגנוני הקצאת זיכרון שאפשר לבנות בג'אווה).
- ויש חשודים ששמם עולה בחקירה: סריקה של הקוד תראה את מי הוא עשוי לשנות
- לגבי החשודים הללו, צריך לבדוק שמצבם לא משתנה בדרכים שאסור לו להשתנות
- הבדיקה של חשודים רגילים (הקצאת זיכרון) היא חלק מבדיקות הקופסה השחורה
- הבדיקה של חשודים שנמצאו בחקירה היא כמובן חלק מבדיקות הכיסוי

בדיקות של היררכיית טיפוסים

- (נדבר על זה אחרי שנלמד על היררכית טיפוסים).
- לגבי מחלקות שממשות מנשק: בדיקת קופסה שחורה מול החוזה של המנשק (אם לא חיזקנו אותו) וכיסוי של המימוש
- לגבי מחלקות שמרחיבות מחלקות: בדיקה מלאה של מחלקת הבסיס, בדיקה של השירותים הנוספים/מחוזקים של המרחיבה, ובדיקת כיסוי של המרחיבה
- מחלקה מופשטת צריך לבדוק בעזרת מחלקה מוחשית מרחיבה

איך בודקים?

- בבדיקות מעורבים שני סוגי קוד: מנועים ורכיבים חלופיים
- מנוע (driver) הוא קוד שמדמה לקוח של המודול הנבדק וקורא לו
- רכיב חלופי (stub) מחליף ספק שמשרת את המודול הנבדק
- למשל מחלקה A משתמשת ב-B שמשתמשת ב-C
- בדיקת יחידה ל-B תדמה לקוח של B ותספק מחלקה חלופית ל-C, על מנת שניתן יהיה לבדוק את B בנפרד מ-A ו-C
- רכיב חלופי צריך להיות פשוט ככל האפשר
- לפעמים הרכיב החלופי לא יכול להיות משמעותית יותר פשוט מהמודול שאותו הוא מחליף, ואז כדאי להשתמש במודול האמיתי לאחר בדיקות יסודיות שלו

מתי הרכיב האמיתי פשוט כמו רכיב חלופי?

- מקרה אחד: כאשר הרכיב האמיתי פשוט מאוד, וקשה למצוא חלופה יותר פשוטה.
- מקרה שני: כאשר הרכיב האמיתי מקיים חוזה מורכב או חוזה שקשה לקיים בדרך פשוטה. למרות זאת כדאי לחשוב היטב האם אפשר בכל זאת לממש רכיב חלופי פשוט.
- לפעמים קשה לממש רכיב חלופי שיכול להחליף מודול אמיתי בכל מצב, אבל לא קשה לממש רכיב חלופי מוגבל מאוד שפועל לפי החוזה במקרים הספיציפיים שמתעוררים בבדיקה. זה דורש תיאום בין המנוע ובין הרכיבים החלופיים של תוכנית הבדיקה.

למה צריך לבדוק באופן יסודי מודול שממשם בבדיקה של מודול אחר?

- כדי לדעת בקלות היכן לחפש את הפגם אם בדיקה מוצאת פגם. אם בודקים ביחד שני מודולים 'א' ו-'ב', שאין לנו ביטחון בנכונות של אף אחד מהם, קשה לדעת האם פגם שנחשף בבדיקה הוא פגם במודול 'א' או במודול 'ב', וצריך לחפש בשניהם.

דמי ביטחון

- חוזה מורכב דורש הרבה בדיקות קופסה שחורה
- בפרט, אם החוזה מבטיח ששירות יפעל בכל מקרה, אבל במקרים שונים יקיים תנאי אחר שונים (למשל את תנאי האחר הרצוי ללקוח במקרים מסוימים והודעה על חריג במקרים אחרים), אז צריך לבדוק את כל האפשרויות הללו
- לתכנות דפנסיבי יש מחיר: יותר בדיקות קופסה שחורה
- מימוש מורכב דורש הרבה בדיקות כיסוי, בין אם המורכבות היא תוצאה של חוזה מורכב או של שאיפה ליעילות
- אם הקוד עצמו לא בודק את החוזה (קשה לבדוק בג'אווה בלי כלי עזר), כדאי אולי לבדוק את תנאי הקדם ברכיבים חלופיים; זה מגביר את הביטחון שהלקוחות מקיימים את תנאי הקדם; את תנאי האחר בודק המנוע

עיקרון הזריזות

- רצוי למצוא פגם בתוכנה קרוב ביותר לנקודה שבה נוצר הפגם
- זה נכון לגבי זמן הריצה: כדאי שהתוכנה תגלה את הפגם ותדווח עליו (למשל על ידי בדיקת החוזים והמשתמרים) קרוב ביותר לנקודה שבה הקוד הפגום פעל; זה יקל על מציאת הפגם בחיפוש אחורה מנקודת הדיווח על הפגם
- נכון גם לזמן הפיתוח: כדאי לגלות את הפגם מהר ככל האפשר לאחר שנוצר (פגמים הם תוצרי יצירתיות המפתחים, לא תכונה מובנית של תוכנה); זה יקל ויוזיל את תיקונו
- לכן רצוי לממש בדיקות יחידה מוקדם ככל האפשר; בדיקות קופסה שחורה אפשר ורצוי לממש לפני המימוש של המודול, ובדיקות כיסוי רצוי לממש מייד לאחר המימוש; לא כדאי להתעכב

בדיקות רגרסיה

- כל פעם שמגלים פגם בתוכנה, בכל שלב בחיי התוכנה (גם לאחר שנכנסה לשימוש) יש להוסיף בדיקה שחושפת את הפגם: נכשלת בגרסה עם הפגם אך עוברת בגרסה המתוקנת
- לפעמים הבדיקה תתווסף לבדיקות הקופסה השחורה ולפעמים לבדיקות הכיסוי (אם הפגם קשור באופן הדוק למימוש ולא לחוזה)
- את סט הבדיקות השלם, כולל כל הבדיקות הללו שנוצרו בעקבות גילוי פגמים, מריצים לאחר כל שינוי במודול הרלוונטי, על מנת לוודא שהשינוי לא גרם לרגרסיה, כלומר להופעה מחודשת של פגמים ישנים
- סט הבדיקות מייצג, כמו התוכנה המתוקנת, ניסיון מצטבר ויש לו ערך טכני וכלכלי משמעותי

מתי להפסיק להשתמש בבדיקה

- ככלל לעולם לא
- כאמור, לסט הבדיקות יש ערך רב ואין טעם, בדרך כלל, להפסיק להשתמש בבדיקה שעשויה לגלות פגמים
- עם זאת, יש להשתמש בהגיון בריא
- אין טעם להשתמש בבדיקת רגרסיה שבדקה פגם שהיה קשור באופן הדוק למימוש אם החלפנו לחלוטין את המימוש
- אין טעם להמשיך להשתמש בסט עצום של בדיקות אם בשלב מסוים עוברים לבדיקות כיסוי מקיפות (מחקרים מצאו מקרים שבהם סט עצום של בדיקות לא כיסה את כל הקוד, שניתן היה להשתמש בחלקיק מתוכן בלי להוריד בהרבה את הכיסוי, ושניתן היה להשלים את הכיסוי במספר קטן של בדיקות נוספות; וזה לתוכנה עם סט בדיקות טוב!)

דוגמה לבדיקה מוזרה אבל מוצדקת

- נניח שמחלקה מסוימת משתמשת בפונקצית ספרייה, בצורה נכונה לחלוטין, כלומר תוך שימוש נכון בחוזה של פונקצית הספרייה
- לקוח מדווח על פגם בתוכנה, ואחרי בירור מסתבר שהבעיה היא שבגרסה של הספרייה הסטנדרטית שמותקנת אצל הלקוח (נניח JDK 1.4.1) יש פגם בפונקצית הספרייה (למשל באג מספר 4302884 במחלקה `java.applet.AudioClip` או באגים בספרייה שמשפיעים רק על גרסאות מסוימות של מערכת ההפעלה)
- נוסיף בבדיקה שמדמה את הפגם בספרייה הסטנדרטית ונתקן את המחלקה שלנו; למי שלא היה מודע לפגם המדווח הקוד יראה כעת משונה, והבדיקה תיראה מוזרה, אבל שניהם מוצדקים והבדיקה הזו עשויה למנוע חזרת הפגם

בדיקות צריכות להיות אוטומטיות

- בדיקה שדורשת התערבות של אדם היא בדיקה לא טובה, כי קשה ויקר לחזור עליה אחרי כל שינוי בתוכנה
- לכן,
- כל בדיקה בדידה צריכה להיות אוטומטית
- וצריך מנגנון (תוכנה) שמריץ את כל הבדיקות ומדווח על כל הפגמים שהתגלו
- לפעמים צריך להריץ אולי רק חלק, למשל אם ביצענו שינוי קטן בתוכנה; אבל אם הבדיקות מהירות כדאי להריץ את כולן

איך נמנעים מהתערבות אנושית

- אם התוכנה מופעלת על ידי מנשק אדם-מכונה (למשל מנשק משתמש גרפי, או על ידי דיבור, וכדומה), כדאי לבנות לה גם מנשק חלופי, לצורך בדיקות בלבד, שיפעיל ויבדוק את החלק הפונקציונאלי
- ובמקרים כאלה את המנשק למשתמש נבדוק לחוד
- יש גם כלים מיוחדים לבדיקת מנשקים גרפיים: כלים שאפשר לתכנת בהם תנועה של עכבר, הקשה על המקלדת, וכדומה
- לפעמים בונים חומרה מיוחדת שתפקידה להפעיל את המנשק למשתמש בזמן בדיקות, כמו זרוע רובוטית שלוחצת על reset
- במקרה חרום אפשר להסתמך על בודקים אנושיים שיפעילו את המערכת על פי הנחיות כתובות, אבל זה פחות אמין, זה יקר, וזה משעמם

פיתוח מונחה בדיקות

מתודולוגיה ששמה דגש על הבדיקות כגורם המניע את התהליך.

חוזרים שוב ושוב על התהליך הבא:

- הוסף במהירות בדיקה.
- הרץ את כל הבדיקות וראה שהחדשה לא עוברת.
- בצע שינוי קטן בקוד.
- הרץ את כל הבדיקות וראה שכולם עוברות.
- בצע refactoring (מושג זה יוסבר בהמשך) לביטול כפילות בקוד.

Kent Beck, Test-Driven Development By example,
Addison-Wesley

כלי עזר לבדיקות יחידה JUnit

- JUnit היא תוכנה חופשית לביצוע בדיקות יחידה בג'אווה.
- נכתב ע"י אריך גאמה וקנט בק.
- נועד בעיקר לפיתוח מונחה בדיקות, אך לא מחייב את אימוץ השיטה באופן מלא.
- מיועד לבדיקות שמבצע המפתח.
- הבדיקות מריצות את עצמן ומדווחות על התוצאות מייד.
- הכלי נתמך ע"י אקליפס (אבל גם בסביבות אחרות)
- בתרגול תלמדו להשתמש ב JUnit

סיכום נושא הבדיקות

- בדיקות יחידה, אינטגרציה, קבלה, וקבלת דיווחים מהשטח
- בדיקות קופסה שחורה לעומת בדיקות כיסוי
- חיפוש ממוקד של תוצאי לוואי אסורים
- בדיקות יש לבצע מוקדם לאחר המימוש (אפשר לממש את בדיקות הקופסה השחורה לפני מימוש המודול) ולעיתים תכופות בהמשך (תכופות לעומת קצב השינויים בקוד)
- מיכון הבדיקות מקל על ביצוען התכוף
- בדיקות מצטברות ושימוש מתמשך בהן מונע רגרסיה של קוד
- חוזים סבוכים וקוד מורכב מובילים לעלות בדיקה גדולה