

# חלק 12

## לפני תכנות: מבוא

### להנדסת תוכנה

# לפני ואחרי פיתוח תוכנה

- תהליך הפיתוח של תוכנה אינו מורכב רק מתיכנות ובדיקות, הנושאים שעליהם דיברנו עד כה
- התהליך מתחיל לפני הפיתוח ונמשך גם אחרי שהפיתוח הסתיים
- הנדסת תוכנה היא תחום הנדסי העוסק בכל ההיבטים של יצירת מערכות תוכנה.
- בחלק הזה של הקורס נדון בשלבים שלפני ואחרי הפיתוח, במה שמשותף להם ולפיתוח ובמה ששונה
- הדיון יהיה תמציתי ולא ממצה; הנושא רחב מדי
- הדיון אינו ספציפי לתכנות מונחה עצמים

# מחזור החיים של תוכנה

- ניתוח דרישות (requirements analysis)
- תיכון (design)
- מימוש ובדיקות
- בדיקות קבלה
- ייצור (production)
- תחזוקה ושינויים

התייחסות מיוחדת למקרה שמערכת התוכנה היא חלק ממערכת ממוחשבת הכוללת חומרה ותוכנה.

# מפל או ספירלה?

- המודל המסורתי של מחזור חיים נקרא מודל מפל המים (waterfall model). כל שלב מתבצע לאחר שקודמו הסתיים (אך ניתן לחזור לשלב קודם לצורך תיקון).
- מודל הספירלה (spiral model) שהוצע מאוחר יותר מפתח את המערכת באופן אבולוציוני. מתחילים מפיתוח מערכת מינימלית, ומבצעים את כל השלבים. לאחר סיום מעריכים את המוצר הנוכחי, מחליטים מה להוסיף, וחוזרים על כל השלבים
- מודל הספירלה מאפשר לראות מוצר חלקי ולהעריך אותו.
- אבל מפל המים משקף את הרצוי: רצוי לא לטעות.
- קיימים גם מודלים אחרים לתהליך הפיתוח.

# מחירן של טעויות

- **ככל שטעות מתגלה מוקדם יותר, מחיר תיקונה קטן יותר**
- נניח שטעינו בניתוח הדרישות ושכחנו פעולה מסוימת שהתוכנה צריכה לבצע
- אם נגלה את הטעות לפני המעבר לתיכון, המחיר יהיה מינימאלי, אולי עיכוב קטן בלוח הזמנים
- אם נגלה בזמן התיכון, נצטרך אולי לזרוק חלק מהתיכון שלא יתאים לדרישות המתוקנות
- אבל אם נגלה את הטעות רק בזמן בדיקות הקבלה, נצטרך אולי לזרוק חלקים גדולים מהתיכון ומהמימוש!
- עדיף לגלות טעויות מוקדם; לשם כך צריך לתכנן בקפדנות את תהליך הפיתוח הכולל, ולהשתדל להשתמש בשיטות שימזערו טעויות ואת הצורך לחזור אחורה לשלב קודם

# ניתוח דרישות

- מטרת השלב הזה להבין איך מה מוצר התוכנה צריך לעשות ואיך הוא צריך להתנהג
- באופן יותר פרטני, מטרת השלב הזה היא לחבר מסמך דרישות שיהווה בסיס לתיכון התוכנה
- מה צריך להכיל מסמך הדרישות.
- קיימים סטנדרטים למבנה מסמך דרישות.
- האם מבנה המסמך מבטיח שלא נשכח דרישה חשובה?

# מערכות תוכנה לדוגמה

- כדי לנסות לברר איך לנתח את הדרישות מתוכנה, נחשוב על שלוש דוגמאות שונות מאוד זו מזו
- המטרה היא להבין את המכנה המשותף בתהליך ניתוח הדרישות, אבל גם להבין היכן דרושה התאמה לסוג התוכנה
- אם משתמשים בדוגמאות דומות, נוטים לבנות תהליך שמתאים לסוג מסוים של תוכנות אבל לא לאחרות
  
- מערכת תוכנה להגשת תרגילים באוניברסיטה
- תוכנת טייס אוטומטי לדור הבא של מטוסי הנוסעים
- משחק מחשב שקהל היעד שלו הן ילדות בנות 6 עד 9

# סוגים של מערכות תוכנה

- מערכת להגשת תרגילים היא דוגמה טיפוסית למערכת מידע. קיימות מערכות דומות רבות, כמו מערכת המידע של ספריית השאלה, מערכות לטיפול ברישום לקורסים ובציונים, וכדומה.
- תוכנת טייס אוטומטי היא תוכנת זמן אמת שרוב הנתונים שהיא מטפלת בהם רציפים. אלו מערכות תגובתיות. דוגמאות אחרות כוללות תוכנות לבקרת תהליכים במפעלים (מפעלים כימיים, מפעלי מזון, וכדומה), תוכנות לרובוטים (החל ברובוטים תעשייתיים וכלה בגשושיות מאדים), ועוד.
- משחק מחשב, הוא מוצר בידור או לימוד. הדרישות פחות ברורות מאשר ממוצרי תוכנה אחרים. הגדרת הדרישות הכללית של משחק כוללת סוגה (ז'אנר, כגון הרפתקאות, פעולה, לומדה, וכולי) וקהל יעד (מין וגיל).
- קטגוריות אחרות, מאפיינים אחרים.



# חזרה למסמך הדרישות

- דרישות פונקציונליות: איך התוכנה צריכה להגיב למשתמש (כולל משתמשים שהם בעצם תוכניות אחרות); איך התוכנה צריכה להתמודד עם כשלים בחומרה ובתוכנה
- דרישות ביצועים: זמני תגובה לפעולות והחשיבות של עמידה בזמני התגובה הנדרשים, מגבלות משאבים (כמות זיכרון ונפח אחסון, מהירות מעבד ורכיבים אחרים)
- שינויים צפויים בעתיד; נשמע כמו סתירה (אם השינויים ידועים כיום למה צריך לחכות לעתיד?), אבל בדרך כלל אפשר לקבל מושג על שינויים שסביר שנצטרך ושינויים שסביר שלא
- לוחות זמנים לפיתוח ומסירה (לפעמים במסמך אחר).
- נימוקים להחלטות (ולהחלטות שנדחו)
- אנו נתרכז בהגדרת הדרישות הפונקציונליות

# שימושים נוספים למסמך הדרישות

- מסמך הדרישות חשוב לא רק לצורך התיכון והמימוש
- אפשר להשתמש בו על מנת לחבר את המדריך למשתמש; מדריך כזה צריך לייצר ממילא, וייצור מוקדם שלו על סמך מסמך הדרישות יכול להצביע על כך שהמערכת תהיה קשה לשימוש, ויכול לסייע ללקוח להבין כיצד המערכת תעבוד
- אפשר להשתמש בו על מנת לתכנן את בדיקות הקבלה; שיקולים דומים

# כיצד מגלים את הדרישות הפונקציונליות

- בעיקר על ידי דוגמאות לשימוש במערכת (use scenarios)
- מייצרים סדרה של דוגמאות לשימוש במערכת ומתעדים אותם, החל בדוגמאות פשוטות של אינטראקציה פשוטה ו- "נכונה" והלאה לדוגמאות מסובכות עם שגיאות
- בכל דוגמה: מה המשתמש עושה ואיך המערכת מגיבה
- השיטה הזו מבוססת על העיקרון שניתוח הדרישות צריך להתחיל בקבלת מושג על מה המערכת צריכה לעשות ולא על איך היא צריכה להיות בנויה
- יש צורך לראיין משתמשים אופייניים (או נציגים שלהם).
- את החשיבה על המבנה דוחים עד שיהיה ברור מה היא צריכה לעשות

# דוגמאות שימוש (מערכת הגשת תרגילים)

- המרצה יוצר תרגיל חדש; זה מאפשר לו להעלות קבצים לתרגיל; שאר סגל הקורס יכול לראות את התרגיל החדש, אבל לא התלמידים (כי התרגיל עדיין לא הוטל)
- המרצה מטיל תרגיל; נקבע לתרגיל מועד הגשה, ועכשיו גם תלמידים בקורס יכולים לראות את התרגיל
- תלמידה מתחילה לעבוד על תרגיל; היא יכולה לייצר קבצים חדשים במדריך של התרגיל ולשנות קבצים; היא יכולה לשמור גרסאות מסוימות של התרגיל (snapshots)
- תלמידה מגישה תרגיל; הגרסה הנוכחית של הקבצים נשמרת כתמונת מצב, תמונת המצב הזו מסומנת כמוגשת; סגל הקורס יכול כעת לראות את ההגשה ולשנות את הקבצים; התלמידה יכולה לראות את הגרסה המוגשת אבל לא לשנות אותה

# **דוגמאות שימוש בסוגי מערכות אחרים**

- סביר שדוגמאות שימוש הן דרך טובה להגדיר גם טייס אוטומטי
- דוגמאות השימוש צריכות לכלול תגובה למידע מסנסורים, ולא רק להוראות ממשתמש אנושי
- במערכות זמן אמת הדרישה כוללת פרק זמן מירבי לתגובה.
- האם דוגמאות שימוש מועילות גם לגילוי הדרישות ממשחק?  
לא בטוח

# המטרה בגיבוש הדרישות הפונקציונליות

- דוגמאות השימוש הן אמצעי בדרך למטרה: הגדרה פורמלית ומלאה ככל האפשר של הדרישות הפונקציונליות
- הגדרת הדרישות היא בעצם הגדרה של חוזה של המערכת מול הלקוחות שהם המשתמשים (אנושיים או לא אנושיים)
- המערכת מספקת קבוצה של שירותים
- לשירותים אין תנאי קדם; המערכת לא סומכת על המשתמש
- אם הקלט שלהם תקין, השירותים משנים את המצב המופשט של המערכת בהתאם לתנאי אחר מוגדרים
- לפעמים המערכת משנה את המצב המופשט ביוזמתה על מנת לקיים אילוצים (למשל בטייס אוטומטי)
- בדרך כלל השאילתות קשורות בעיקר למנשק למשתמש

# המצב המופשט

- מהי השפה שבה נגדיר את המצב המופשט של המערכת?
- זו שאלה מרכזית בהגדרת הדרישות של כל תוכנה
- בשפות מתמטיות פשוטות קשה לתאר מערכות מורכבות
- תשובה אפשרית אחת (לא מוצלחת): נשתמש בשפה דומה לשפת התכנות, כלומר בעצמים ושירותים; זה לא מוצלח כי זה מערבב את שלבי הגדרת הדרישות והתיכון; כדאי להפריד
- יש הטוענים שיש שפות אוניברסליות שמתאימות לתיאור המצב המופשט של כל מערכת
- זו טענה לא סבירה: כנראה שהמצב המופשט של טייס אוטומטי, למשל, יוגדר על ידי מודל מתמטי רציף של טיסה, ולכן דרושה שפה רציפה לתיאור מצב המערכת; שפה כזו לא תתאים למערכת להגשת תרגילים או לספריית השאלה

# הבעת מצב מופשט בעזרת ישויות ויחסים

- למרות שדרוש מגוון של שפות לתיאור המצב המופשט של מערכות מחשב, יש משפחה של שפות שהצליחה לתאר את המצב של מגוון גדול של מערכות
- המשפחות הללו מבוססות על תיאור המצב על ידי קבוצות של ישויות ותתי קבוצות שלהן, על ידי יחסים בין הישויות, ועל ידי אילוצים על הקבוצות והיחסים
- תיאור כזה של מצב של מערכת נקרא מודל נתונים ( data model); לפעמים קוראים לתיאור כזה מודל עצמים, אבל זה שם לא מוצלח כי הוא מרמז על ערבוב בין הגדרת הדרישות והתיכון
- משפחת השפות הללו כוללת את UML (נפוצה, גראפית, לא פורמלית), את Alloy (מחקרית, תיאור גראפי חלקי, פורמלית ומאפשרת הוכחות ידניות או אוטומטיות), ועוד



# מניתוח דרישות לתיכון

- ניתוח דרישות עוסק ב"מה" - בעולם הבעייה.
- תיכון הוא התחלת הטיפול ב"איך" - עולם הפתרון, המימוש.
- בשלב ניתוח הדרישות אין התייחסות לאילוצי מימוש.
- לפני שמתחילים בתיכון, יש לברר את אילוצי המימוש (פלטפורמה - חומרה ותוכנה, שפת תכנות וכו').

# המטרות של שלב התיכון

- להמציא מבנה כללי לתוכנה
- רכיבי המבנה צריכים לייצג ישויות עם משמעות ברורה
- המבנה צריך לאפשר מימוש של הדרישות (כולל שינויים צפויים)
- המבנה צריך להיות קל לפיתוח ותחזוקה; זה בדרך כלל דורש מבנה מודולרי עם מעט תלויות בין מודולים
- פיתוח עקרון המימוש של כל רכיב במבנה, עד רמת המחלקה/פרוצדורה
- התיאור של מחלקה או פרוצדורה צריך לכלול את החוזה שלה ואת עקרון המימוש אם הוא לא ברור מאליו (למשל דורש אלגוריתם לא טרויאלי)

# מסמך התיכון (design notebook)

- תרשים שמתאר את חלוקת התוכנה למודולים ואת התלויות בין מודולים
- צמתים הם מחלקות ופרוצדורות
- כאשר צומת א' משתמש בצומת ב' (מחלקה משתמשת במחלקה אחרת או בפרוצדורה, למשל), קשת מ-א' ל-ב'
- כאשר מחלקה א' מרחיבה את מחלקה ב' (או מממשת מנשק), קשת מ-א' ל-ב'
- וכן פירוט על כל מחלקה ופרוצדורה
- הפירוט כולל תיאור של מה היא מייצגת, מה החוזה שלה, ושיקולים שהובילו להגדרות הללו (כולל שיקולים שפסלו מבנים חלופיים)

# מסמכולוגיה

- מסמך תיכון מלא ומפורט יאפשר לוודא שהתיכון עונה על הדרישות, יאפשר לתכנן את המימוש (ואת לוחות הזמנים שלו) יספק למממשים הגדרות ברורות של המימוש הנדרש, ויאפשר להעריך את ההשפעה והעלות של שינויים עתידיים
- קשה להפיק מסמך כזה (כמו שקשה להפיק מסמך דרישות מלא ומפורט), אבל בדרך כלל העבודה הזו משתלמת
- מאידך, הפקת המסמכים הללו איננה מטרה בפני עצמה, והיצמדות קפדנית לפורמט זה או אחר לא תבטיח שמערכת התוכנה תהיה מוצלחת; העיקר הוא להבין לעומק את הדרישות ולתכנן מערכת טובה; התיעוד של השלבים הללו חשוב, אבל תיעוד טוב של החלטות גרועות לא יועיל

# ביקורת (design review)

- לפני שמסיימים את שלב התיכון, צריך לבדוק שהמערכת שתכננו טובה ועונה על הדרישות
- תהליך הבדיקה נקרא design review
- לבדיקה שני חלקים
- בדיקה מול הדרישות: איך המערכת מייצגת את המצב המופשט של מודל הנתונים? האם הייצוג שומר על האילוצים של מודל הנתונים? האם הפעולות מקיימות את חוזהן? האם הפעולות עומדות בדרישות הביצועים? האם ניתן יהיה לבצע במערכת שינויים ושיפורים צפויים?
- בדיקה מול מדדים של איכות תוכנה: בעיקר מודולריות של המבנה הכולל, הימנעות תלויות לא הכרחיות והחלשת תלויות הכרחיות

# תכנון המימוש (והבדיקות)

- לאחר ששלב התיכון מסתיים, אפשר להתחיל במימוש
- לא לגעת עדיין במקלדת! קודם צריך לתכנן את המימוש
- מימוש מלמטה למעלה: קודם את מחלקות העזר ואחר כך את המחלקות שמשתמשות בהן; בדרך כלל את מחלקות העזר הנמוכות קל יותר לממש; מהקל אל הקשה; האסטרטגיה מקלה על מימוש בדיקות, כי מחלקות העזר יבדקו מוקדם
- מימוש מלמעלה למטה; קודם את המחלקות הגבוהות ובסוף את מחלקות העזר; התמודדות מוקדמת עם קשיים ואי ודאויות, כי בדרך כלל המחלקות העליונות יותר ייחודיות
- באופן כללי, את הקשיים צריך להקדים; מלמעלה למטה משיג את זה, אבל לפעמים יש מחלקות נמוכות קשות למימוש
- הקושי בסוף כמעט מובטח: קשיי האינטגרציה

# המעגל נסגור

- הגענו לנקודה שבה מתחילים לממש, הנקודה שבה התחלנו את הקורס
- השלבים שלפני המימוש מיועדים לתכנון התוכנה: מה היא תעשה (שלב ניתוח והגדרת הדרישות) ואיך היא תעשה את זה (שלב התיכון)
- קשה להגדיר מה יעשה משהו שלא קיים עדיין, ואיך; לשם כך דרושה שפה מתאימה; השפה צריכה לאפשר להגדיר את המצב המופשט של המערכת ואת הפעולות על המצב הזה
- הגדרות פורמאליות עדיפות על הגדרות לא פורמאליות, אבל צריך להתחשב במאמץ הדרוש לעומת התועלת הצפויה
- תהליך מסודר ומסמכים מפורטים מועילים, אבל רק אם ההחלטות שמתקבלות הן החלטות טובות