חלק 11
בדיקות
(Testing)

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

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

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 תקף לכל המימושים של מנשק מסוים, גם העתידיים, ובפרט לשינויים ותיקונים במימוש הנוכחי
- בדיקות כיסוי צריך לעדכן כאשר מעדכנים את הקוד

מינוח שמשקף 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

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<=i<=length()` צריך לבדוק את המקרה `i=0`, `i=length()`, `0<i<length()`, `i=0<length()`, וגם מקרה אחד לפחות שבו `i=2`
- מקרי הקצה (`i=0` ו-`i=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 הבדיקה צריכה לגרום לו לקיים כל אחד מהם

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

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

תוצאי לוואי רצויים ולא רצויים

- בדיקות הקופסה השחורה צריכות גם לבדוק תוצאי לוואי רצויים שמצוינים בתנאי האחר
- 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 לכתיבת 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 לשירות תוצאי לוואי פרט למותרים; היכן לחפש?
- ובכל זאת, לפעמים רצוי לחשוד ולבדוק

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

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); ראינו שגם בג'אווה אפשר לבנות מנגנוני הקצאת זיכרון
- 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

- 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

למה צריך לבדוק באופן יסודי מודול שמשמש 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.

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

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 (כפי שאמרנו קשה לבדוק בג׳אווה), כדאי אולי לבדוק את תנאי הקדם ברכיבים חלופיים; זה מגביר את הביטחון שהלקוחות מקיימים את תנאי הקדם; את תנאי האחר בודק המנוע

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. אבל התנאי השני מתקיים לעיתים רחוקות וכדאי 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 הספיציפים שמתעוררים בבדיקה. זה דורש תיאום בין המנוע ובין הרכיבים החלופיים של תוכנית הבדיקה.

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
- וזה נכון לזמן הפיתוח: כדאי שנגלה את הפגם מהר ככל האפשר לאחר שיצרנו אותו (פגמים הם תוצרי היצירתיות של מפתחים, לא תכונה מובנית של תוכנית או תוכנה בכלל); 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

- בכל פעם שמגלים פגם בתוכנה, בכל שלב של חיי התוכנה (גם לאחר שנכנסה לשימוש) יש להוסיף בדיקה שחושפת את הפגם, כלומר שנכשלת בגרסה עם הפגם אבל 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

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

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!)

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

דוגמה לבדיקה מוזרה אבל מוצדקת

- נניח שמחלקה מסוימת משתמשת בפונקצית ספריה, 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 בפונצית הספריה (למשל באג מספר 4302884 במחלקה `java.applet.AudioClip`) או באגים בספריה שמשפיעים רק על גרסאות מסוימות של מערכת ההפעלה)
- נוסיף בדיקה שמדמה את הפגם בספריה 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

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 לביטול כפילות בקוד.

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

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

מפתחים רק מבצעים את התיכון והעיצוב

- מהנדס מנשק האנוש מחליט איך המנשק יתנהג, המעצב מחליט איך בדיוק הוא יראה (ישמע, יורגש)
- תוכניתנים מממשים את המנשק הגרפי בהתאם

חלק 12
מנשקי אדם־מכונה
(גראפיים)

שקף אחד על הנדסת מנשקי אנוש

- קונסיסטנטיות; המנשק צריך להתנהג בהתאם לציפיות המוקדמות של המשתמש/ת; פעולות אוטומטיות (גזור-הדבק, למשל), המראה של פריטים (צלמיות, למשל), המראה וההתנהגות הכללית של התוכנית, של הפלטפורמה
- המשתמש/ת בשליטה, לא המחשב; חזרה אחורה באשף, ידיעה מה המצב הנוכחי של התוכנית ומה היא עושה כרגע
- יעילות של המשתמש, לא של המחשב; חומרה היא זולה, משכורות הן יקרות, ואכזבות הן עוד יותר יקרות
- התאמה לתכיפות השימוש וללימוד התוכנה; האם משתמשים בה באופן חד פעמי (אשף לכתיבת צוואות) או יומיומי (דואל); גם משתמש יומיומי בתוכנה היה פעם מתחיל חסר ניסיון
- פעולה ישירה על ייצוג נראה של עצמים, ללא שיום (בד"כ)

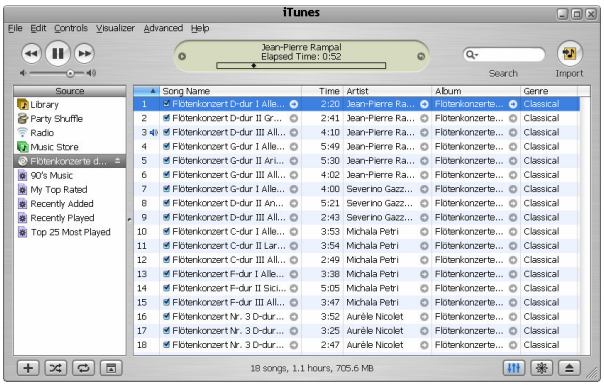
ענווה

- לנו כמפתחי תוכנה יש רק חלק קטן בפיתוח מנשקים גרפיים
- פיתוח מנשק גרפי מתחיל במהנדס מנשקי אנוש: איש/אשת מקצוע שיודעים לפתח מנשק שיהיה מובן, יעיל, ונעים
- מהנדס מנשקי האנוש יודע גם למדוד את איכות המנשק על קבוצות משתמשים ולתקן את המנשק בהתאם; גם כאן בדיקות הן מרכיב חשוב, אבל דרך ביצוען שונה לגמרי
- פיתוח המנשק ממשיך במעצב/ת גראפי/ת; עיצוב גרוע או סידור גרוע של האלמנטים על המסך מקשים על הבנת המנשק ועל השימוש בו
- מהנדס המנשקים והמעצב דואגים שגם משתמשים עם מוגבלויות (בעיקר ליקויי ראיה) יוכלו להשתמש בתוכנה
- אם יש מרכיבי קול או מישוש (טעם וריח?), צריך לעצב אותם

שקף אחד על עיצוב גראפי (של מנשקים)

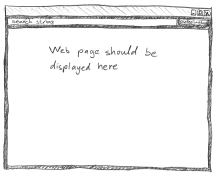
- קונסיסטנטיות
- קונטרסט להדגשת מה שבאמת דרוש הדגשה; עומס ויזואלי מפחית את הקונטרסט
- ארגון ברור של המסך (בדרך כלל תוך שימוש בסריג)
- כיוון וסדר ברורים לסריקת המידע (מלמעלה למטה משמאל לימין, או ימין לשמאל)
- העיצוב הגרפי של מנשק של תוכנית בדרך כלל אינו מוחלט; המשתמש ו/או הפלטפורמה עשויים להשפיע על בחירת גופנים ועל הסגנון של פריטים גראפיים (כפתורים, תפריטים); העיצוב צריך להתאים את עצמו לסביבה

ועכשיו, למימוש



מה הדפדפן אמור לעשות

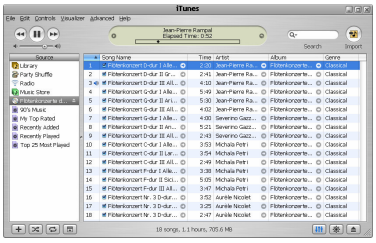
- המשתמשת תקליד מחרוזת חיפוש בשדה בצד שמאל למעלה
- לחיצה על הכפתור I'm feeling lucky מימין לשדה הטקסט תשלח את מחרוזת החיפוש ל-Google
- כאשר תתקבל התשובה, הדפדפן ישלוף מהתשובה של Google את הכתובת (URL) הראשונה ויטען אותה לרכיב הצגת ה-HTML בתחתית המסך, וכן ישנה את כותרת החלון כך שתציג את ה-URL



- נממש את הדפדפן בעזרת ספרייה למימוש מנשקים גראפיים בשם SWT (Standard Widget Toolkit)
- ספריות אחרות למימוש מנשקים גראפיים בג'אווה הן AWT ו-Swing

שלושת הצירים של תוכנה גראפית

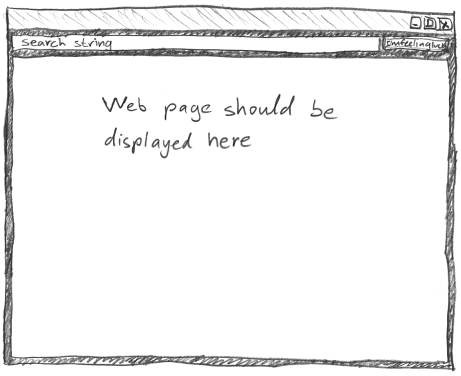
- אלמנטים מסוגים שונים על המסך (היררכיה של טיפוסים)
- הארגון הדו־מימדי של האלמנטים, בדרך כלל בעזרת מיכלים
- ההתנהגות הדינמית של האלמנטים בתגובה לפעולות של המשתמש/ת (הקלדה, הקלקה, גרירה)



מבנה המימוש

```
public class GoogleBrowser {
  private Shell   shell   = null;
  private Button  button  = null;
  private Text    text    = null;
  private Browser browser = null;
  public static void main(String[] args) {
    call createShell and run event loop }
  private void createShell() { create the GUI }
  private static String search(String q) {
    send query to Google and return the first URL }
}
```

דוגמה ראשונה: דפדפן זעיר מגולגל



Widgets

- השדות `text`, `button`, `browser` ו-`shell` יתייחסו לרכיבי המנשק הגראפי; רכיבים כאלה נקראים widgets
- מעטפת (shell) הוא חלון עצמאי שמערכת ההפעלה מציגה, ושאינו מוכל בתוך חלון אחר; החלון הראשי של תוכנית הוא מעטפת, וגם דיאלוגים (אשף, דיאלוג לבחירת קובץ או גופן, וכדומה) הם מעטפות
- עצם המעטפת בג'אווה מייצג משאב של מערכת ההפעלה
- הרכיבים האחרים הם אלמנטים שמוצגים בתוך מעטפת, כמו כפתורים, תפריטים, וכדומה; חלקם פשוטים וחלקם מורכבים מאוד (כמו Browser, רכיב להצגת HTML)
- לפעמים הם עצמים שממופים לבקרים שמערכת ההפעלה מציגה בעצמה (controls), ולפעמים הם עצמי ג'אווה טהורים

הלולאה הראשית

```
public static void main(String[] args) {
  Display display = Display.getDefault();
  GoogleBrowser app
    = new GoogleBrowser();
  app.createShell();
  while (!app.shell.isDisposed()) {
    if (!display.readAndDispatch())
      display.sleep();
  }
  display.dispose();
}
```

בניית רכיבי מנשק

- בנאי שבונה רכיב מנשק מקבל בדרך כלל שני ארגומנטים: ההורה של רכיב המנשק בהיררכיית ההכלה, והסגנון של רכיב המנשק
- כאשר בנינו את שדה הטקסט, העברנו לבנאי את הארגומנטים shell (ההורה) ו-SWT.BORDER (סיבית סגנון)
- למעטפת אין הורה (אבל יכלו להיות לה סיביות סגנון)
- את תכונות ההורות והסגנון אי אפשר לשנות לאחר שהרכיב נבנה
- רכיבים שונים משתמשים בסיביות סגנון שונות; למשל, למעטפת יכולה להיות או לא להיות מסגרת עם כפתורי סגירה ומיזעור (המסגרת נקראת trim), אבל לרכיב פנימי אי אפשר לבחור סגנון שכולל מסגרת כזו

יצירת המנשק הגראפי

```
private void createShell() {
  shell = new Shell();
  shell.setText("Browser Example");
  shell.setLayout(new  layout manager: a grid with
    GridLayout(2, false));    2 unequal columns
  text = new Text(shell, SWT.BORDER);
  text.setLayoutData(new
    GridData(SWT.FILL,       horizontal alignment
             SWT.CENTER,     vertical alignment
             true,           grab horizontal space
             false));        don't grab vertical space
```

המשך יצירת המנשק

```
button = new Button(shell, SWT.NONE);
button.setText("I'm feeling lucky");
button.setLayoutData(new
  GridData(SWT.RIGHT, SWT.CENTER,
           false, false));
browser = new Browser(shell, SWT.NONE);
browser.setLayoutData(new
   GridData(SWT.FILL, SWT.FILL,    fill both ways
            false,
            true,          row grabs vertical space
            2, 1));        widget spans 2 columns
```

פריסת הרכיבי המנשק במעטפת

- מעטפות הם רכיבי מנשק שמיועדים להכיל רכיבי מנשק
- את הרכיבים המוכלים צריך למקם; רצוי לא למקם אותם באופן אבסולוטי (ערכי x ו-y בקואורדינטות של הרכיב המכיל)
- מנהלי פריסה (layout managers) מחשבים את הפריסה על פי הוראות פריסה שמצורפות לכל רכיב מוכל
- GridLayout הוא מנהל פריסה שממקם רכיבים בתאים של טבלה דו-מימדית; רכיבים יכולים לתפוס תא אחד או יותר
- רוחב עמודה/שורה נקבע אוטומטית ע"פ הרכיב הגדול ביותר
- GridData הוא עצם שמייצג הוראות פריסה עבור GridLayout; כאן ביקשנו מתיחה אופקית של הרכיב עצמו בתוך העמודה ושל העמודה כולה

הפרוצדורה שהכפתור מפעיל

```
button.addSelectionListener(
  new SelectionAdapter() {
    public void
    widgetSelected(SelectionEvent e) {
      String query = text.getText();
      String url = search(query);
      shell.setText(url);
      browser.setUrl(url);
    }
  });
```

אירועים והטיפול בהם

- מערכת ההפעלה מודיעה לתוכנית על אירועים: הקשות על המקלדת, הזזת עכבר והקלקה, בחירת אלמנטים, ועוד
- ההודעה מתקבלת על ידי עצם יחיד (singleton) מהמחלקה `Display`, שמייצג את מערכת ההפעלה (מע' החלונות)
- קבלת אירוע מעירה את התוכנית מהשינה ב-`sleep`
- כאשר קוראים ל-`readAndDispatch`, ה-`display` מברר לאיזה רכיב צריך להודיע על האירוע, ומודיע לו
- הרכיב מפעיל את העצמים מהטיפוס המתאים לסוג האירוע שנרשמו להפעלה על ידי קריאה ל-`add*Listener`

Listener לעומת Adapter

- לכפתור הוספנו מאזין ספיציפי ממחלקה אנונימית שמרחיבה את `SelectionAdapter`
- `SelectionAdapter` היא מחלקה שממממשת את המנשק `SelectionListener` שמגדיר שני שירותים
- ב-`SelectionAdapter`, שני השירותים אינם עושים כלום
- הרחבה שלה מאפשרת להגדיר רק את השירות שרוצים, על פי סוג האירוע הספיציפי שרוצים לטפל בו; ארועים אחרים יטופלו על ידי שירות שלא עושה כלום
- אם המחלקה האנונימית הייתה מממשת ישירות את `SelectionListener`, היא הייתה צריכה להגדיר את שני השירותים, כאשר אחד מהם מוגדר ריק; מסורבל

שלוש גישות לטיפול באירועים

- בעזרת טיפוסים סטאטיים ספיציפיים לסוג האירוע; למשל, `KeyListener` הוא מנשק שמגדיר שני שירותים, `KeyPressed` ו-`KeyReleased`, שכל אחד מהם מקבל את הדיווח על האירוע בעזרת עצם מטיפוס `KeyEvent`
- ללא טיפוסים סטאטיים שמתאימים לאירועים ספיציפיים; האירוע מפעיל עצם מטיפוס `Listener` שמממש שירות בודד, `handleEvent`, והאירוע מדווח בעזרת טיפוס `Event`; יותר יעיל, פחות בטוח
- יש ספריות של מנשקים גראפיים, למשל AWT, שמשתמשות בירושה: המחלקה שמייצגת את המנשק שלנו מרחיבה את `Frame` (מקביל ל-`Shell`) ודורסת את השירות `handleEvent`, ש-`Frame` קוראת לו לטיפול באירועים

כמעט סיימנו

- נותרו רק שתי שורות שלא ראינו ב-`createShell`,

```
    button.addSelectionListener(...);
    shell.pack();          causes the layout manager
                                to lay out the shell
    shell.open();          opens the shell on the screen
}
```

- והפרוצדורה שמחפשת במנוע החיפוש Google ומחזירה את ה-URL של התשובה הראשונה

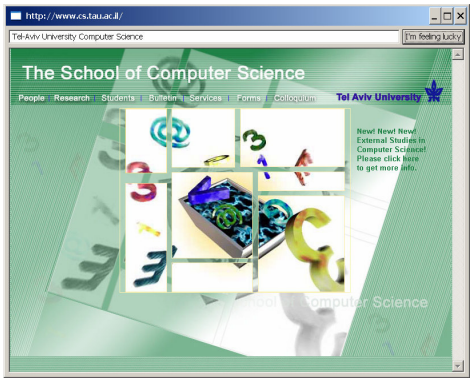
דוגמה לשימוש במאזין לא ספיציפי

```
button.addListener(
  SWT.Selection,          the event we want to handle
  new Listener() {
    public void handleEvent(Event e) {
      String query = text.getText();
      String url = search(query);
      shell.setText(url);
      browser.setUrl(url);
    }
  });
```

חיפוש ב-Google (לא ממש רלוונטי)

```
private static String search(String q) {
  GoogleSearch s = new GoogleSearch();
  s.setKey         (" my secrete key");
  s.setProxyHost   ("proxy.tau.ac.il");
  s.setProxyPort   (8080);
  s.setQueryString(q);
  s.setStartResult(0);
  try {
   GoogleSearchResult r = s.doSearch();
   return
    (r.getResultElements())[0].getURL()
```

והתוצאה,



Look and Feel

- מערכות הפעלה עם מנשק גראפי מספקות שירותי מנשק (למשל, Windows ו-MacOS; אבל לא לינוקס ויוניקס)
- שימוש במנשקים של מערכת ההפעלה תורם למראה אחיד ולקונסיסטנטיות עם ציפיות המשתמש ועם קביעת התצורה שלו (אם יש דרך לשלוט על מראה הרכיבים, כמו בחלונות)
- ספריות מנשקים משתמשות באחת משתי דרכים על מנת להשיג אחידות עם המנשקים של מערכת ההפעלה
- שימוש ישיר ברכיבי מנשק של מערכת ההפעלה; AWT, SWT
- אמולציה של התנהגות מערכת ההפעלה אבל כמעט ללא שימוש ברכיבי המנשק שלה (פרט לחלונות); למשל Swing, JFace, Qt; זה מאפשר להחליף מראה, & pluggable look feel

סיכום ביניים

- ראינו את המחלקות שמייצגות רכיבי מנשק גראפי
- ראינו איך נרשמים להגיב על אירוע כגון לחיצה על כפתור
- ראינו כיצד מגדירים את הפריסה של הרכיבים על המסך

- האם המנשק הגראפי של התוכנית מוצלח? לא, הכפתור מיותר, ובעצם, אפשר היה להשתמש בשדה הטקסט גם עבור חיפוש וגם עבור הקלדת URL באופן ישיר
- המחלקות שמייצגות את רכיבי המנשק מורכבות מאוד: צריך ספר או מדריך מקוון, צריך להתאמן, ורצוי להשתמש במנגנון עריכה ייעודי למנשקים גראפיים (GUI Builder)

יתרונות וחסרונות של Pluggable L&F

- מאפשר להגדיר מראות חדשים לרכיבים; שימושי עבור משחקים, עבור תוכניות שרוצים שלא יראו כמו תוכנות מחשב (בעיקר נגני מוסיקה וסרטים), ובשביל מיתוג (branding)
- מאפשר לבנות יישומים עם מראה אחיד על כל פלטפורמה; שימושי ליישומים ארגוניים
- קשה לממש look & feel חדש
- סכנה של מראה מיושן, אם מערכת ההפעלה החליפה את המראה של הרכיבים אבל האמולציה לא עודכנה (למשל מראה של חלונות 2000 על מערכת חלונות XP)
- אי התאמה לקביעת התצורה של המשתמשת (אם היא בחרה למשל להשתמש במראה של חלונות 2000 על חלונות XP)

שחרור משאבים

- חלק מהעצמים שמרכיבים את המנשק הגראפי מייצגים למעשה משאבים של מערכת ההפעלה, כמו חלונות, כפתורים, צבעים, גופנים, ותמונות
- כאשר עצם שמייצג משאב נוצר, הוא יוצר את המשאב, ואם לא נשחרר את המשאבים הללו, נדלדל את משאבי מערכת ההפעלה
- למשל, צבעים בתצוגה של 8 או 16 סיביות לכל פיקסל
- ב-SWT, אם יצרנו עצם שמייצג משאב של מערכת ההפעלה, צריך לקרוא לשירות `dispose` כאשר אין בו צורך יותר
- `dispose` משחרר גם את כל הרכיבים המוכלים
- על מנת לחסוך במשאבים, יש הפרדה בין מחלקות שמייצגות משאבים (למשל `Font`) וכאלה שלא (`FontData`)

תחושת המנשק בפלטפורמות שונות

- בחלונות ולינוקס משתמשים בצירופים Control-C, Control-V עבור גזור והדבק
- במחשבי מקינטוש יש מקש Control, אבל יש גם מקש Command, וגזור והדבק מופעלי על ידי Command-C, Command-V, ולא על ידי צירופי Control
- תוכנית שמפעילה גזור והדבק ע"י Control-C/V תחוש לא טבעית במקינטוש
- ב-SWT מוגדרים המקשים Control וכדומה, אבל גם "מקשים מוכללים" MOD1, MOD2, ו-MOD3, כאשר MOD1 ממופה ל-Control בחלונות אבל ל-Command במקינטוש
- בעיה דומה: הפעלת תפריט הקשר; הקלקה ימנית בחלונות, אבל במקינטוש יש לעכבר רק לחצן אחד; מוגדר אירוע מיוחד

פריסה נכונה

- פריסה נכונה של רכיבים היא אחד האתגרים המשמעותיים בפיתוח מנשק גראפי
- התוכנית צריכה להבטיח עד כמה שאפשר שהמנשק יראה תמיד "נכון", למרות מסכים בגדלים שונים וברזולוציות שונות, כאשר רכיבים כגון טבלאות ושדות טקסט מציגים מעט מידע או הרבה, וכאשר המשתמשת מקטינה או מגדילה את החלון
- אלגוריתמי פריסה מתוחכמים עבור מיכלים, כגון `GridLayout`, מסייעים, אבל צריך להבין כיצד מתבצעים חישובי הפריסה וכיצד להשפיע עליהם

שני סיבוכים

- יש רכיבים שגובהם תלוי ברוחבם או להיפך; למשל תווית או סרגל כלים שניתן להציג בשורה אחת ארוכה, או לפרוס על פני מספר שורות קצרות
- לכן, `computeSize` מאפשר לשאול את הרכיב מה גובהו הרצוי בהנתן רוחב מסוים ולהיפך, ולא רק מה הגודל הרצוי ללא שום אילוץ
- יש רכיבים שעלולים לרצות גודל עצום, כמו עורכי טקסט, טבלאות, ועצים (ובעצם כל רכיב שעשוי לקבל פס גלילה)
- הגודל הרצוי שהם מדווחים עליו אינו מועיל; צריך לקבוע את גודלם על פי גודל המסך, או על פי מספר שורות ו/או מספר תווים רצוי

חישובי פריסה

- חישובי פריסה מתבצעים ברקורסיה על עץ ההכלה, אבל בשני כיוונים: מלמטה למעלה (מרכיבים מוכלים למיכלים שלהם עד מעטפות חיצוניות) ומלמעלה למטה
- חישובים מלמטה למעלה (postorder ברקורסיה) עונים על השאלה "באיזה גודל רכיב או מיכל רוצים להיות?"
- חישובים מלמעלה למטה (preorder) עונים על השאלה "בהינתן גודל למיכל, היכן ובאיזה גודל למקם כל רכיב?"

חישובים מלמעלה למטה

- השירות `layout` פורס את הרכיבים המוכלים במיכל לאחר שגודל המיכל נקבע (על ידי `setSize` או `setBounds`)
- המיכל פורס בעזרת אלגוריתם הפריסה שנקבע לו
- לפעמים, הפריסה לא תלויה בגודל הרצוי של הרכיבים; למשל, אלגוריתם הפריסה `FillLayout` מחלקת את המיכל באופן שווה בין הרכיבים המוכלים, לאורך או לרוחב
- בדרך כלל, הפריסה כן תלויה בגודל הרצוי של הרכיבים; ב-`GridLayout`, למשל, הרוחב של עמודות ושורות לא נמתחות נקבע על פי הרכיב עם הגודל הרצוי המקסימאלי בהן, ושאר העמודות והשורות נמתחות על מנת למלא את שאר המיכל
- רכיבים זוכרים את גודלם הרצוי כדי לא לחשבו שוב ושוב

פריסה מלמטה למעלה

- כל רכיב צריך לדעת באיזה גודל הוא רוצה להיות (שם השירות ב-SWT הוא `computeSize`, בספריות אחרות `preferredSize`)
- יש ספריות שבהן כל רכיב צריך לדעת מה גודלו המינימלי (`minimumSize`), אבל לא ב-SWT
- רכיב פשוט מחשב את גודלו הרצוי על פי תוכנו (למשל על פי גודל התווית או הצלמית שהוא מציג) ועל פי החוקים הויזואליים של המנשק (רוחב המסגרת סביב התווית, למשל)
- מיכל מחשב את גודלו הרצוי על ידי חישוב רקורסיבי של הגודל הרצוי של הרכיבים המוכלים בו, והרצת אלגוריתם הפריסה של המיכל על הגדלים הללו
- אבל זה מסתבך

אריזה הדוקה

- השירות `pack` מחשב את גודלו הרצוי של רכיב או מיכל וקובע את גודלו לגודל זה; המיכל נארז באופן הדוק
- שימושי בעיקר לדיאלוגים לא גדולים
- סכנת חריגה: אם המיכל מכיל רכיב עם גודל רצוי ענק (טבלה ארוכה, תווית טקסט ארוכה), החלון עלול לחרוג מהמסך
- עבור חלונות (כולל דיאלוגים), עדיף לחשב את הגודל הרצוי ולקבוע את גודל המעטפת בהתאם רק אם אינו חורג מהמסך, אחרת להגביל את האורך ו/או הרוחב
- סכנת איטיות: אם המיכל מכיל המון רכיבים, חישוב גודלו הרצוי יהיה איטי (רוחב עמודה בטבלה ארוכה); כדאי להערך את הגודל הרצוי בדרך אחרת

אלגוריתמי אריזה

- `FillLayout`: רכיבים בשורה/עמודה, גודל אחיד לכולם
- `RowLayout`: רכיבים בשורה/עמודה, עם אפשרות שבירה למספר שורות/עמודות, ועם יכולת לקבוע רוחב/גובה לרכיבים
- `GridLayout`: כפי שראינו, סריג שניתן לקבוע בו איזה שורות ועמודות ימתחו ואיזה לא, ולקבוע רוחב/גובה לרכיבים
- `FormLayout`: מיקום בעזרת אילוצים על ארבעת הקצוות (או חלקם) של הרכיבים; אילוצים יחסיים או אבסולוטיים ביחס למיכל (למשל, באמצע רוחבו ועוד 4 פיקסלים) או אילוצים אבסולוטיים ביחס לנקודת קצה של רכיב אחר (דבוק לרכיב אחר או דבוק עם הפרדה של מספר פיקסלים נתון)
- `StackLayout`: ערימה של מיכלים בגודל זהה אבל רק העליון נראה; שימושי להחלפה של תוכן מיכל או חלון

סיכום מנשקים גראפיים

- דע/י את מקומך
- שלושה מנגנונים כמעט אורתוגונליים: ירושה, הכלה, אירועים
- פגמים במנשק גראפי נובעים במקרים רבים או מפריסה לא נכונה של רכיבים במיכל, או מחוסר תגובה או תגובה לא מספיקה לאירועים
- לא קשה, אבל צריך להתאמן בתכנות מנשקים גראפיים
- ספר, GUI Builder, ודוגמאות קטנות מסייעים מאוד
- ממשקים מורכבים בנויים לפעמים תוך שימוש בעצמי תיווך בין רכיבי המנשק ובין החלק הפונקציונאלי של התוכנית (המודל); למשל, jface מעל SWT; קשה יותר ללמוד להשתמש בעצמי התיווך, אבל הם מקטינים את כמות הקוד שצריך לפתח ומשפרים את הקונסיסטנטיות של המנשק