מגישים:

ינקל ז'אוי- 324523075

דן יוסף אביטן- 203109996

נופר דוכן- 322599424

בס"ד

עבודת מבוא- Refactoring:

תקציר

בפיתוח תוכנה, Refactoring הוא תהליך של שיפור המבנה הפנימי או העיצוב של הקוד הקיים מבלי לשנות את ההתנהגות החיצונית שלו, על פי מרטין פאולר שהציג מושג זה בספרו: " Refactoring: Improving the Design of Existing Code". [5]

ה- refactoring למעשה זהו תהליך שאנו עושים בכדי להפוך את הקוד לקל יותר להבנה, לשינוי ולתחזוקה, תוך הפחתת הסיכון להחדרת באגים חדשים.

אנו בעצם נבצע אוסף של פעולות, שלב אחר שלב בכדי להגיע לתוצאה סופית של refactoring (שיהיה קשה להשלים את התהליך עד הסוף מכיוון שתהליך ה refactoring מתבצע בקביעות והוא לוקח המון זמן).

בעבודה שלנו נציג 2 שיטות שונות לביצוע Refactoring וכלי ליישומו.

השיטות שבחרנו להתעסק בהן:

- Composing method
- Moving Features Between Objects

מבוא על עקרון ה- Refactoring [1]

כמו שאמרנו, בפיתוח תוכנה, Refactoring הוא תהליך של שיפור המבנה הפנימי או העיצוב של הקוד הקיים מבלי לשנות את ההתנהגות החיצונית שלו, על פי מרטין פאולר שהציג מושג זה בספרו: " Refactoring: Improving the Design of Existing Code". [1]

ישנן מספר סיבות שיובילו אותנו לביצוע refactoring : [1]

- אחת הסיבות היא לשפר את עיצוב הקוד. זה יכול לכלול פישוט שיטות, הסרת כפילות ושיפור הארכיטקטורה הכוללת של הקוד. [1]
- סיבה נוספת, היא להקל על הוספת תכונות חדשות או שינוי קיימות. כאשר הקוד מובנה היטב, קל יותר לבצע שינויים מבלי להכניס באגים חדשים או לשבור את הפונקציונליות הקיימת. [1]
- ניתן להשתמש ב-refactoring גם כדי לתקן באגים או לטפל בבעיות ביצועים. על ידי שיפור המבנה הפנימי של הקוד, אנו יכולים לעתים קרובות לבטל צווארי בקבוק בביצועים או למנוע התרחשות של באגים. [1]

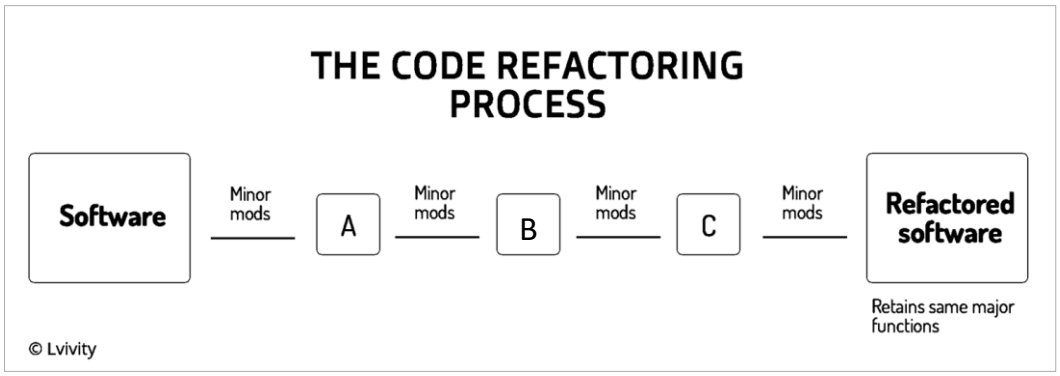
נשים לב כי Refactoring צריך להיעשות באופן קבוע, כחלק מתהליך הפיתוח, כדי למנוע הצטברות חוב טכני. וכי חשוב גם לבצע refactoring code בעת ביצוע שינויים או הוספת תכונות חדשות כדי למנוע מהחוב הטכני לגדול. [1]

ישנן מספר דרכים לשחזר קוד, אך הגישה הטובה ביותר היא ביצוע צעד אחד בכל פעם ובדיקה לאחר כל שינוי. בדיקה מבטיחה שהפונקציונליות העיקרית תישאר, אבל הקוד משופר באופן צפוי ובטוח - כך שלא יוצגו שגיאות בזמן בניה מחדש של הקוד. [4]

מספר יתרונות ל- refactoring המקבילים לסיבות ויתרונות נוספים: [1]

- שיפור איכות הקוד- ביטול קוד מיותר ומשוכפל משפר את הקריאות וההבנה של הקוד, מה שמקל על התחזוקה והעדכון שלו. [1]
- הפחתת חוב טכני (כמות העבודה שצריך לעשות בעתיד)- שחזור קוד באופן קבוע שומר על חוב טכני ברמה ניתנת לניהול ומפחית את הזמן והמאמץ הנדרשים לתחזוקת הקוד. [1]
- שיפור עיצוב הקוד- משפר את המבנה הפנימי של הקוד, מה שהופך אותו למודולרי וקל יותר להוספת תכונות חדשות ושינויים ללא השפעה על המבנה הבסיסי. [1]

- זיהוי ותיקון באגים- פישוט שיטות וביטול דברים מיותרים מפחיתים את הסיכוי להחדרת באגים חדשים ומסייעים בזיהוי ותיקון באגים קיימים, מה שמשפר את אמינות ויציבות התוכנה. [1]
- חיסכון בזמן ובכסף- עיבוד מחדש של הקוד יכול לחסוך בטווח הארוך על ידי הפחתת החוב הטכני, שיפור איכות הקוד והקלה על תחזוקה ועדכון, וכן צמצום הזמן והמאמץ הנדרשים להוספת תכונות חדשות. [1]



[2]

<u>סקירה ספרותית:</u>

בעבר, לא היה נהוג לבצע ארגון מחדש של הקוד במהלך הפיתוח. דוגמה לכך היא קובצי CVS משנת 1984, שאינם כוללים תיעוד של שינויים או העברות של קבצים ותיקיות. [6]

ארגון הקוד מחדש היה ידוע מזה שנים רבות, וויליאם פ. אופדיק נחשב לראשון שבחן נושא זה לעומק. [6]

עם זאת מרטין פאולר מציג את עקרון ה- refactoring, בספרו "Refactoring: Improving the Design of Existing Code" - והספר נחשב לספר סמכותי ובעל השפעה רבה שמרבים לפנות אליו ולהשתמש בו כמקור ידע מרכזי בנושא.

השימוש הראשון המוכר במושג "Refactoring" בספרות היה בספטמבר 1990, במאמר של וויליאם פ. אופדיק וראלף א. ג'ונסון. אופדיק השתמש במונח זה שוב בתזה שלו שפורסמה ב-1992. [6]

התזה מפרטת שלוש פעולות ארגון מחדש מורכבות במיוחד: הוספת היררכיית ירושה, התמקדות בהיררכיית ירושה ושימוש בדרך שבה אנו מייצגים את הקשרים בין מחלקות (או אובייקטים) בתוכנית מונחת עצמים. [6] פעולות אלו מפורקות לפעולות פשוטות יותר, והכוח שלהן נבחן מהיבטים של אוטומציה- היכולת לבצע את פעולות הארגון מחדש באופן אוטומטי על ידי כלי תוכנה, מבלי שהמפתחים יצטרכו לעשות זאת ידנית, זה כולל בדיקת התנאים המוקדמים של הפעולות והבטחה שהן מבוצעות בצורה נכונה שמשמרת את ההתנהגות של התוכנית. [6]
ושימושיות בתמיכה בתכנון- עד כמה פעולות הארגון מחדש עוזרות לתכנון טוב יותר של התוכנית. זה כולל שיפור הקריאות, התחזוקה והשימוש החוזר בקוד, כך שהמבנה של התוכנית יהיה ברור ויעיל יותר. [6]

Refactoring מתבצע על 2 פרדיגמות של תכנות:

- תכנות פרוצדורלי: גישה שמתמקדת בכתיבת רצפים של פעולות או פקודות לביצוע משימות.
- תכנות מונחה עצמים(OOP) : גישה שמתמקדת ביצירת עצמים שמכילים נתונים ופונקציות.

קשה יותר לבצע את תהליך הריפקטורינג על תכנות מונחה עצמים אנו יכולים לראות זאת כאשר בתכנות פרוצדורלי חלק מתהליך הריפקטורינג זה שינוי שם לפונקציה, יצירת משתנה ואילו בתכנות פרוצדורלי זה הוספת היררכיה חדשה ושימוש בקשרים בין מחלקות. [6][3]

2 שיטות ליישום עיקרון ה-refactoring:

נוכל לראות כי לעקרון ה-refactoring יש כמה שיטות כגון: [1][4]

- Red-Green-Refactor
- Refactoring by Abstraction
- **Composing method**
- Organizing Data
- Simplifying Conditional Expressions
- Simplifying Method Calls
- **Moving Features Between Objects**
- Preparatory Refactoring

לכל שיטה ישנם מספר שלבים הניתנים לביצוע, וכך כל שיטה שמה דגש על כמה מיתרונות ה-refactoring ומביאה אותם לידי ביטוי בקוד. [1][4]

אנו בחרנו להשתמש בשתי השיטות המודגשות, ובמהלך הסמסטר נתמקד בשיטת Composing method.

Composing method

שיטה זו משמשת בעיקר להבנת השיטות הארוכות, היא מתמקדת בפירוק קטעי קוד כגון מחלקות או פונקציות גדולות לרכיבים קטנים יותר שתפקידם ומשמעותם מובנת יותר וקלה יותר לניהול, כך נעשה שיפור בקריאות הקוד, ביכולת התחזוקה של הקוד וביכולת הבדיקה של הקוד וזאת על ידי כך שלכל רכיב תפקיד ברור ואחריות אחת ויחידה. באמצעות זאת, הקוד ניתן להבנה יותר בקלות ולשינוי והרחבה עתידיים. [1][4]

שלבים לביצוע Composing method: [3]

1. Extract Method

   העברת קוד שניתן לקבץ יחד למתודה או פונקציה חדשה ונפרדת ולאחר מכן החלפת הקוד הישן בקריאה למתודה \ הפונקציה החדשה – כך הקוד הופך קריא ומובן, כפילויות מוסרות והופכות לקריאה לפונקציה, יש סבירות נמוכה יותר לשגיאות. [3]

2. Inline Method

   כאשר גוף הפונקציה ברור יותר מהפונקציה עצמה (לעיתים קוד זה של גוף הפונקציה קצר ממש) נעדיף להחליף קריאות לפונקציה זו, בקוד גוף הפונקציה ולאחר מכן נמחק את הפונקציה – כך הקוד נעשה פשוט יותר על ידי צמצום פונקציות מיותרות בקוד. [3]

3. Extract Variable

   הצבת ביטויים שקשה להבין או חלק מהם במשתנים נפרדים כך שמשמעות הביטויים ותוצאתם תהיה ברורה יותר בקוד. לאחר הצבה זו יהיה ניתן להשתמש במהלך הקוד במתנים אלו המבטאים את הביטויים שקודם היה קשה לנו להבין. [3]

4. Inline Temp

   כאשר יש משתנה המבטא ביטוי קצר ופשוט להבנה לפעמים נעדיף להחליף קריאות למשתנה בביטוי עצמו כך זה יכול להועיל במעט לקריאות הקוד. [3]

5. Replace Temp with Query

   במקום להציב ביטוי במשתנה מקומי(בו לא ניתן להשתמש בפונקציות אחרות) אך נעשה שימוש בביטוי זה בכמה פונקציות, נעביר את הביטוי לפונקציה נפרדת וממנה נחזיר את תוצאה הביטוי, כך שכל פעם שנרצה להשתמש בביטוי, נקרא לפונקציה שיצרנו עם ערכים שונים. [3]

6. Split Temporary Variable

כאשר יש משתנה מקומי שיש לו מספר משמעויות בקוד [3] (לדוגמא: משתנה num שאחראי על תוצאת חישוב שטח משולש ולאחר מכן בהמשך הקוד אותו משתנה num מבטא תוצאת חישוב של נפח קובייה) נרצה להשתמש במשתנים שונים (בעלי שם עם משמעות) כדי לבטא ערכים שונים[3] (לדוגמא: משתנה area יבטא את חישוב שטח המשולש ומשתנה volume את חישוב נפח הקובייה.) כך יהיה ניתן להקל על תחזוקת הקוד. [3]

7. Remove Assignments to Parameters

כאשר אנחנו מקבלים ערך כפרמטר בפונקציה, לעיתים נרצה להקצות משתנה מקומי (בעל שם עם משמעות) במקום השימוש בפרמטר בדומה לסיבות בשלב הקודם. [3]

8. Replace Method with Method Object

כאשר ישנה פונקציה שהיא מסורבלת מבחינה לוגית, בעלת מספר תפקידים שונים, ארוכה או שהמשתנים הלוקלים שלה מסורבלים מדיי עד שלא ניתן להשתמש ב-extract method, נהפוך את הפונקציה למחלקה נפרדת, כך שהמשתנים המקומיים שלה יהפכו לשדות של המחלקה ולאחר מכן נוכל לפצל את הפונקציה למספר מתודות בתוך אותה מחלקה. [3]

9. Substitute Algorithm

לפעמים פונקציה עלולה להכיל המון בעיות כך שעדיף להתחיל מחדש את הטיפול בה מאשר לחפש מה לתקן. במידה ונמצא אלגוריתם פשוט ויעיל יותר לביצוע תפקידה של הפונקציה, פשוט נחליף את האלגוריתם שהיווה בעיות בפונקציה, באלגוריתם החדש שמצאנו. [3]

Moving Features Between Objects

בשיטה זו מתבצע העברת בטוחה של פונקציונליות בין מחלקות או אובייקטים ויצירת מחלקות חדשות, כדי לשפר את העיצוב הכולל של הקוד. על ידי קיבוץ של פונקציונליות קשורה יחד והפרדת פונקציונליות לא קשורה, מפתחים יכולים לשפר את המודולריות ואת יכולת התחזוקה של הקוד. [1][4]

שלבים לביצוע Moving Features Between Objects [3]

1. Move Method

לעיתים מתודה נכתבה במחלקה A אך במחלקה B נעשה שימוש רב יותר במתודה זו, לכן ניצור מתודה חדשה במחלקה B וגוף המתודה יהיה העתקה של גוף המתודה שנכתבה במחלקה A, את הקוד של המתודה ממחלקה A נחליף בקריאה למתודה שיצרנו במחלקה B. [3]

2. Move Field

בדומה לשלב הקודם, אך כאן ההתעסקות נעשית בשדה המחלקה. אם שדה נכתב במחלקה A, אך נעשה בו שימוש רב יותר במחלקה B, ניצור שדה במחלקה B ונפנה אליו את כל המשתמשים בשדה שבמחלקה A. [3]

3. Extract Class

כאשר מחלקה אחת מבצעת פעולה של 2 מחלקות ניצור מחלקה חדשה אליה נעביר אליה שיטות ושדות רלוונטיים על פי השלבים הקודמים, ניצור קשר בין המחלקה הוותיקה לחדשה – כך הקוד שלנו מובן יותר כי כל מחלקה אחראית על תחום משלה. [3]

4. Inline Class

לעיתים יכולה להיות מחלקה שלא מבצעת כמעט כלום וניתן לראות שאפשר לוותר עליה כי לא אחראית או עתידה להיות אחראית על דבר מסוים, (זה יכול לקרות לאחר שהעבירו תכונות של מחלקה זו לאחרות) לכן נעדיף להעביר את השדות\מתודות של מחלקה זו למחלקה אחרת ונמחק אותה. – דבר שמפנה זיכרון הפעלה במחשב - ורוחב פס בראש. [3]

5. Hide Delegate

כדי שיהיה לנו קל יותר לבצע שינויים, קוד הלקוח צריך לדעת כמה שפחות על היחסים בין האובייקטים. ולכן כאשר לקוח מקבל אובייקט B משדה או מתודה של אובייקט A ואז הלקוח קורא למתודה של אובייקט B, ניצור מתודה חדשה ב A שמשתמשת במחלקת B וכך נגרום למשתמש להכיר רק את המחלקה A. [3]

6. Remove Middleman

כאשר יש מחלקה שיש לה הרבה פונקציות שלא מבצעות דבר חוץ מקריאות לפונקציות ממחלקות אחרות זה יכול ליצור קושי בתחזוקה וכו', נתמודד עם הבעיה על ידי מחיקת הפונקציות האלה וקריאה שלהם מתוך אובייקט של המחלקה בהתאמה. [3]

7. Introduce Foreign Method

כאשר נעשה שימוש במחלקת עזר שלא מכילה את המתודה שאנו צריכים ואין ביכולתנו להוסיף אותה מסיבות כאלה ואחרות. אז נרצה להוסיף את המתודה הזאת למחלקה (A לצורך הדוגמא), ונשתמש במחלקת עזר כפרמטר במתודה שיצרנו במחלקה A וכך תהיה גישה לכל השדות של מחלקת העזר ובתוך המתודה יהיה ניתן לעשות כל מה שנרצה כאילו היא חלק ממחלקת השירות. [3]

8. Introduce Local Extension

כאשר יש לך מחלקת שירות שמשום סיבה אין ביכולתך להוסיף לה את המתודות הנצרכות לך, תוכל ליצור מחלקת בת המכילה את המתודות הרצויות ובכך להרחיב את הפונקציונליות של המחלקה המקורית. [3]

כלי ליישום ה- Refactoring [1]

זהו כלי שעוזר למפתחים לבצע משימות refactoring ביעילות רבה יותר ועם סיכון מופחת להחדרת שגיאות, הכלים עוזרים להשתמש בשיטות שונות בדרך בטוחה, הכלים השונים הם היכולות של סביבות העבודה להודיע על שינוי קוד לא נכון מבחינת syntax למשל או נראות הקוד כך למשל visual studio הוא סביבת עבודה המספר כלים ליישום הריפקטורינג, הוא מודיע על שגיאה בקוד או על דרך יותר אפקטיבית לכתוב לוגיקה של קוד. [1]

ניתן להיעזר גם בתוסף SonarLint (הוא פועל בתוך ה IDE-סביבות פיתוח משולבות ומשתמש במנוע האנליזה של SonarQube כדי לסרוק את הקוד. כאשר המפתח כותב קוד, SonarLint מנתח אותו בזמן אמת ומספק דוחות ותגובות על בעיות פוטנציאליות בקוד. הכלי מציג את הבעיות שנמצאו בצורה ברורה ומציע תיקונים אוטומטיים כאשר אפשרי.) [1]

מסקנות

Refactoring מתבצע בשלבים קטנים ובאופן קבוע כך שהוא מביא לנו יתרונות רבים כגון: קוד קריא יותר, קלות בתחזוקה, והפחתת חוב טכני.

יש כמה שלבים בביצוע refactoring וכמה שיטות המתחלקות ל-2 פרדיגמות מרכזיות בתכנות: פרוצדורלי ו-OOP

מומלץ מאוד לעשות refactoring בכל פעם שאנו נוגעים בקוד ובסופו של דבר הקוד שלנו יהיה גם קל להבנה.

ביבליוגרפיה:

[1] SonarSource. (n.d.). Refactoring. SonarSource.
https://www.sonarsource.com/learn/refactoring/#sonarlint

[2] Lviv IT Center. (n.d.). What is Code Refactoring? Lviv IT Center.
https://lvivity.com/what-is-code-refactoring

[3] Fowler, M. (n.d.). Replace Method with Method Object. Refactoring Guru.
https://refactoring.guru/replace-method-with-method-object

[4] BMC Software. "Code Refactoring Explained." BMC Software, a.n.d. Web.
https://www.bmc.com/blogs/code-refactoring-explained/

[5] Fowler, M. (n.d.). Refactoring.
https://refactoring.com/

[6] Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks (Doctoral dissertation, University of Illinois at Urbana-Champaign).
http://www.laputan.org/pub/papers/opdyke-thesis.pdf