חלק 15

תוספים ותסריטים

(Plugins & Scripts)

למה תוספים?

- לפעמים תוכנית משתמשת ברב-צורתיות דרך מנשקים כדי לספק למשתמשים מגוון גדול של מנגנונים מאותו סוג
- למשל: פילטרים בתוכנות לעיבוד תמונה/וידאו/קול, מחלקות להצגת והדפסת תמונות וסרטונים מסוגים שונים בדפדפנים ומעבדי תמלילים, תוספים שונים לסביבות פיתוח תוכנה (סינכרון מול מאגרי קוד מקור, עורכים, קומפיילרים)
- במקרים כאלה, עדיף שלא לארוז את כל המנגנונים הללו ביחד עם התוכנה המרכזית, אלא לטעון אותם דינאמית ובאופן מפורש בזמן ריצה כתוספים (plugins)
- התוכנית המרכזית מחפשת ומוצאת את התוספים בעצמה
- זה מאפשר עדכון והוספת מנגנונים בנפרד מהתוכנה המרכזית
- וזה מאפשר ליצרני תוכנה עצמאיים להוסיף יכולות לתוכנה

תוכנה כפלטפורמה לתוספים

- בארכיטקטורה מבוססת תוספים או תומכת תוספים, התוכנה המרכזית משמשת כפלטפורמה פתוחה שתפקידה (בעיקר או בין היתר) לטעון ולהפעיל תוספים
- מדוע כדאי ליצרן התוכנה המרכזית לתמוך בתוספים?
- בעיקר כי תמיכה כזו מעודדת שוק משני של תוספים, שוק שבו משתמשים רוכשים תוספים ממפתחים עצמאיים
- שוק כזה מפצל את הפיתוח (והסיכון) בין מספר חברות, שכל אחת יכולה להתמחות בנושא ספיציפי
- שוק כזה עוזר למלא צרכים של קבוצות משתמשים שהן קטנות מכדי להיות חשובות כלכלית ליצרן המרכזי
- יש מתח בין הרצון של היצרן המרכזי להגדיל את הכנסותיו ובין הצורך שלו לשמור על הכדאיות של מפתחי התוספים

דוגמאות לתוכנות מבוססות תוספים

סביבת הפיתוח Eclipse היא מבוססת תוספים לחלוטין.

- כמעט כל היכולות שלה מוקנות לה על ידי תוספים
- בלי התוספים זאת סביבה גראפית ללא יכולות מועילות למפתחי תוכנה.
- התצורה הבסיסית של אקליפס כוללת מספר רב של תוספים, בעיקר לפיתוח בג׳אווה.
- יצרני תוכנה רבים נוספים משווקים תוספים לאקליפס או תוכנות שניתן להפעיל גם כתוסף של אקליפס.
- לקוחות קונים את הקומבינציה הדרושה להם של תוספים.

דוגמא נוספת: Adobe InDesign, תוכנה לעימוד מסמכים.

תוכנות תומכות תוספים

- יש תוכנות רבות שבהן הארכיטקטורה של המוצר הבסיסי אינה מבוססת על תוספים, אבל הן תומכות בתוספים.
- דוגמאות כוללות דפדפנים, תוכנות עריכת תמונה, קול, ווידאו, שתומכות בדרך כלל בפילטרים כתוספים (למשל Adobe Photoshop), מעטפות של מערכות הפעלה (Windows Explorer, למשל, תומך בתוספים מסוגים שונים, כמו תוספים שמאפשרים סינכרון קבצים מול שרת), וסביבות פיתוח (Microsoft Visual Studio למשל תומך בתוספים שמוסיפים קומפיילרים, תמיכה בסינכרון קבצים, עורכים גראפיים למנשקים גראפיים, ועוד).

שווקים משניים של תוכנה

- מאפשרים ומעודדים התמקצעות. חברות מתמחות בסוג מסוים של תוספים, אולי כתוספים למספר תוכנות בסיס שונות. למשל, עורכים לנוסחאות מתמטיות למספר מעבדי תמלילים ותוכנות עימוד.
- המתח בו פועל היצרן של התוכנה המרכזית:
  - מצד אחד, שוק התוכנה המשני משרת אותו, מכיוון שהזמינות של תוספים מגדילה את האטרקטיביות של המוצר הבסיסי, ולכן היצרן המרכזי חייב לשמר אותו.
  - מצד שני, הוספת יכולות לתוכנה הבסיסית מגדילה את האטרקטיביות שלה ומוזילה את העלות הכוללת שלה לחלק מהצרכנים. כך גם שינוי דרסטי במבנה, שיפריע לכותבי התוספים, עשוי לשרת את היצרן המרכזי.

האינטראקציה של תוסף עם תוכנת הבסיס

- לתוסף יש שלושה סוגי אינטראקציה עם תוכנת הבסיס שטוענת אותו
- התוכנה הבסיסית צריכה להיות מסוגלת לברר מהו סוג התוסף (מה המנשק שהוא מממש) ומה שם המחלקה המממשת בתוסף (על מנת ליצור עצמים או לקרוא לשגרות)
- התוכנה הבסיסית צריכה להיות מסוגלת לקרוא לקוד בתוסף, מה שמחייב אותו לכלול מחלקה שמממשת מנשק מוסכם
- בדרך כלל התוסף אינו עצמאי לחלוטין; על מנת לממש את הפונקציונאליות שלו, הוא קורא לשירותים של התוכנה הבסיסית (למשל על מנת לחקור את העצם שהועבר לו)
- לפעמים התוסף יכול להיות תלוי בתוספים אחרים או אפילו לאפשר לתוספים משניים להרחיב את יכולותיו

מציאת התוסף ומידע התצורה שלו

- איזה תוספים יש לנו?
- מה סוג התוסף (עורך, מסנכרן, פילטר)? מה שם המחלקה/-מחלקות שתוסף תורם? מתי להפעיל אותו (רק תמונות שחור-לבן או כל תמונה, קבצי ג׳אווה או כל קובץ), מה התווית/צלמית שמייצגת אותו עבור המשתמש?
- מציאת התוספים: או מבנה נתונים קבוע לרישום התוספים, או חיפוש שלהם בזמן ריצה, למשל חיפוש כל התוספים במדריך plugins של התוכנה; המנגנון השני עדיף כי הוא מונע חוסר התאמה בין מבנה הנתונים ובין התוספים
- המידע על התוספים יכול להיות בקובץ תצורה שהתוכנה קוראת, ויכול להיות מוחזר על ידי שגרה בתוסף עצמו; בג׳אווה משתמשים בקובץ תצורה בגלל שאי אפשר להשתמש בשם שגרה קבוע להחזרת המידע; לכל שגרה שם ייחודי

קריאה לתוסף

- בדרך כלל תוסף מספק מחלקה או מחלקות שמממשות מנשק עם חוזה מוגדר שהתוכנה הבסיסית מגדירה
- לפעמים המנשק הזה פשוט: למשל, פילטר עבור תוכנה לעיבוד תמונה עשוי לממש מנשק עם שירות אחד, `filter`, שמקבל תמונה ומחזיר תמונה חדשה
- ולפעמים הוא מורכב: תוסף להצגת סוג תמונה מסוים בדפדפן צריך לספק שירותים שיתארו את התמונה (גודלה בפיקסלים), שיציגו אותה על המסך בגודל נתון, ושיוסיפו ייצוג שלה לקובץ הדפסה

התוסף קורא לתוכנה הבסיסית

- בדרך כלל התוסף צריך להשתמש בשירותים של התוכנה הבסיסית
- למשל, פילטר על תמונה יקבל ככל הנראה את התמונה כהתייחסות לעצם שמייצג תמונה בתוכנת הבסיס
- על מנת לחקור את התמונה (גודלה, צבע של פיקסלים נתונים) הוא צריך לקרוא לשירותים של העצם הזה
- מפתח התוסף צריך להכיר את המנשק לתוכניתן (API) שתוכנת הבסיס חושפת לתוספים; זה יכול להיות ספר של 1000 עמודים, אם תוכנת הבסיס מורכבת וחשופה
- מפתחי תוכנת הבסיס צריכים לכתוב תיעוד חיצוני (למשתמשים מחוץ לחברה) של שירותים שתוספים צריכים; יש לזה עלות וזה מקטין את היכולת לשנות בעתיד

דוגמה: תוספים באקליפס

- תוספים מורכבים בדרך כלל ממארז jar ומקובץ תצורה בפורמט XML (קובץ טקסט)
- שני הקבצים הללו נשמרים במדריך שמייצג את התוסף בתוך מדריך ה-plugins של אקליפס
- תוסף מוסיף לתוכנה מחלקות שכל אחת מהן מרחיבה נקודת הרחבה (extension point) מסוימת; כל נקודת הרחבה מגדירה סוג של תוסף (עורך, פעולה שמופעלת מתפריט, וכו')
- קובץ התצורה מציין את נקודות ההרחבה ואת המחלקות, את התוספים שהתוסף תלוי בהם (ואולי את הגרסה הנדרשת), ותוויות טקסט וצלמיות
- תוספים יכולים להגדיר נקודות הרחבה חדשות שתוספים משניים יכולים להרחיב

תסריטים (scripts)

- תסריטים הם תוכניות קטנות שכתובות בדרך כלל בשפה קלה לשימוש ומיועדים בעיקר לאפשר אוטומציה של פעולות חוזרות (פעולות שהמשתמש יכול לעשות גם ידנית)
- במובנים מסוימים דומים לתוספים: התוכנית מוצאת תסריטים (למשל במדריך מסוים) ומאפשרת למשתמש להפעיל אותם (למשל מתפריט); התסריט יכול להשתמש ביכולות של התוכנה ובשירותים של העצמים שלה
- אבל בדרך כלל תסריטים מיועדים לאוטומציה ולא להרחבה
- למשל, תוכנת עיבוד תמונה עשויה להשתמש בתוספים עבור פילטרים, אבל לא סביר שפילטר יוגדר כתסריט; איטי מדי
- אבל פעולה כמו פתיחה של כל תמונה במדריך, ביצוע סדרת פעולות עליה ושמירה בשם אחר כן מתאימה לתסריט

תמיכה בתסריטים

- התוכנית הבסיסית צריכה לכלול פרשן לשפה שבו כתובים התסריטים; יש שפות מיוחדות לכך, למשל TCL, ושפות לא מיוחדות אבל מתאימות, כמו Python וגרסאות של jscript
- יש תוכנות שמגדירות שפה פרטית משלהן עבור תסריטים; מתאים בעיקר לתסריטים פשוטים יחסית
- שתי גישות ביחס לאינטראקציה בין תסריט ובין העצמים של התוכנית
- בגישה אחת, הפרשן מנהל את העצמים של התסריט ואת הזיכרון שלו בנפרד מניהול הזיכרון של תוכנית הבסיס; זו הגישה בשפות תסריטים מוכנות, כמו TCL
- בגישה השנייה, הפרשן רק מבצע את התסריט אבל משתמש בעצמים וניהול הזיכרון של תוכנית הבסיס

תסריטים ו-reflection

- בשתי הגישות, reflection מהווה מנגנון חשוב בגישור בין התסריט ובין העצמים של תוכנית הבסיס
- נניח (למרות הכל) שתסריט מממש פילטר על תמונה; נשתמש בתחביר דמוי ג'אווה

```
image = getSelection(); פונקציה מובנית
for (x=image.left(); x<=image.right(); x++)
  for (y=image.top(); i<=image.bottom() ...
    color = image.getPixel(x,y);
    ...
```

- בהינתן ביטוי כמו `image.left()` הפרשן משתמש ב-reflection על מנת לברר האם לארגומנט `image` יש שירות בשם `left` ועל מנת להפעיל אותו

סיכום תוספים ותסריטים

- ארכיטקטורה תומכת תוספים מאפשרת ליצרן תוכנה ליצור תוכנה שהיא גם פלטפורמה לתוספים שיפותחו על ידי גורמים אחרים (בד"כ חברות תוכנה, אבל לפעמים גם לקוחות גדולים)
- תמיכה בתוספים מקשה על תיכון התוכנה, גם בגלל הצורך להגדיר ולקָבֵּע באופן ברור את המנשקים למפתחי תוספים (שהם הרבה יותר מפורטים מהמנשק למשתמש) וגם בגלל הצורך לנהל את חיי התוכנה מול שוק התוספים
- למרות זאת, תוספים מאפשרים לבצע הרחבה של תוכנה קיימת באופן מודולרי מאוד ובלי תלות בשחרור גרסה חדשה
- תמיכה בתסריטים מאפשרת להרחיב יכולות של תוכנה באופן יותר פשוט אבל בד"כ יותר מוגבל מתוספים; תסריטים מפותחים בדרך כלל על ידי או עבור משתמשי קצה מסוימים (בודדים או ארגוניים)

חלק 16

לפני תיכנות: 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

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?

- את השלבים הללו במחזור החיים ניתן לראות כשלבים לינאריים שמתבצעים באופן סדרתי; כאשר מסיימים אחד מתחילים את הבא אחריו
- אבל ניתן גם לראות בשלבים אילוצים חד כיווניים: אי אפשר להתחיל בשלב לפני שמסיימים את קודמיו, אבל אפשר לחזור לשלב קודם במקום לעבור לשלב מתקדם יותר, אם מתגלה בעיה (כלומר מתגלה טעות בפלט של שלב קודם)
- למודל הראשון קוראים מודל מפל המים (waterfall model) ולשני מודל הספירלה (spiral model)
- מודל הספירלה ריאלי יותר: כשעושים טעויות יש לחזור לאחור
- 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
- 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

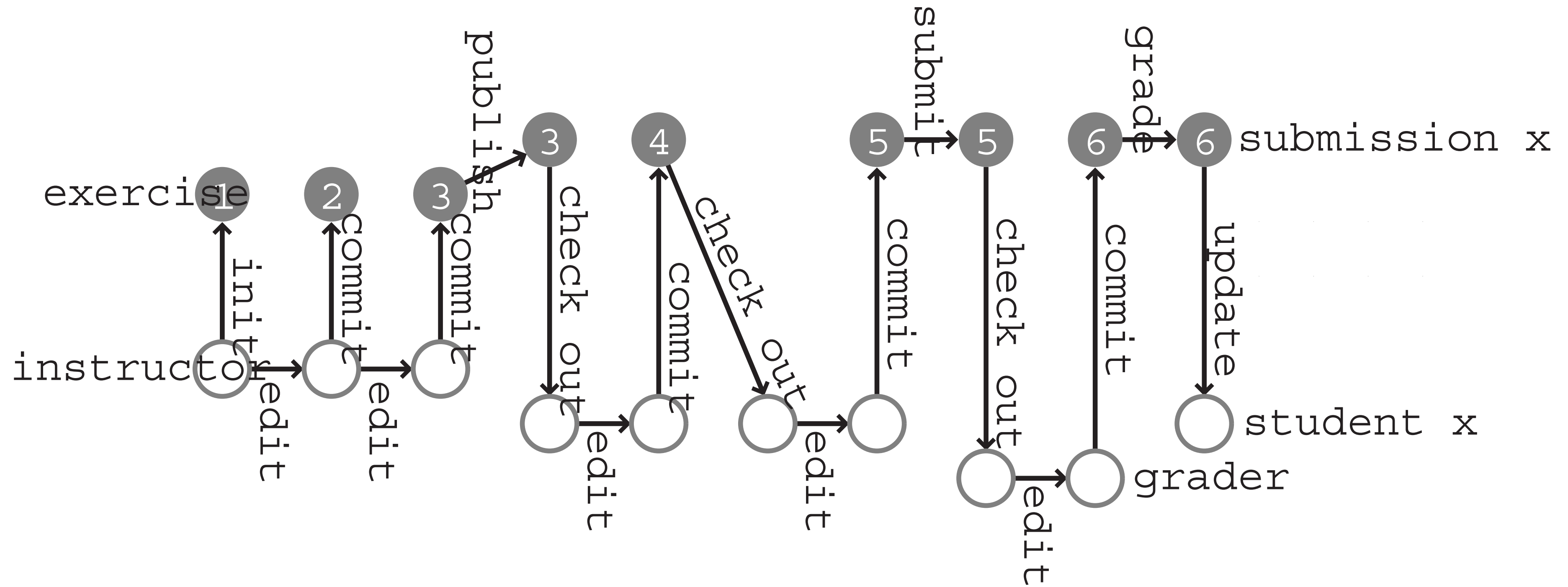
כיצד מגלים את הדרישות הפונקציונליות

- 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 אופיניים (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 השימוש באופן גראפי



- עיגולים מלאים מייצגים גרסה של תרגיל או הגשה במאגר המרכזי (על שרת); עיגולים ריקים מייצגים עותקים לעריכה על המחשב של מרצה/תלמידה/בודק

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

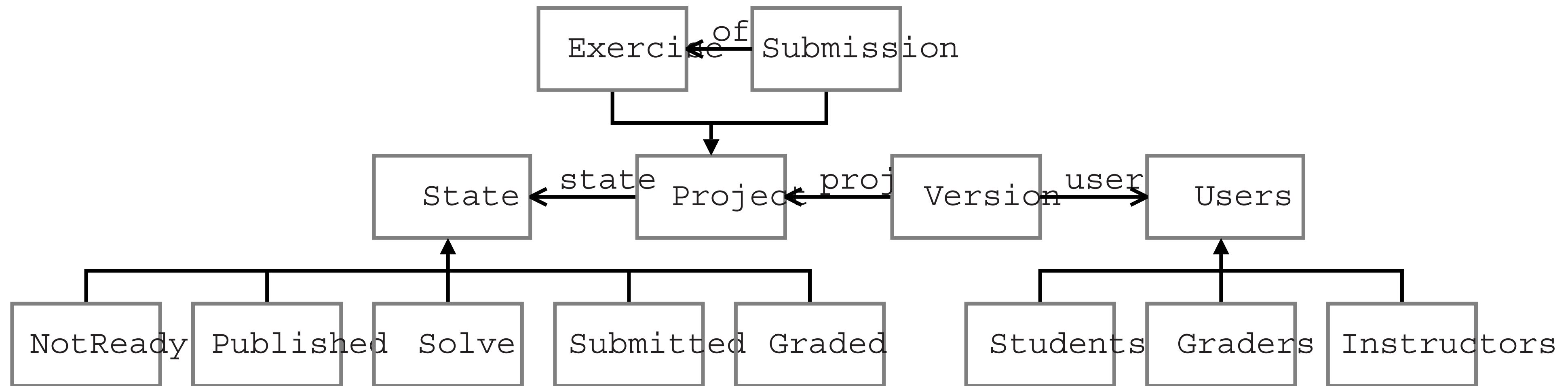
מדוגמאות שימוש לישויות ויחסים

- הישות הראשונה שנתקלנו בה במערכת הגשת התרגילים הייתה תרגיל; בכל קורס יש קבוצה של תרגילים, חלקם כבר הוטלו וחלקם עדיין לא (שני תתי קבוצות זרות וממצות); הקבוצה הזו יכולה לגדול
- ישות אחרת היא פרויקט, אוסף של קבצים ומדריכים (יש פרויקטים ששייכים למרצה ויש ששייכים לתלמידים); לפרויקט יש גרסאות ממוספרות
- יש שני סוגי פרויקטים: תרגילים (פרויקט ששייך למרצה) והגשות (פרויקט ששייך לתלמידה)
- כל הגשה קשורה לתרגיל; זהו יחס; יחס הוא קבוצה של זוגות סדורים; כאן כל הגשה קשורה לתרגיל אחד, אבל לתרגיל יכולות להיות קשורות הרבה הגשות, או אף אחת; זה אילוץ

המשך הישויות והיחסים בהגשת תרגילים

- תרגיל יכול להיות באחד משני מצבים: בהכנה או פורסם
- הגשה נוצרת כאשר תלמידה מורידה תרגיל ומעדכנת אותו
- הגשה יכולה להיות באחד משלושה מצבים: בעבודה, הוגשה, נבדקה
- את המצבים נמדל על ידי קבוצות מצבים קטנות
- יש גם דרכים אחרות למדל את המבנה הזו, למשל על ידי חלוקה של תרגילים/הגשות לתתי קבוצות שמתאימות למצבים
- יש הרבה מודלי נתונים שמתאימים לבעיה; הפתרון לא יחיד
- יש במודל של הגשת התרגילים עוד קבוצות ויחסים; התיאור הזה חלקי

דיאגרמת מודל הנתונים

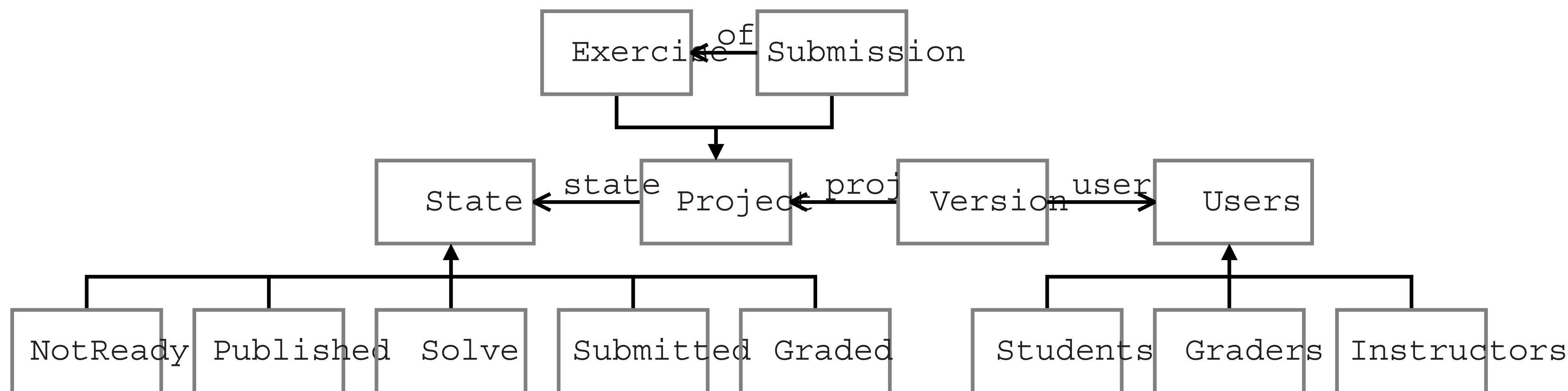


- מלבנים מייצגים קבוצות או תתי קבוצות
- חיצים סגורים מייצגים תתי קבוצות (ממצות אם החץ מלא)
- חיצים פתוחים מייצגים יחסים
- הדיאגרמה יכולה לתאר גם אילוצים פשוטים על גדלי הקבוצות והיחסים

אבני הבניין של מודל נתונים

- קבוצות ותתי קבוצות שלהם
- אילוצים על הקבוצות: גודל (בדיוק 1, 0 או 1, 0 או יותר, 1 או יותר), קבוצה קבועה או משתנה
- אילוצים על תתי הקבוצות: זרות, ממצות (או לא)
- יחסים: פורמלית, אלה קבוצות של זוגות סדורים, האיבר הראשון מקבוצה א' והשני מקבוצה ב'
- אילוצים על יחסים: כמה פעמים איבר בקבוצה מופיע ביחס (1-הרבה זו פונקציה מא' לב', 1-1 פונקציה חד ערכית, וכו')
- אילוצים יותר מורכבים, אם אין ברירה
- האילוצים מתארים מצבים של הקבוצות והיחסים שמשקפים מצב מופשט חוקי של המערכת

אילוצים במודל הנתונים

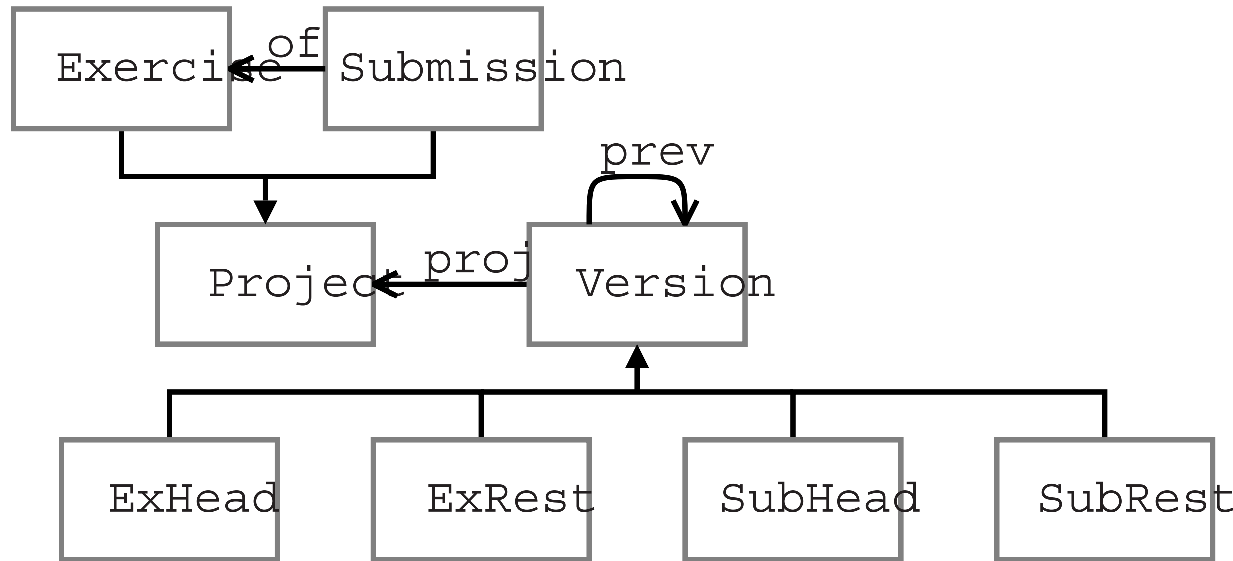


- גודל קבוצות: תתי הקבוצות של State בגודל 1 כל אחת
- סוגי יחסים: user ממפה כל איבר של Version לאיבר אחד בדיוק של Users, אבל איבר של Users יכול להיות ממופה לכל מספר אי שלילי של איבר Versions
- מורכבים: *if (p in Exercises) then p.state in NR or Pub*

מודל נתונים לעומת מחלקות ועצמים

- קבוצות דומות למחלקות (טיפוסים)
- תתי קבוצות זרות וממצות דומות למחלקות שמרחיבות מחלקה מופשטת (הקבוצה הכוללת)
- פונקציה מא׳ לב׳ דומה לשדה בא׳ שמתייחס לעצם של ב׳
- מאידך, יחסים שאינם פונקציות אי אפשר לבטא בעזרת שדות; אולי היחס עצמו הוא מחלקה
- זה עוזר להבין מהו מודל נתונים, אבל צריך לשמור על הפרדה בין מודל הנתונים, שתפקידו לאפשר לנו להגדיר את הדרישות (החוזה של המערכת), ובין עצמים ומחלקות, שהם המימוש
- למשל, סביר שלא נצטרך מחלקות עבור State ותתי הקבוצות שלו, אבל סביר שנצטרך עוד הרבה מחלקות עזר שלא מופיעות במודל הנתונים

אילוץ מורכב לדוגמה: שרשראות גרסאות



- גרסאות הן שרשראות של פרוייקט מסוים
- שרשרת של הגשה מתחילה בגרסה האחרונה של שרשרת של תרגיל

מיפוי לאחד לכל היותר: $prev: Version^* \rightarrow Version?$

גרסאות של תרגילים לעומת גרסאות של הגשות:

$if\ (v \in ExHead+ExRest)\ then\ (v.proj \in Exercises)$

$if\ (v \in SubHead+SubRest)\ then\ (v.proj \in Submissions)$

. . .

המשך האילוץ על שרשראות גרסאות

שורשים: $if\ (v \in ExHead)\ then\ v.prev = \{\}$

גרסאות של תרגיל מהוות שרשרת:

$$if\ (v \in ExRest)\ then\ v.prev \in ExHead+ExRest$$
$$and\ for\ all\ v' \in Version,\ v'.prev \neq\ v.prev$$

הגרסה הראשונה של הגשה היא האחרונה של תרגיל:

$$if\ (v \in SubHead)\ then\ v.prev \in ExHead+ExRest$$
$$and\ for\ all\ v'\ in\ ExRest,\ v'.prev\ \notin v.prev$$

גרסאות של הגשה מהוות שרשרת

$$if\ (v \in SubRest)\ then\ v.prev \in SubHead+SubRest$$
$$and\ for\ all\ v' \in Version,\ v'.prev \neq\ v.prev$$

- ברור שלא נצטרך ארבע תתי-מחלקות כאלה במימוש

רמת הפירוט ורמת הפורמאליות

- מצד אחד, מודל הנתונים אמור לאפשר לנו להבין מהם המבנים העיקריים של המערכת ואת היחסים ביניהם; הפירוט והפורמאליות מפריע להבין את העיקר
- מצד שני, לא טוב לסיים את שלב הגדרת הדרישות בלי פירוט מלא של מודל הנתונים ושל השירותים של המערכת; זה יקשה על קביעת לוחות הזמנים ועל התיכון והמימוש
- עדיף להתרכז בעיקר לפני שהמודל מתייצב; את הפרטים (שמות לכל תרגיל, שעה ותאריך לכל הגשה, וכו') אפשר להוסיף לאחר שהמודל מתייצב
- כדאי להגדיר את האילוצים בצורה פורמאלית לפני שעוברים לשלב התיכון על מנת שאפשר יהיה להוכיח תכונות של המערכת, וכדי שאפשר יהיה לבדוק את הנכונות של התיכון והמימוש

הגדרת הפעולות במערכת הגשת התרגילים

`init(User u, Folder f)`

Checks: u in Instructors

Effects: Creates a new Exercise

`checkout(User u, Version v)`

Checks: v.user == u or u in Instructors

Effects: copies the version to a new local folder

`checkout(User u, Exercise e)`

Checks: ((u in Students) and

(e.state in Published) and

(no Submission s: s.user==u and s.of==e))

Effects: gets the last version of e, creates a submission s

הגדרת הפעולות (המשך)

```
checkout(User u, Submission s)
```

Checks: ((u in Graders) and

(e.state in Submitted))

Effects: returns the last version of e

```
commit(User u, Project p, Folder f)
update(User u, Version v, Folder f)
publish(User u, Exercise e)
submit(User u, Submission s)
grade(User u, Version v)
enumerate_projects(User u)
enumerate_versions(User u, Project p)
```

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.
- לני 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 טריוואלי)

דרך הפעולה

- בוחרים ישות מסוימת במודל הנתונים כמטרה (או יחס) וממציאים עבורה ייצוג (כטיפוס, מחלקה)
- מפתחים את עקרון המימוש של הפעולות הקשורות בישות; זה דורש בדרך כלל הגדרה של מחלקות עזר ו/או פרוצדורות
- פיתוח עקרון המימוש של מחלקה דורש הגדרה של החוזה של מחלקות העזר, ואז אפשר לפנות ולפתח עבורן עקרון מימוש
- בעיקרון, כיוון התהליך הוא מלמעלה למטה (ממחלקות מרכזיות למחלקות עזר), אבל לפעמים הכיוון מתהפך, בעיקר כאשר מגלים שקשה או בלתי אפשרי לממש חוזה שהוגדר עבור מחלקת עזר, או כאשר יש שימוש ברכיבים קיימים.
- בעיקרון, כשמסיימים עם ישות אחת, עוברים לבאה; אבל יתכנו גם תלויות מעגליות בין ישויות

איזה ישות לבחור עבור השלב הבא?

- יש שני סוגי בחירה
- בחירה שמיועדת לחקור חלק של המבנה שיש לגביו אי ודאות משמעותית; חלק שלא קשור אולי למה שכבר חקרנו, או חלק שיש לנו הרגשה שאיננו מבינים אותו לחלוטין
- או בחירה שמיועדת להשלים את התיכון של חלק מהמבנה, למשל בחירה של מחלקת עזר
- בדרך כלל כדאי לבחור ישויות או מחלקות שפיתוח עקרון המימוש שלהן יקטין משמעותית את אי הודאות לגבי מבנה התוכנה; זו אסטרטגיה שמזרזת גילוי טעויות
- אבל לפעמים כדאי לסיים חלק אחד לפני שעוברים לאחר, על מנת להגיע לרמת וודאות גבוהה לגבי היכולת לממש את החלק, ועל מנת למנוע העברת מאמץ הלוך ושוב בין חלקים

איך ממציאים מחלקות עזר

- המחלקות המרכזיות מתאימות בדרך כלל לישויות המרכזיות של מודל הנתונים, ולכן עם הקושי להמציא אותן התמודדנו כבר בשלב ניתוח הדרישות
- בשלב התיכון, עיקר הקושי הוא לפתח מחלקות עזר
- אפשר להשתמש במחלקות והפשטות קיימות, למשל הפשטות בשפת התכנות (מספרים בנקודה צפה, מחרוזות), מחלקות בספריה הסטנדרטית, ומחלקות שכבר פיתחנו או שאחרים פיתחו ואנחנו מודעים להן
- אפשר להשתמש בידע לגבי אלגוריתמים ומבני נתונים; ידע כזה עוזר לדעת איך לייצג ישויות ביעילות ואיזה בעיות אפשר לפתור ביעילות. כך גם לגבי תבניות תיכון.
- אפשר להשתמש בניסיון מתוכניות דומות, ובתבניות תיכון

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 חוזיהן? 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 נסגר

- 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