חלק 13

נושאים נוספים

נושאים נוספים

- מבוא להנדסת תוכנה - מה צריך לעשות בנוסף לכתיבת קוד ובדיקתו?
- למה תכנות מונחה עצמים?
- לחשוב עצמים.
- תבניות תיכון - לתיאור מבנים נפוצים, לפתור בעיות שחוזרות
- refactoring - לשיפור מבנה התכנית

מבוא להנדסת 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 בקצרה 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

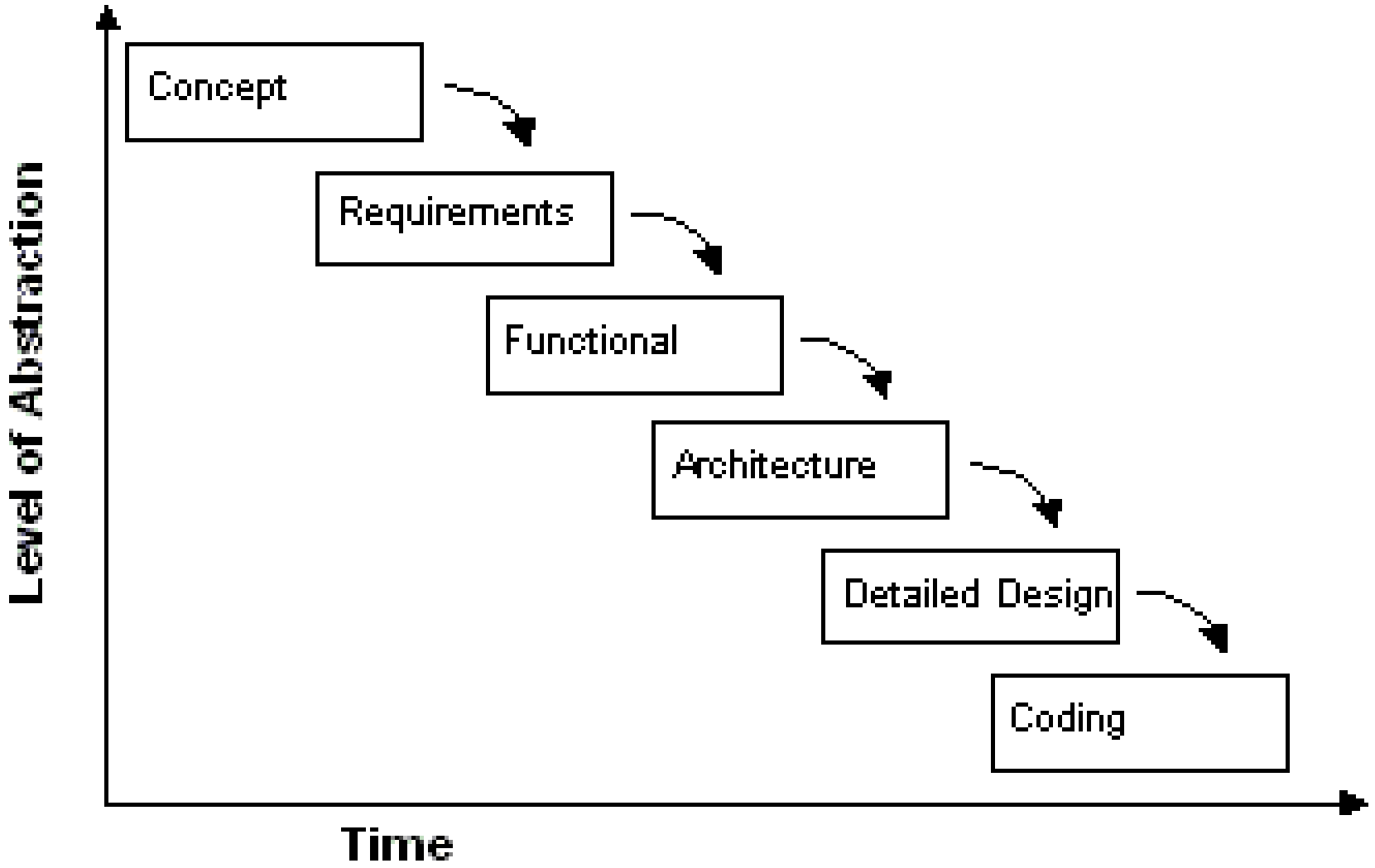
- ניתוח דרישות (requirements analysis)
- תיכון (design)
- מימוש ובדיקות
- בדיקות קבלה
- ייצור (production)
- תחזוקה ושינויים

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

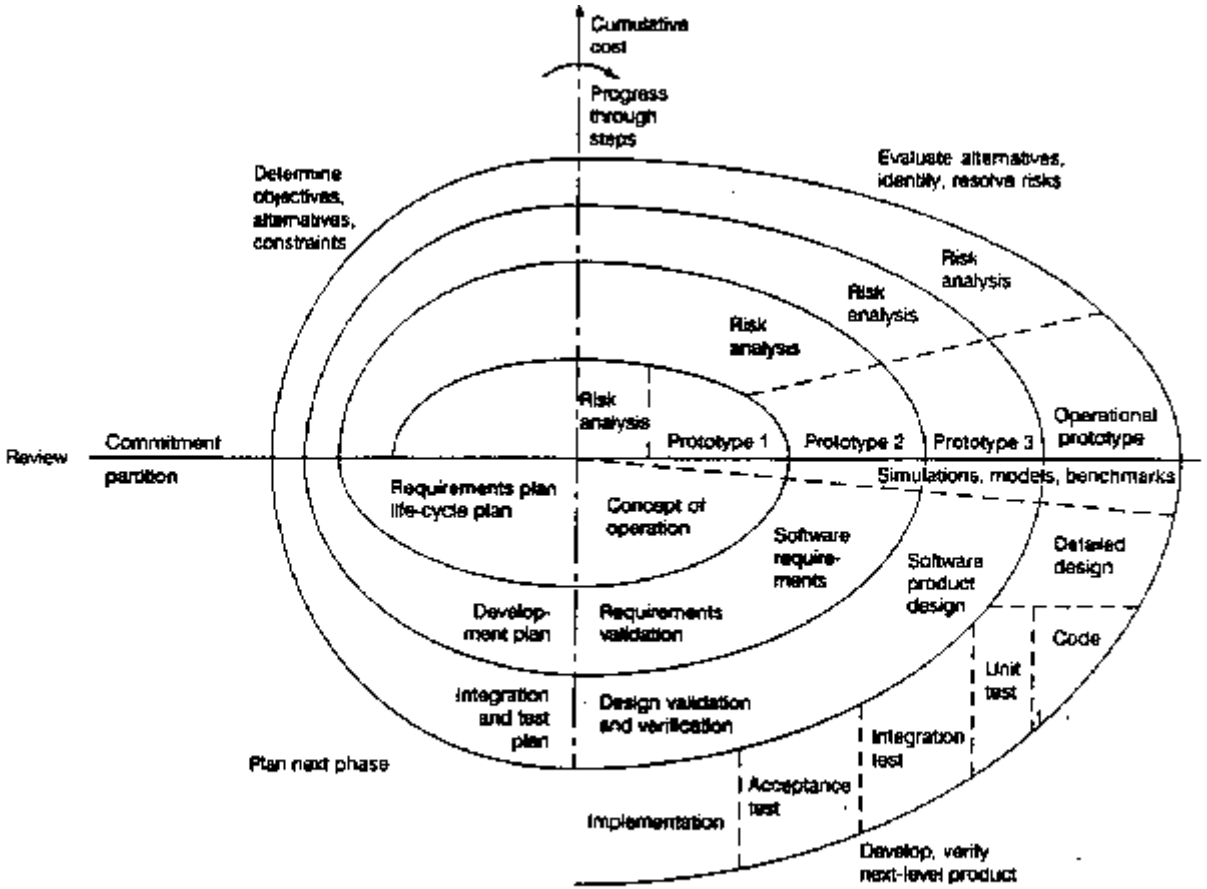
מפל או ספירלה?

- 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.

מודל מפל המים



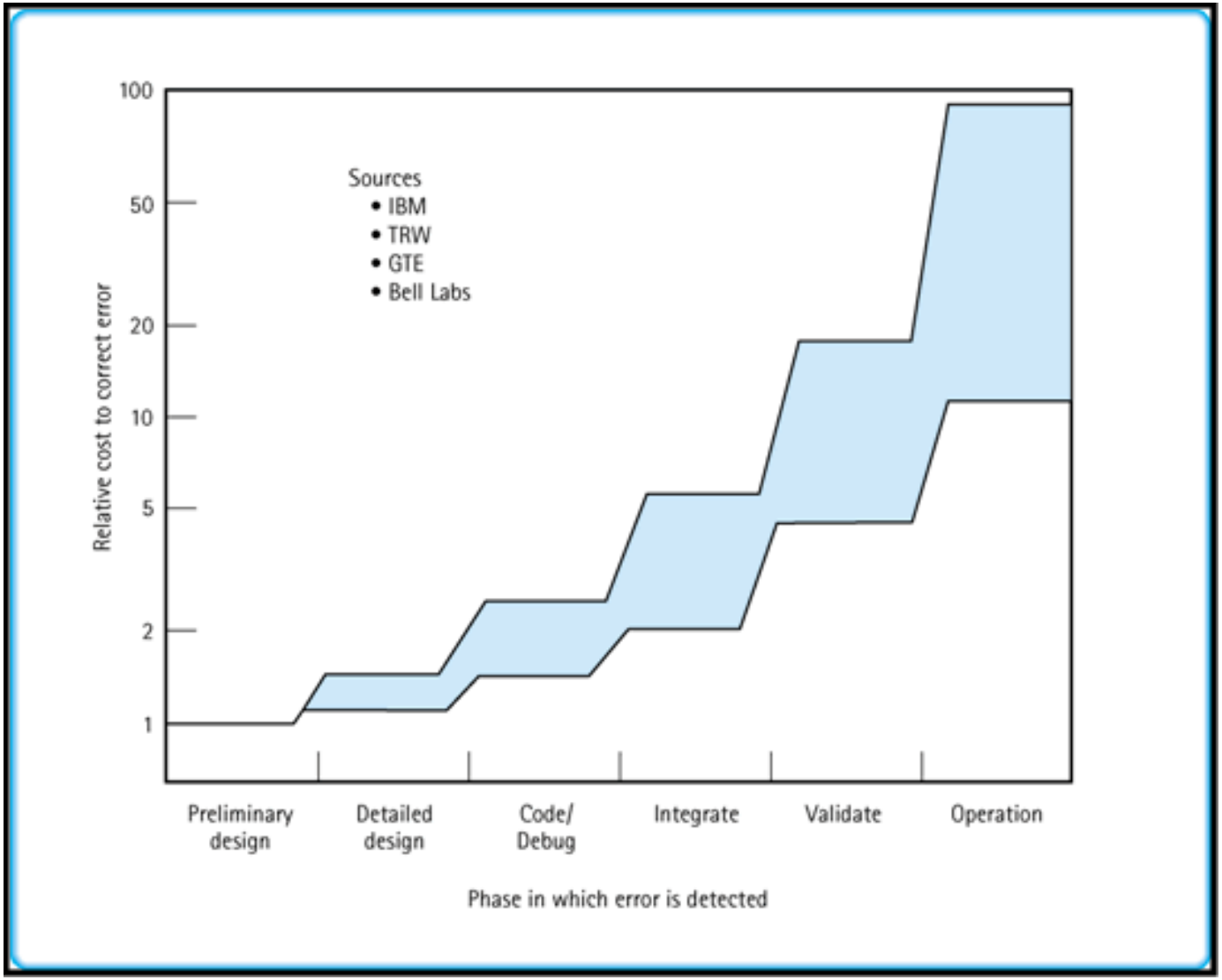
מודל הספירלה



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

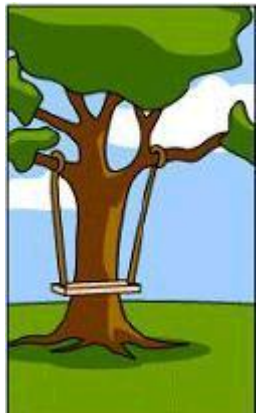
מחירן של טעויות

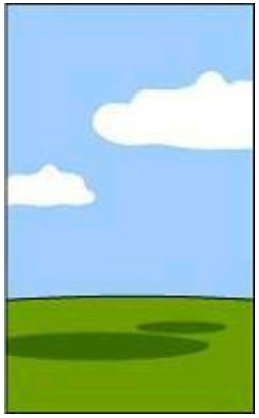
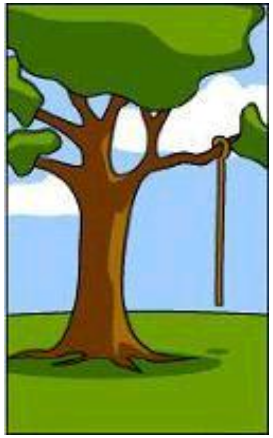
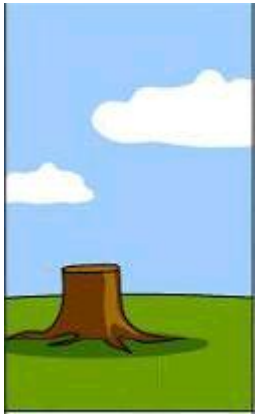


ניתוח דרישות

- 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?
- זהו שלב קשה, אולי הקשה ביותר בתהליך הפיתוח

Software Development Life Cycle

				
How the customer explained it	How the Project Leader understood it	How the Analyst designed it	How the Programmer wrote it	How the Business Consultant described it

 How the project was documented	 What operations installed	How the customer was billed	 How it was supported	What the customer really needed

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 אופיניים (או נציגים שלהם).
- את החשיבה על המבנה דוחים עד שיהיה ברור מה היא צריכה לעשות

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

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 טרויואלי)

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 לממשים 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

למה תכנות מונחה עצמים?

- למה תכנות מונחה עצמים?
  - עמידות לשינויים
  - מודולריות
  - שימוש חוזר בתוכנה

עמידות לשינויים

- מערכות תוכנה מתקיימות שנים רבות, במיוחד מערכות משובצות (לדוגמא מטוסים).
- עלות התחזוקה (תיקוני שגיאות מאוחרים, הוספת תכונות, התאמה לשינויי טכנולוגיה) היא יותר ממחצית העלות הכוללת של מחזור החיים.
- לכן השקעה נוספת בשלבים המוקדמים שתקטין את עלות התחזוקה עשויה להיות כדאית
- לכן חשוב שהתוכנה תהיה קריאה, ומודולרית.
- כמו כן התיכון צריך לאפשר שינויים עתידיים צפויים.
- אבל קשה כמובן לצפות.

מודולריות

- מודולריות היא תכונה חשובה של תוכנה.
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, ולשפר קריאות לצורך תחזוקה.
- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו, רציפות, הגנה
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה.
- ארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות.
- מכאן היתרון של פיתוח תוכנה מונחה עצמים.

שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה.
- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח.
- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון.
- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורות קוד
- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורליים.

לחשוב עצמים

- כדי לפתח קוד מונחה עצמים צריך "לחשוב עצמים".
- איך למצוא את העצמים (והמחלקות?) - להציע מועמדים, לפסול את חלקם (אם כל תכונותיהם כבר כוסו).
- שם של מחלקה יהיה בדרך כלל שם עצם.
- שם של מנשק יכול להיות שם תואר, למשל Comparable
- השגיאה הרווחת - להגדיר כמחלקה משהו שאינו כזה. אם אומרים "המחלקה XXX מבצעת ..." כנראה זה שרות.
- מחלקה עם שרות יחיד בדרך כלל לא צריכה להיות מחלקה
- ירושה - לא להתחיל את הקלסיפיקציה מוקדם מדי
- הפשטה מעורבת - כל השירותים של מחלקה צריכים להתייחס אל הפשטה יחידה, שברור מהי.

דוגמא undo

- בתוכנות עריכה ויישומים אחרים יש צורך לאפשר למשתמש לבטל את הפקודה האחרונה שבוצעה.
- דרישות מהיישום:
- שלא יהיה צורך לתכנן מחדש בכל פעם שנוסיף למערכת פקודה חדשה. מזה נובע שלא נוכל להתייחס ל undo ו redo כאל פקודות רגילות
- שצריכת הזיכרון תהיה סבירה. מזה נובע שלא נוכל לשמור את כל המצב לפני ביצוע כל פקודה
- שהפתרון יאפשר ביצוע undo למספר רמות כלשהו.

פתרון

- פקודה תהיה עצם.
- נגדיר מנשק Command

```
interface Command {
   abstract public void execute();
   abstract public void undo();
}
```

- כל סוג של פקודה יהיה מחלקה שממשת את Command עם שדות להחזיק את המידע הדרוש כדי לבטל את הביצוע.
- כדי להפעיל פקודה, ניצור עצם, נקרא ל `execute()` ונשמור את העצם.

פקודה לדוגמא

- עבור מחיקת שורה יש לזכור את מספר השורה ואת תוכנה. כאן מספר השורה נקבע בזמן יצירת העצם, ותוכן השורה נשמר בזמן ביצוע המחיקה.

```
public class LineDeletion
        implements Command {
  private int deletedLineIndex;
  private String deletedLine;
  public LineDeletion(int n) {
    deletedLineIndex = n;
  }
```

המשך מחיקת שורה

- מימוש השרותים:

```
   public void execute() {
      deletedLine =  line at position deletedLineIndex
      Delete the line in position deletedLineIndex
   }
   public void undo () {
      Put back deletedLine at position deletedLineIndex
   }
}
```

- באופן דומה נממש פקודות אחרות

קוד הלקוח

- במערכת עם GUI קוד הלקוח יופיע קרוב לוודאי בתוך listener מתאים, שיופעל כאשר ארוע מתאים יתבצע.
- למשל, האירוע שיגרום לביצוע הפקודה `LineDeletion` הוא בחירה של הפריט המתאים delete line מתפריט.
- קוד הלקוח ייראה כך

```
Command com = new LineDeletion(cursor);
com.execute();
lastCommand = com;
```

- כאשר `cursor` הוא משתנה שעוקב אחר השורה הנוכחית בטקסט, ו `lastCommand` אמור לזכור את הפקודה האחרונה שבוצעה, כדי לאפשר ביצוע undo

קוד הלקוח ל undo

- קוד הלקוח ל undo (שיימצא בתוך listener מתאים) יראה בערך כך:

```
if (lastCommand == null)
        write a message "nothing to undo"
    else lastCommand.undo();
```

מימוש undo במספר רמות

- כדי לממש undo במספר רמות צריך לשמור רשימה history של פקודות, ומצביע current למקום הנוכחי ברשימה.
- כאשר נתבקש לבצע undo , יבוצע (אם current מצביע לאבר קיים)

```
history.current.undo();
```

move current back one step

- בהקשר זה ניתן לראות כיצד יבוצע redo (באותה הנחה):

move current forward one step

```
history.current.redo();
```

תבניות תיכון - מוטיבציה

בחיי יום יום אנחנו מתארים דברים תוך שימוש בתבניות חוזרות:

- מכונית א׳ היא כמו מכונית ב׳, אבל יש לה 2 דלתות במקום 4.
- אני רוצה ארון כמו זה, אבל עם דלתות במקום מגרות.

גם בפיתוח תוכנה, אנחנו יכולים להסביר כיצד לעשות משהו ע״י התייחסות לדברים שעשינו בעבר, ובצורה כזאת להקל על התקשורת עם עמיתים.

- נאחסן את המבנה בעץ בינרי, ונבצע חיפוש לרוחב.

הגדרות מהספרות:

- "Design Patterns are recurring solutions to design problems you see over and over." [Alpert, Brown, Wof, 1998].
- "Design Patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." [Pree, 1994].
- "A pattern address a recurring design problem that arises in specific design situations and presents a solution to it." [Buschmann et al, 1996].
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [Gamma et al, 1993].

מהי תבנית תיכון?

- תבנית תיכון היא פתרון מקובל לבעית תיכון נפוצה בתכנות מונחה עצמים.
- תבנית תיכון מתארת כיצד לבנות מחלקות כדי לענות על הדרישה הנתונה.
- מספקת מבנה כללי שיש להשתמש בו כשממשים חלק מתכנית.
- לא מתארת את המבנה של כל המערכת.
- לא מתארת אלגוריתמים ספציפיים.
- מתמקדת בקשר בין מחלקות.
- מתארת נסיון מצטבר של מתכננים, שניתן ללמוד ועוזר לתקשורת בין מהנדסי תוכנה.

מקורות

- המושג נעשה פופולרי בעקבות הספר שמכונה GoF :

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements Of Reusable Object-Oriented Software. 1995.

- הגישה אומצה מעבודותיו של כריסטופר אלכסנדר, מתחום ארכיטקטורה של מבנים. הוא כתב :

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

רקע

- אבחנותיו של אלכסנדר רלבנטיות גם למערכות תוכנה.
- במקום לדבר על קירות ודלתות, אנו עוסקים בעצמים ומנשקים.
- נעשה מאמץ לקטלג תבניות תיכון כלליות, וגם תבניות שמתאימות לתחומי יישום מוגדרים.
- הקריטריון לקבלת תבנית תיכון ־ נעשה בה שימוש במספר יישומים אמיתיים.
- סוגים נוספים של תבניות, לתיעוד דרכים טובות לפתור בעיות מסוגים שונים (כולל למשל ניהול כוח אדם). Best Practice
- אנטי תבניות (Anti patterns) דברים שיש להימנע מהם.

מרכיבים עיקריים של תבנית תיכון

ארבעה מרכיבים חיוניים:

- שם התבנית. שימוש בשם אחיד ומקובל מסייע לתקשורת בין מהנדסי תוכנה, מעלה את רמת ההפשטה.
- הבעייה. מתי ניתן להשתמש בתבנית? איזה בעייה היא אמורה לפתור? יכול לכלול מבנה עצמים שהוא סימפטום לבעייה, או אוסף תנאים שצריכים להתקיים.
- הפתרון. מתאר את מרכיבי הפתרון, היחסים ביניהם, אחריות כל אחד ושיתופי פעולה. תבנית לפתרון, ולא מימוש קונקרטי.
- השלכות. תוצאות וקשרי גומלין (trade-offs) של שימוש בתבנית. חשוב לצורך הערכה של חלופות תיכון והבנת העלות והתועלת של התבנית. השלכות על יעילות, גמישות, יכולת הרחבה ויבילות (portability).

מושגים שקשורים לתבניות תיכון

תבניות תיכון מטפלות ברמת ביניים בין שני סוגי התבניות הבאים (עפ"י Buschmann and al):

- תבניות ארכיטקטוניות: צורת ארגון בסיסית של מערכת תוכנה שלמה. מספקת אוסף תתי מערכות קבועות מראש, חלוקת אחריות, כללים והנחיות לארגון היחסים ביניהן.
- תבניות קידוד, או idioms. תבנית ברמה נמוכה, ספציפית לשפת תכנות מסוימת. מתאר איך לממש היבט מסוים של רכיבים ויחסים ביניהם בעזרת המבנים ששפת התכנות מספקת.

תבניות תיכון אמורות להיות בלתי תלויות בשפה, אבל התבניות של GoF (וגם אחרות) מתייחסות במיוחד לשפות מונחות עצמים.

תבניות תיכון ו Frameworks

Design pattersn focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design … and implementation." [Coplien, Schmidt, 1995]

- Framework (OO Software) הוא אוסף מחלקות קשורות זו לזו, שניתן להרחיב ו/או ליצור מופעים, כדי לממש יישום.
- זה מגדיר ארכיטקטורת תוכנה ניתנת לשימוש חוזר, שמספקת מבנה והתנהגות כלליים של משפחה של יישומים ממוחשבים.

תבניות תיכון ו Frameworks - המשך.

- מספק מידע נוסף על שיתופי הפעולה בין המחלקות והשימוש בהן בתחום יישום מסוים.
- זה אינו יישום מלא שניתן להריץ, כי חסרים חלקים שיש להוסיף או להחליף כדי לקבל את הפונקציונליות הדרושה.
- תבניות תיכון יכולות לתאר את הקשרים בין מחלקות ב Framework
- Framework יכול להשתמש במספר תבניות תיכון

קלסיפיקציה של תבניות תיכון

הספר של GoF מציג 23 תבניות שמחולקות לשלוש משפחות לפי המטרה שלהן:

- תבניות ליצירה Creational: נוגעות לתהליך היצירה של עצמים.
- תבניות מבנה Structural: עוסקות בהרכבה של מחלקות ועצמים.
- תבניות התנהגות Behavioral: מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים ומחלקים אחריות.

קלסיפיקציה נוספת מתייחסת לתחום העיסוק של תבנית - מחלקות או עצמים.

בנוסף, ניתן להתייחס לקבוצות של תבניות שמופיעים בדרך כלל ביחד, או כאלה שמהווים חלופות שונות לפתרון בעיות דומות.

Model View Controller

- גישה לבניית מנשקים גראפיים שהוצעה בסמולטוק 80, ואומצה גם ע"י מפתחים בשפות וסביבות אחרות.
- יישום בנוי משלושה סוגים של עצמים:
  - מודל Model: עצם של היישום.
  - מראה View: הצגה של המודל על המסך.
  - בקר Controller: מגדיר את הדרך שבה המנשק מגיב לקלט של המשתמש.
- גישה זאת עדיפה על טיפוך בשלושת המרכיבים האלה ביחד.
- MVC משפר את הגמישות ואת השימוש החוזר.

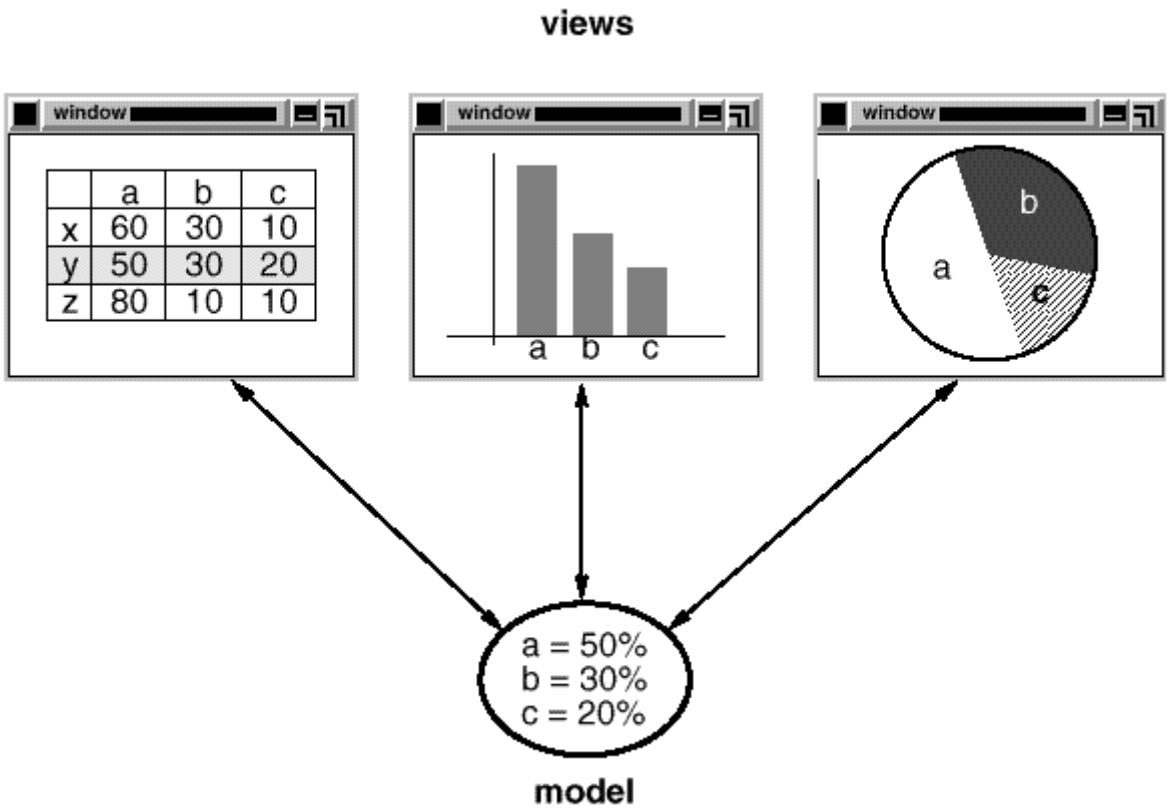
מודל לעומת מראה

- נניח בינתיים שאין בקר.
- למודל יכולים להיות מספר מראות.
- רוצים ליצור הפרדה בין המראה למודל באמצעות פרוטוקול:
- המראה שולח ראשית בקשה להתחבר למודל (להירשם כמנוי).
- כאשר המצב של המודל משתנה, הוא שולח לכל המנויים הודעה על השינוי.
- כל מראה מעדכן את עצמו.
- נשתמש בתבנית Observer (יש לה גם שימושים נוספים)

תבנית התיכון Observer

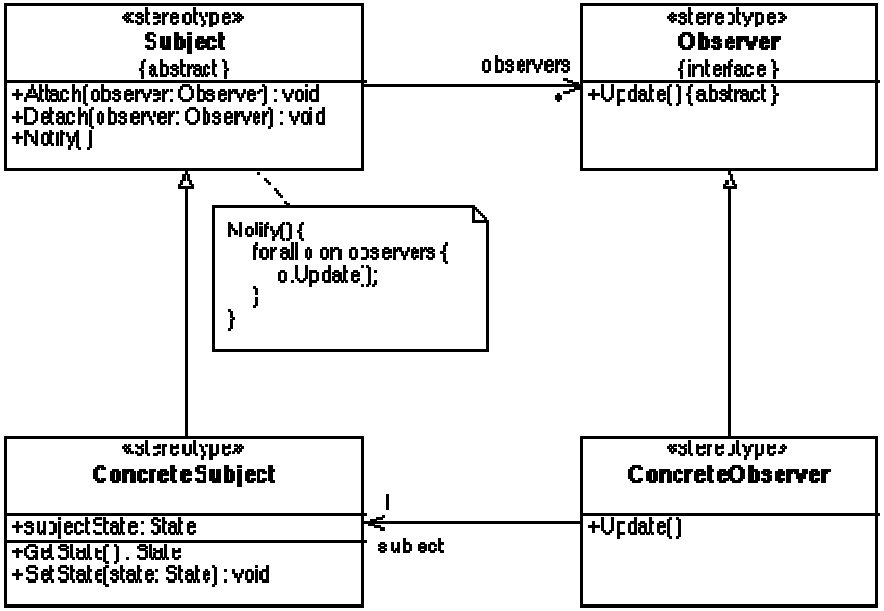
- מטרה: להגדיר תלות של אחד לרבים בין עצמים, כך שכאשר האחד משנה את מצבו, העצמים התלויים מקבלים הודעה ומתעדכנים אוטומטית. (תבנית התנהגות).
- מוטיבציה: לשמור על עקביות בין עצמים קשורים, בלי לגרום לצימוד חזק מדי.
- ישימות: כאשר
  - להפשטה יש שני הבטים, אחד תלוי בשניץ עצמים נפרדים מקלים על שינוי ושימוש חוזר.
  - שינוי בעצם אחד דורש שינוי במספר לא ידוע של עצמים.
  - עצם יכול להודיע לעצמים אחרים בלי להניח משהו עליהם.

תבנית התיכון Observer - מוטיבציה



תבנית התיכון Observer - מבנה

נשתמש בדיאגרמת מחלקות



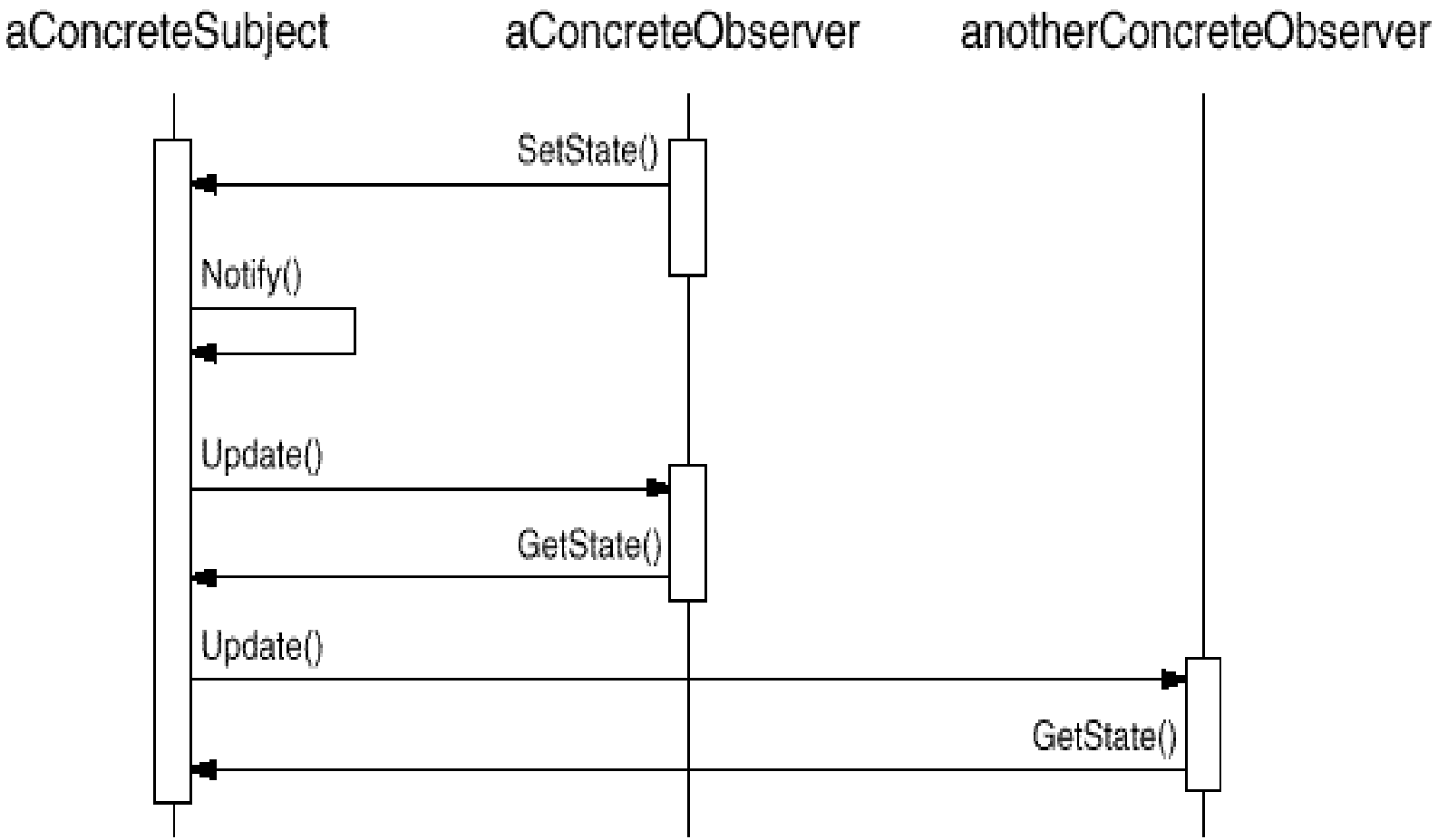
תבנית התיכון Observer - משתתפים

- Subject - שומר רשימה של Observers. מייצא פעולות להוספה והשמטה של Observers. כולל פעולת notify ששולחת הודעת עדכון לכל ה Observers.
- Observer - מייצא פעולה מופשטת של עדכון.
- ConcreteSubject - יורש מ Subject, שומר מצב, קורא ל notify כאשר המצב משתנה.
- ConcreteObserver - יורש מ Observer, מחזיק התייחסות ל ConcreteSubject, מממש את פעולת העדכון.

<u>הערה</u> (לגבי כל תבניות התיכון): שמות המשתתפים בתבנית מתארים את תפקידי המחלקות בתבנית. שמות המחלקות במערכת יהיו בדרך כלל שונים, ויבטאו את התפקיד הכללי יותר של המחלקה.

תבנית התיכון Observer - שיתוף פעולה

נשתמש ב sequence diagram



Model View Controller - המשך

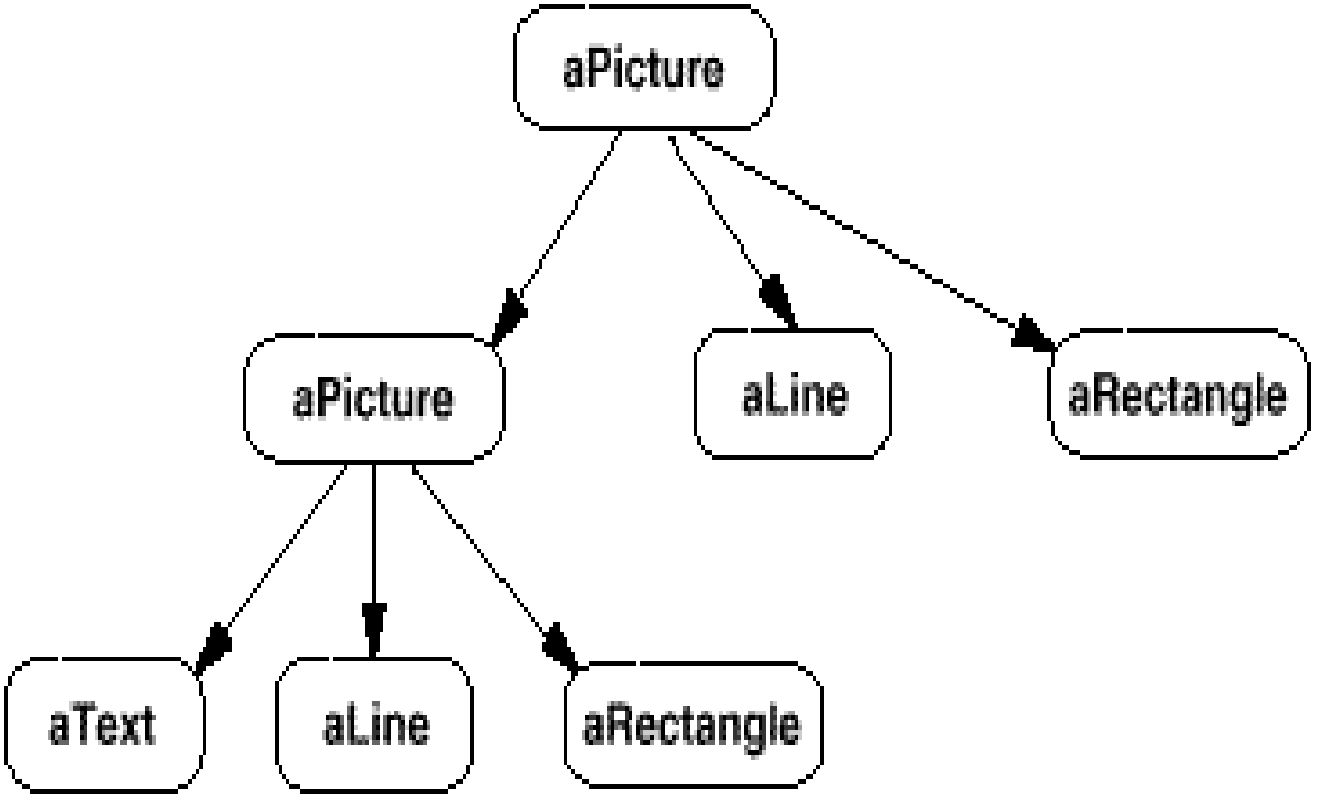
- מראות (Views) יכולים להיות מקוננים.
- לדוגמא, לוח בקרה מכיל כפתורים.
- מראה מורכב מתנהג בדיוק כמו עצם מראה רגיל.
- נשתמש בתבנית התיכון Composite (שימושית בהרבה הקשרים נוספים).

תבנית התיכון Composite

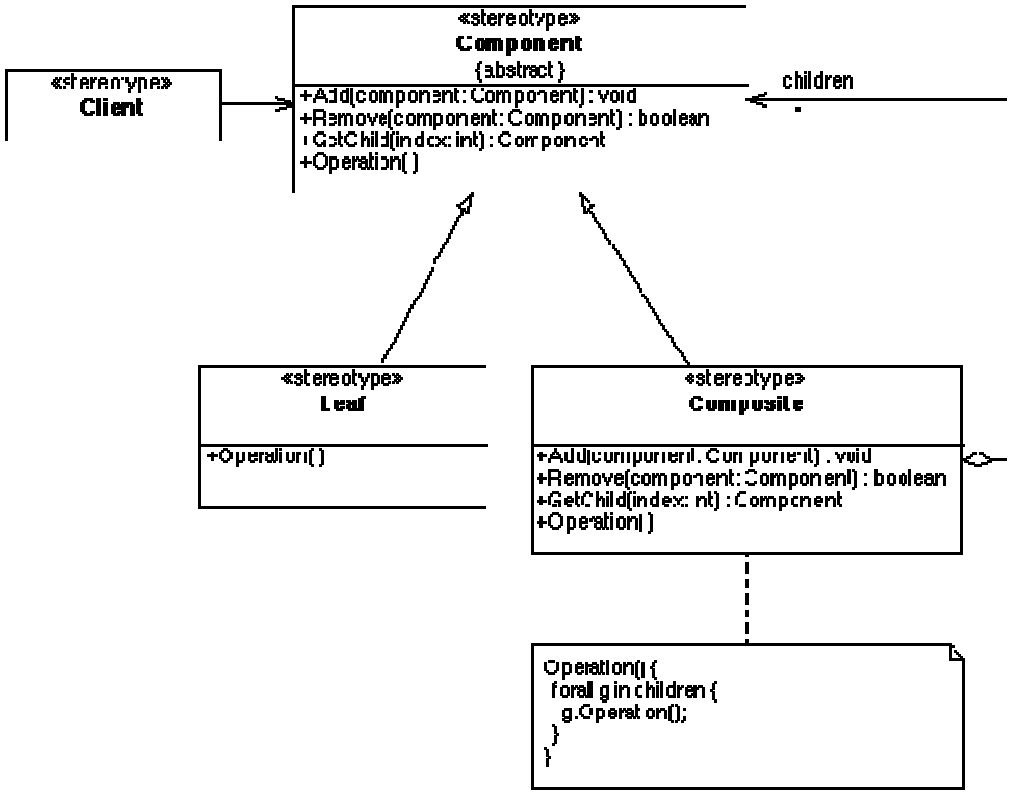
- מטרה: להרכיב עצמים למבני עץ שמייצגים את ההיררכיה של היחס חלק-שלם. לאפשר ללקוחות לטפל בעצמים בודדים ובהרכבות באופן אחיד. (תבנית מבנה).
- דוגמא: עצמים גרפיים.
- ישימות:
  - כאשר רוצים לממש יחס חלק-שלם.
  - כאשר רוצים לאפשר ללקוחות להתעלם מהבדלים בין עצמים מורכבים לעצמים בודדים.

תבנית התיכון Composite - דוגמא

דוגמא למבנה של ציור



תבנית התיכון Composite - מבנה



תבנית התיכון Composite - משתתפים

- Component - מגדיר מנשק לעצמים בהרכבה. מיישם ברירת מחדל לפעולות המשותפות. מגדיר מנשק לגישה לרכיבים הילדים.
- Leaf -(אחדים) - יורש מ Component. מייצג רכיבי עלה. מיישם התנהגות לעצמים הפרימיטיביים.
- Composite (אחדים?) - יורש מ Component. מייצג רכיבים מורכבים. מיישם פעולות על הילדים ע"י איטרציה על רשימת הרכיבים הכלולה בו.
- Client - לקוח של Component , מבצע פעולות על עצמים רק דרך מנשק ה Component .

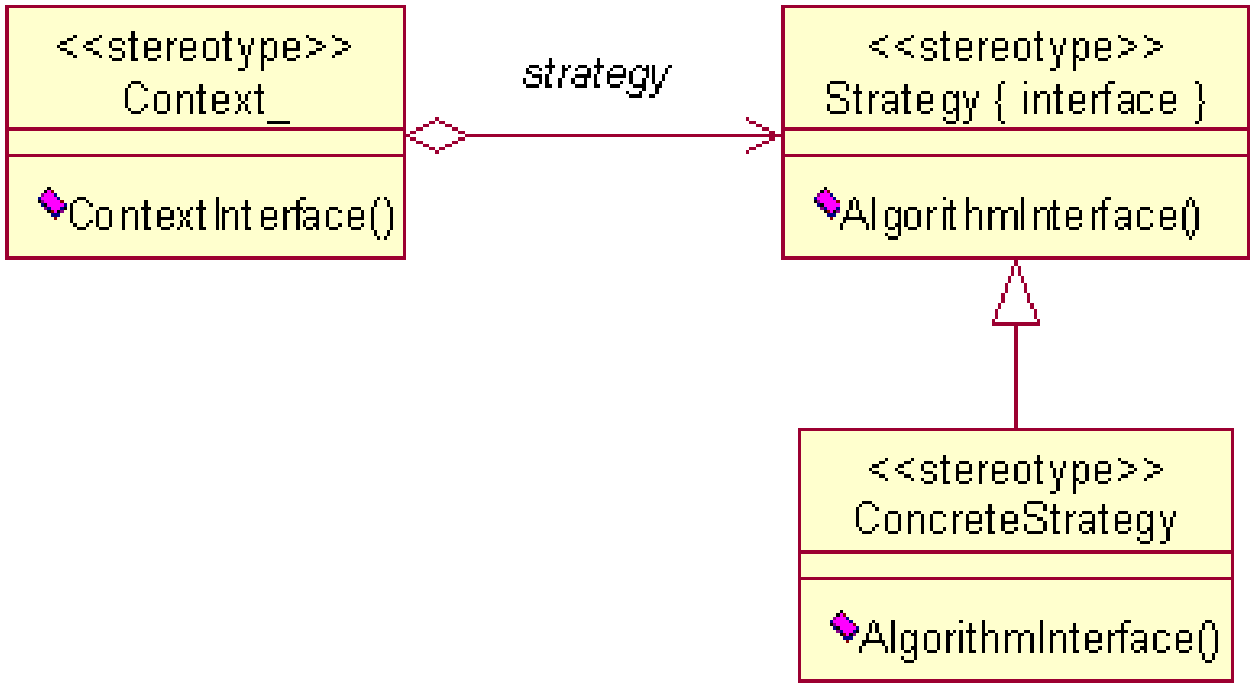
Model View Controller - המשך

- הבקר משמש להכמסה של מנגנון התגובה: איך מראה מגיב לקלט של המשתמש.
- מאפשר שנויים בצורת התגובה בלי לשנות את המראה.
- לדוגמא, החלפת כפתורים לפקודות בתפריט.
- שינוי כזה אפשרי אפילו בזמן ריצה.
- היחס בין מראה לבקר הוא דוגמא לתבנית התיכון Strategy .

תבנית התיכון Strategy

- מטרה: לספק הכמסה למשפחה של אלגוריתמים ולעשותם ברי החלפה. לאפשר לאלגוריתמים להשתנות באופן בלתי תלוי בלקוחות. (תבנית התנהגות).
- מוטיבציה:
  - הסרת האלגוריתם מפשטת את הלקוחות.
  - מדיניות שונה בזמנים שונים.
- ישימות:
  - כאשר אוסף מחלקות שונות זו מזו רק בהתנהגות.
  - יש צורך בחלופות (למשל שוני בזמן/זכרון).
  - האלגוריתם צריך להסתיר נתונים מורכבים.
  - להוציא התנהגות מותנית מהמחלקה.

תבנית התיכון Strategy - מבנה



תבנית התיכון Strategy - משתתפים

- Strategy - מגדיר מנשק משותף לכל האלגוריתמים הנתמכים.
- ConcreteStrategy (אחדים) - מספק מימוש של האלגוריתם בהתאם למנשק Strategy
- Context - לקוח של Strategy . בזמן ריצה, ההתייחסות היא לעצם מטיפוס של אחד מ ConcreteStrategy .

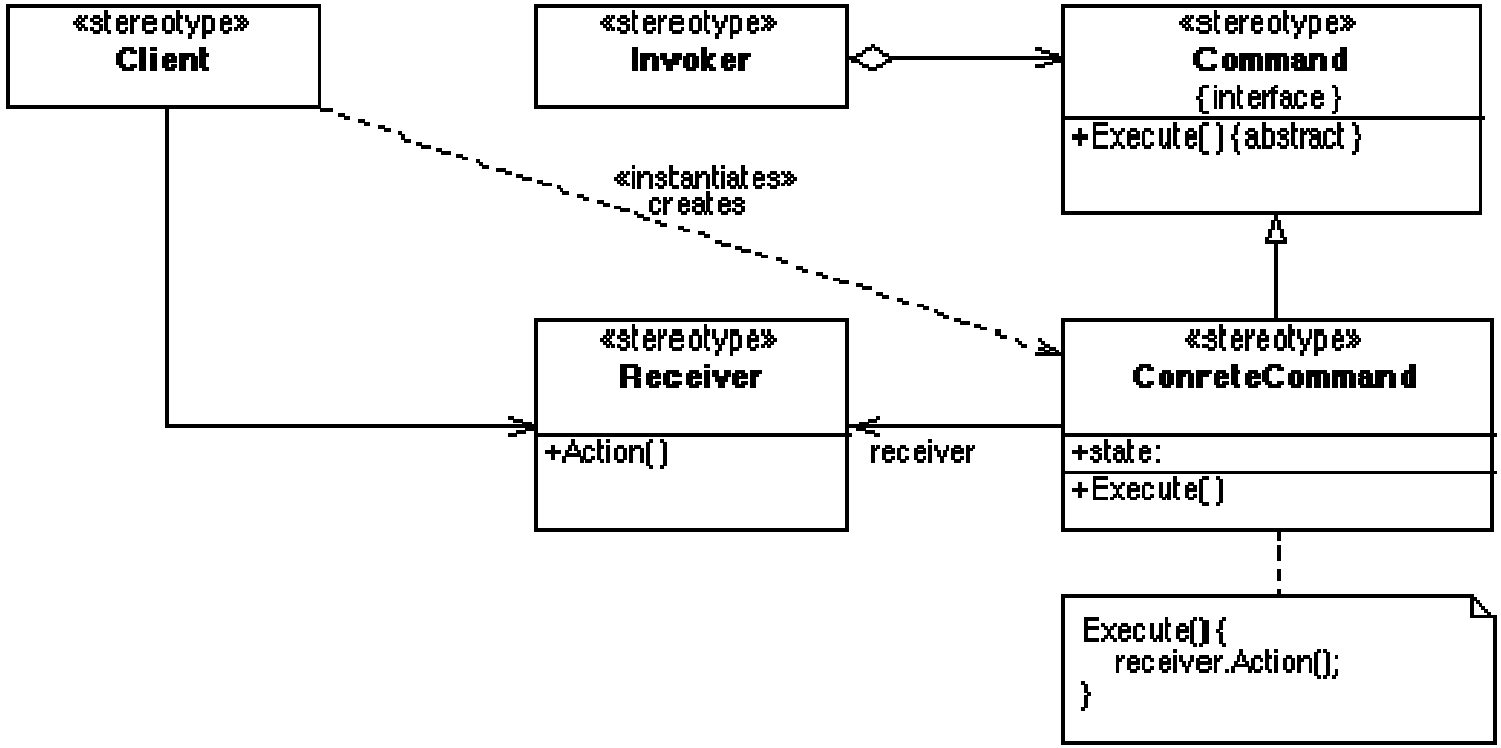
תבניות תיכון ליצירה

- תבנית התיכון Singleton נועדה להבטיח שיש רק אוביקט אחד ממחלקה מסוימת. ראינו איך ליישם זאת בג׳אוה.
- דברנו גם על בית חרושת, גם זאת תבנית תיכון ליצירה.
- למעשה יש כמה תבניות תיכון העוסקות בבית חרושת.(למשל תבנית התיכון Abstract Factory).

תבנית התיכון Command

- מטרה: להגדיר מנשק לצורך ביצוע פעולה. (תבנית התנהגות).
- ישימות:
  - פרמטריזציה של עצמים על פי הפעולה שיש לבצע. תחליף מונחה עצמים למנגנון של callback .
  - לאפיין, לצבור ולבצע בקשות בזמנים שונים, אולי בתהליך נפרד.
  - תמיכה ב undo ו redo
  - לשמור רשימת פעולות לביצוע כאשר מחלימים מקריסת מערכת.
  - לבנות מערכת סביב פעולות ברמה גבוהה הבנויות מפעולות פרימיטיביות (טרנסקציות).

תבנית התיכון Command - מבנה



תבנית התיכון Command - משתתפים

- Command - מגדיר ממשק לביצוע פעולה.
- ConcreteCommand - יוצר קשירה בין עצם Receiver לבין פעולה. מממש שרות execute ע"י הפעלת פעולה או פעולות של ה Receiver.
- Client - יוצר עצם ConcreteCommand וקובע את ה Receiver שלו.
- Invoker - מבקש מה Command לבצע את הפקודה המתבקשת.
- Receiver - יודע כיצד לבצע את הפעולות הדרושות לביצוע פקודה מבוקשת. כל מחלקה יכולה להוות Receiver.

סיכום תבניות תיכון

- פתרון מקובל לבעית תיכון נפוצה בתכנות מונחה עצמים.
- מבנה כללי שיש להשתמש בו כשממשים חלק מתכנית.
- נסיון מצטבר שניתן ללמוד ועוזר לתקשורת בין מהנדסי תוכנה.
- ראינו חלק קטן מהתבניות מהספר של GoF .
- יישום לדוגמא בשפת תכנות נתונה (למשל ג׳אוה).
- בעשור האחרון הוצעו כמה מאות תבניות, לא כולן חשובות באותה מידה.
- השימוש בתבניות חדר גם למושגים אחרים בהנדסת תוכנה, וגם בתחומים אחרים.
- תבניות ואנטי תבניות.

refactoring

- refactoring הוא תהליך של שינוי תוכנה כך שהתנהגותה החיצונית לא תשתנה, אך המבנה הפנימי שלה ישתפר.
- לנקות ולשפר את הקוד בלי להכניס לשגיאות.
- "שיפור התיכון אחרי שהקוד נכתב" סותר לכאורה את העקרונות שמנחים פיתוח תוכנה.
- אבל מכיר בעובדה שבמשך הזמן, שינויים בקוד (למשל להוספת תכונות) גורמים לכך שהמבנה נפגע ומסתבך.
- ב refactoring מבצעים בכל פעם שינוי קטן, טרנספורמציה שמשמרת נכונות (כלומר לא משנה את ההתנהגות החיצונית).
- לאחר כל שינוי יש לבדוק היטב שהשינוי היה נכון - להריץ את אוסף הבדיקות שצברנו.

מקורות

- האנשים שזיהו את חשיבות הרעיון :

Ward Cunningham, Kent Beck

- ספר:

Martin Fowler, Refactoring, Improving the Design of Existing Code, Addison Wesley 2000. (2nd edition 2005).

- קשור ל Extreme Programming - תהליך פיתוח חדש יחסית, ששם דגש על פיתוח אבולוציוני, גירסאות תכופות, בדיקות, קשר הדוק עם הלקוח, קידוד בזוגות, ...
- משפחת תהליכי פיתוח Agile Software Development

למה refactoring ?

- לשפר את תיכון התוכנה - אחרת מבנה המערכת נשחק עם הזמן.
- לעשות את התוכנה קריאה יותר - הקריאות חיונית למתחזקים.
- לעזור למצוא שגיאות - קשה למצוא שגיאה בקוד מסורבל.
- לזרז את כתיבת הקוד - כל השיפורים הללו יקטינו את הזמן שיידרש בהמשך.

מתי לעשות refactoring ?

- כאשר מוסיפים פונקציונליות למערכת - "אם הקוד היה כתוב כך, היה קל יותר להוסיף את הפעולה".
- כאשר צריך למצוא שגיאה - בכל פעם שמסתכלים על קוד ומתקשים להבין אותו יש לבדוק האם ניתן לשפר.
- תוך כדי סקר קוד (Code review)
- באופן כללי, כל פעם שמגלים קוד ש"מריח לא טוב" Bad code smells לדוגמא: כפילות בקוד, שרות ארוך מדי, מחלקה גדולה מדי, רשימת פרמטרים ארוכה, סימפטומים של צימוד חזק מדי בין מחלקות, ...

קטלוג של refactorings

- הספר של Fowler כולל קטלוג של refactorings שכל אחד כולל שם, סיכום קצר, מוטיבציה, תהליך השינוי, ודוגמא.
- חלק מה refactorings ניתנים לאוטומציה, וכן אקליפס (וגם כלי פיתוח אחרים) תומכים במספר refactorings
- הכלים מאפשרים לראות כיצד ייראה הקוד אחרי השינוי, ולהחליט (וכן לבטל שינוי שנעשה).
- הכלים יכולים לציין מתי מובטח שהשינוי נכון (כלומר לא משנה התנהגות).
- אפילו דוגמא פשוטה - שינוי שם של שרות - קשה מאד לשינוי ידני ללא שגיאה. (שינוי גלובלי בעטרך טקסט לא יהיה נכון בהכרח).

דוגמאות מקטלוג ה refactorings

- extract method / inline method
- Introduce Explaining Variable
- Move method/Field
- Rename method
- Add/Remove Parameter
- Pull up/Push down Field/Method
- Extract Subclass/Superclass/Interface
- Collapse Hierarchy
- Replace Inheritence with Delagation / vice versa