הנדסת תוכנה

מסוכם משיעורי מרן הרב הזמיש במלאכתו

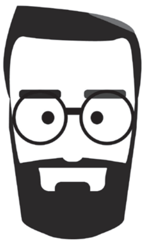
ד"ר דוד דיין שליט"א

עם פירוש

"רי"ח טוב"

מאיתי הצב"י יוחנן חאיק

שנת 01011001111111



"כשם שאי אפשר לבר בלא תבן, כך אי אפשר לתוכנית בלי שגיאות של הבודק האוטומטי"

להערות, הארות ותיקונים:
yohananha@gmail.com
yohanan@ - בטלגרם
ניתן להשתמש בסיכום באופן חופשי לכולם!!

תוכן עניינים

תוכן עניינים 1
קורס הנדסת תבנה 4
סוגי התכנות 5
מה מייצרים 5
מערכת מידע ממוחשבת 5
בעלי עניין 6
השלבים השונים בפרויקט תוכנה 7
שלב 1 לידת הפרויקט 7
שלב 2 -שימוש ותחזוקה 7
שלב 3 -סוף חיי הפרויקט 7
מהם מרכיבי הפרויקט? 7
הידע הדרוש למהנדס תבנה 7
מידול Modeling 8
תהליך בניית התבנה 8
כשלונות פרויקטי תבנה 9
משבר תבנה 9
מדדי הצלחה 11
איכות תבנה 13
ניהול פרוייקט תבנה 14
מחזור חיים של תבנה Software Development Life Cycle (SDLC) 15
מודל מפל המים 15
יתרונות השיטה 16
חסרונות השיטה 16
מתי נשתמש בשיטה ומתי לא 16
ניתוח מערכות מידע 17
שיטת ADISSA 17
תרשים DFD מרכיביו וחוקיו 19
הסבר חוקי המרכיב פונקציה 20
הסבר חוקי המרכיב ישות 21
הסבר חוקי המרכיב מאגר מידע 22
הסבר חוקי המרכיב יישות זמן 22
הסבר חוקי המרכיב יישות זמן אמת 22
הסבר חוקי המרכיב זרם מידע 23
חוקי הקשר בין תרשים האב לתרשים הבן 24

מודלים לפיתוח מערכות תבנה....27
מודל מפל המים 27
מודל "בנה ותקן" 27
הסבר המודל 28
יתרונות המודל....28
חסרונות השיטה 28
שימוש במודל....28
מודל וי V 30
הסבר המודל 30
יתרונות המודל....30
חסרונות המודל 30
מתי נשתמש בשיטה 31
מודל האב-טיפוס 31
הסבר המודל 31
יתרונות המודל....32
חסרונות המודל 32
שימוש במודל....32
ניהול סיכונים....33
גישת ניהול סיכונים: 33
זיהוי הסיכונים....34
הערכת (אומדן) סיכונים....35
השפעות סיכון 36
טבלת סיכונים 36
ניסוח תוכנית פעולה 37
המודל הספירלי 38
הסבר המודל 38
יתרונות המודל....40
חסרונות המודל....40
UP – Unified Process 41
עקרונות UP....41
שלב ההתחלה Inception Phase 42
Use-Case....43
מקרה-בוחן אוסברט אוגלבי 43
שלב הביסוס Elabortion Phase 45
מודל רעיוני Conceptual Model 46
שלב הבניה Construction Phase....52
דרישות וניתוח של שלב הבניה....52

המודל הדינאמי 54
תרשים שיתוף-פעולה Collaboration Diagram 57
תרשים רצף Sequence Diagram 58
שלב ההטמעה Transition Phase 62
פיתוח תכנה בעזרת שיטות Agile 63
המנשר הראשון 63
12 עקרונות האג'ייל 65
הבדלים בין הגישה האג'ילית לגישה המסורתית 66
המנשר השני 67
מתודולוגיות זמישות 68
חסרונות השיטה 69
מתי משתמשים בשיטה 69
בדיקות 70
סוגים שונים של בדיקות תכנה 70
עקרונות ביצוע בדיקות תכנה 72
סוגי בדיקות תכנה עיקריים 72
קידוד בדיקות יחידה 73
דוגמאות לבדיקות יחידה 74

קורס הנדסת תכנה[1]

עד עכשיו כאשר דיברנו על תוכנה, דיברנו על תוכנה עם קוד בשפה כלשהי, והתייחסנו לחלק זה בלבד. קורס זה מתעסק בחלק שמעבר לקוד, אבל הוא חלק אינטגרלי לא פחות ממרכיבי התכנה. האם אנחנו יכולים לחשוב על דברים מעבר לקוד, שמתייחסים לתוכנה? נתקלנו בעבר כבר בתיעוד והערות שנכתבים על גבי הקוד עצמו, אך כל זה הוא רק חלק שולי, ועד כמה שהוא חשוב, לא עליו אנחנו מדברים.

ההגדרה אותה אנחנו יכולים לראות במצגות מדברת על "תוכנות מחשב ותיעוד נלווה כמו דרישות, עיצוב תוכנה ומדריכי משתמש. מוצרי תוכנה יכולים להיות מיועדים עבור לקוח מסוים או עבור משתמש גנרי". נפרט על מה בדיוק מדובר:

Requirements (דרישות) –

לו נקבל דרישה לתכנת מערכת, למשל את מכון-לב. נצטרך להבין את כל הדרישות שהמערכת תידרש לעשות. עומדים לפנינו כל המרכיבים – סטודנטים, כיתות, שיעורים, מרצים וכו', ואנחנו צריכים לעבוד אל מול רשימה של דרישות מסוימות אותם אנחנו צריכים לקיים. למשל: <u>מרצה</u> נדרש להעביר <u>שיעור</u> מסויים ב<u>כיתה</u> מסויימת. תלמיד לא יכול להעביר את השיעור, ואנחנו צריכים שתהיה לנו אפשרות להוסיף ולהחסיר נתונים. לאחר שיש לנו את רשימת הדרישות, ניתן לומר שהגענו ל**מפרט (Specification)** אותו אנחנו מחפשים. שאלה שדנים בה רבות היא האפשרות להוסיף דרישות לאחר שגיבשנו את הרשימה הראשונית – האם זה בכלל אפשרי? נדון בזה בהמשך ונראה שיש גישות שונות בעניין שנעות על הקשת הרחבה בין קיבעון מוחלט של הדרישות, לבין האפשרות לשנות כל הזמן תוך כדי תנועה. כמובן שיש הרבה מרווח בין שתי גישות אלו, ואת כל זה נראה בהמשך.

תוכניות עיצוב (Design Models) –

חלק זה מתחלק לשני דברים – **ניתוח (Analysis)** - הבנה מלאה של כל הדרישות. כאשר לקוח מבקש מאיתנו דרישה מסוימת במפרט, לא בטוח שאנחנו מבינים אותו בפעם הראשונה. מן הסתם, לא כל דרישה שתופיע לנו תהיה פשוטה כמו שנתקלנו בה עד עכשיו – תוסיף, תוריד. עלינו להבין מול מה הלקוח עומד ומה הדרישה המסויימת שהוא ביקש דורשת מאיתנו. יותר מזה, בזמן הניתוח, אנחנו לא מדברים על המחשב והתכנה עצמה! לא מעניין אותנו אילו בסיסי נתונים עומדים לפנינו ובאיזה מעבד נשתמש – אנחנו מדברים על הבנה של הבעיה הניצבת מולנו. לאחר שנבין את כל זה, נוכל לפנות לשלב **עיצוב התוכנה (Software Design)** בניית המודל עליו נתבסס ויצירת כל החלקים המרכיבים את התוכנה הדרושה – כל זאת עשינו אפילו בלי לכתוב שורה אחת של קוד[2]!

יש לזכור – אנחנו עובדים בסופו של דבר מול לקוח. לכן עלינו לדאוג שתהיה דרך למשתמש להתקין את התכנה, ועליו להבין את כל האפשרויות הגלומות בה. לכן, עלינו לתת למשתמש איזשהו מדריך שמסביר לו מה הוא עושה וביצד עליו להגיע לכל דבר.

תוכנה למשתמש מסוים – כל הדרישות מגיעות מצד הלקוח, כל העבודה נעשית מולו ובשבילו.

תוכנה גנרית – כמו למשל וורד, תוכנה אחת שאמורה לספק את כל המשתמשים באשר הם. עלינו להיות קשובים למה שכל הלקוחות אומרים, אך לזכור שעלינו לקלוע למכנה המשותף הרחב יותר.

---

[1] מצגת 1

[2] ה-UML שהתחלנו ללמוד עכשיו, מכיל בתוכו 14 מודלים שונים, חלקם קשורים לניתוח הדרישות, וחלקם קשורים לעיצוב התכנה, אך למעשה, כל זאת נעשה לפני בניית התכנה עצמה.

לאחר כל הכתיבה וההטמעה, נגיע לשלב הטסטים.

סוגי התכנות

האם כל תחומי התכנות דומים אחד לשני? ניהול מחסן, בתי ספר וכו'. האם הם דומים לאפליקצית ווב? בעוד שהאחד דורש מאיתנו הבנה של בסיסי נתונים, בבניית אפליקציית ווב אנחנו צריכים לדבר על קליינט וסרבר וכל הנוגע בעניין, עד שניתן לומר שכמעט כל הכלים שדיברנו על תוכנת הניהול לא מתאימות לתוכנות ווב. מעבר לזה - ברור שאנחנו צריכים שעבור כל תכנה נהיה תלויים גם בחומרה – למשל – משחקים ישתמשו יותר בגרפיקה ואלגוריתמים מאשר תוכניות אחרות. כך שלכל סוג תכנה נצטרך להשתמש באותם כלים אך באופן אחר. המודלים השונים אותם נלמד יהיו מתאימים עבור כל תוכנה באופן שונה.

מה מייצרים

קוד מקור – האלגוריתמיקה מאחורי התוכנית. היינו רגילים לחשוב עד עכשיו שזה העיקר ובזה מיקדנו את מירב המאמצים, אבל אנחנו כבר מבינים שזה חלק חשוב והכרחי,אבל אך ורק חלק מהעבודה.

קוד הרצה – התוכנית לצד הלקוח. זאת המטרה, אנחנו לא עובדים על ריק ולמטרת הנאה. בסופו של דבר אנחנו צריכים להגיע לאיזשהו מוצר, מוצר בו ישתמשו בצורה טובה ונוחה.

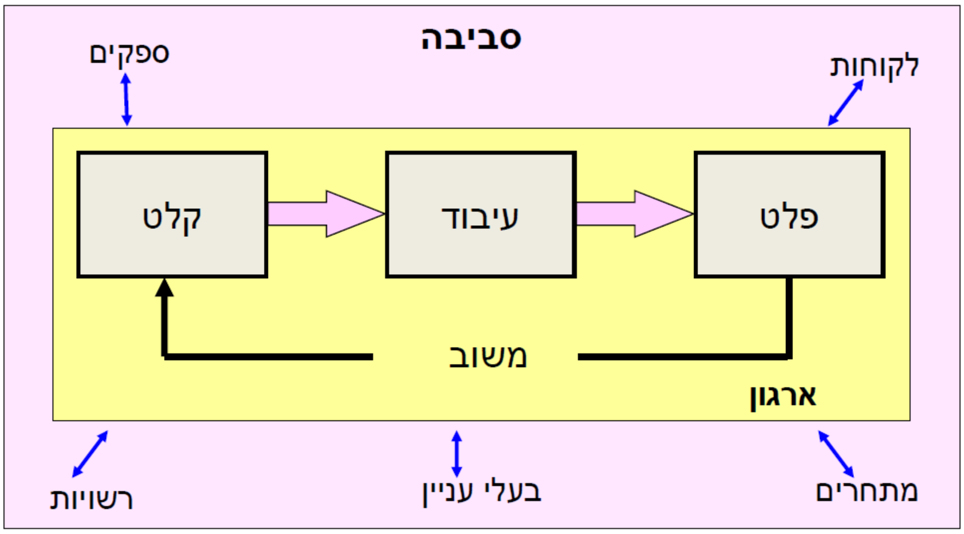
מערכת מידע ממוחשבת

מהי "מערכת מידע ממוחשבת"? למעשה זה מה שאנחנו דורשים להגיע אליו בסוף התהליך. ננתח את הביטוי הזה צעד אחר צעד.

מערכת – קבוצת רכיבים הפועלים יחדיו למטרה משותפת. הגדרה זו מתאימה לכל מערכת – ניהולית, אנושית, צבאית או כל דבר אחר.

מערכת מידע – מערכת האוספת נותנים, מעבדת, מאחסנת ומפיצה את התוצאה. מערכת מידע קיימת בחברות שונות ומתייחסת למה שמעבד את כל הנתונים – לכל ארגון יש סביבה משלו, לא מדובר רק על הכנסת הנתונים עצמם והפלט שלהם, אלא גם הספקים של המידע, הלקוחות שדורשים את המידע, ואנחנו צריכים גם להתעסק עם רשויות, מתחרים ובעלי עניין. המטרה היא להסתכל כמה שאפשר ברמת המאקרו, ולהגדיר את הסביבה בצורה נכונה.

מבחינת רכיבי מערכת המידע, ניתן לחלק את זה לארבעה חלקים עיקריים: 1. <u>אנשים</u> – המערכת האנושית המורכבת החל מהמנכ"ל של החברה עד לפועל הייצור הפשוט ביותר. 2. <u>חומרה</u> – האמצעים הפיזיים עליהם אנחנו עובדים – המכונות והמכשירים השונים. 3. <u>נתונים</u> – בסיס המידע. יכול להיות שמדובר במידע שמגיע מהאנשים או מהחומרה השונה. 4. <u>תוכנה</u> – העיבוד הנעשה לכל המידע הנאסף והפלט היוצא ממנו. יכול להיות גם שהפלט חוזר למכונות ולא באמת "יוצא" אך אין זה משנה.



תרשים זה משקף את מערכת המידע בסביבה הארגונית. כאשר אנחנו מגדירים את ה"סביבה" אנחנו לא מתייחסים רק למידע הנאסף בתוכבי החברה, ובין המכונות השונות, אלא גם המידע המתקבל מהספקים השונים, מבעלי העניין שבוודאי רוצים דברים שונים ממה שאנחנו בדיוק מעוניינים בהם. באופן דומה אנחנו צריכים לאסוף מידע גם על המתחרים שלנו – לראות היכן הם עומדים ומה כבר לא רלוונטי להמשך הפיתוח.

מערכת מידע ממוחשבת – מערכת מידע העושה שימוש בטכנולוגיית מחשבים.

בעלי עניין

מי הם בעלי העניין והאנשים שמשפיעים על המערכת. ישנם שלושה סוגי אנשים דומיננטיים שאנחנו צריכים להתייחס אליהם.

המשתמש –

אליו ממוענת התוכנה. אנחנו לא עובדים באויר, וצריכים לזכור שעלינו להיות נוחים למשתמש. "נוחים" בכל המובנים – גם שיהיה פשוט וקל להשתמש בתוכנה, וגם שימלא את הרצונות השונים. עלינו להתייעץ עם המשתמשים הרלוונטיים שיודעים מה דרוש: אם אנחנו מתכננים כספומט, אנחנו נבדוק עם המשתמש כיצד הוא רואה את "חווית" המשיכה, אם אנחנו רוצים איזה דוח מסוים, אנחנו יכולים לשאול פקיד שמבין היטב מה הלקוח מבקש ומה המידע הדרוש לו. בעבר התעלמו לא מעט מהמשתמשים, אך היום מבינים את החשיבות של עירוב הגורמים האלה.

המפתח – מי שכותב את קוד התוכנה

היזם או הלקוח –

הבעלים של התוכנה, או המשקיע בחברה שיש לו דעה שצריך להתחשב בה. לפעמים מדובר במנהל ראשי שמגיע מהתחום הדרוש, ולפעמים מדובר במשקיע שלא מבין עד הסוף את התחום אך עדיין דורש להיות בעל הדעה.

השלבים השונים בפרויקט תוכנה

שלב 1 לידת הפרויקט

מישהו חושב על רעיון כלשהו במוח – הרעיון יכול לבוא מצורך כלשהו המגיע מחוסר, או רעיון שמגיע מיכולת לשפר דבר קיים או כל דבר אחר. הוא פונה ליזם, וניגשים ביחד למפתחים שכותבים את התכנה העונה על הצורך של היזם או בעל הרעיון, ובסוף מוציאים את התכנה בחזרה ליזם (חשוב לזכור – שלב מסירת התוכנה ליזם עדיין שייך לחלק לידת הפרוייקט. הופיע במבחנים בעבר).

שלב 2 -שימוש ותחזוקה

בשלב שהתוכנה כבר קיימת, מתחילים להתקין אותה אצל היזם, או מוציאים אותה ללקוחות אחרים. בשלב זה בדרך כלל מתגלות כל הבעיות, כל הבאגים צצים ועולים, ואז חוזרים לסבב תיקונים נוסף וחוזר חלילה, עד שמגיעים לשלב שהוא אופטימלי – התוכנה עובדת ואין כמעט באגים קיימים, או אלו שיש כבר אינם קריטיים כמו קודם, ושלב הפיתוח עובר לשלב של אחזקה.

שלב 3 -סוף חיי הפרויקט

לאחר זמן מה, כאשר התוכנה כבר אינה רלוונטית ותחזוקה שלה עולה יותר מכפי שהיא שווה, מסיימים את חיי התוכנה ועוברים הלאה לאתגר הבא.

מהם מרכיבי הפרויקט?

מה הדברים שצריכים להיות בכל פרוייקט-

מאפיינים – דרישות – לדאוג בראש ובראשונה לכל הדרישות מא' עד ת'. דבר שכבר הוזכר מקודם – להבין את הדרישות ואת המאפיינים השונים של התוכנה עד לרמה הקטנה ביותר שלהם.

זמן – תמיד אנחנו מוגבלים בזמן. עד כמה שנרצה לעבוד על כל פיצ'ר הכי קטן כמה שיותר זמן ולשכלל אותו לרמת שלמות, כל פרויקט צריך לעמוד בדד-ליין מסוים. יותר מזה, בדרך כלל ישנם אבני דרך – נקודות ציון בזמן אליהם אנו שואפים להגיע לאחר שכבר השגנו התקדמות משמעותית כזו או אחרת בפרויקט.

משאבים – כל מה שאנחנו צריכים בשביל לפתח את הפרוייקט – המפתחים עצמם, תהליכי הפיתוח והשפות המסוימות, טכנולוגיה בה נשתמש ועוד. מעבר לזה, אנחנו צריכים לחשוב על כל המאפיינים של הידע אותו אנחנו נדרשים להכיר – לא רק **מה צריך לדעת**, אלא גם **מה לא צריך לדעת**. אם אנחנו עובדים על ויז'ואל סטודיו או על שפת תכנות מסוימת, אין לנו צורך לדעת את כל המרכיבים עד לפקודות הכי קטנות ונדירות, אלא אנו יכולים להסתפק במה שנכלל תחת תחום הידע הרלוונטי שלנו ומשם להמשיך ולהתפתח. המשאבים גם מכילים את כל החומרה הרלוונטית לעניין - האם לקחת מחשבים חזקים או כאלה שיספיקו למערכת, כמה זיכרון לדרוש מהמרכיבים השנים ועוד.

כל אלו מובילים אותנו ל**איכות המוצר** -כל מה שיוחלט צריך להוביל למוצר הכי איכותי – אך כמובן שאנחנו צריכים להגדיר מה נכנס תחת הגדרת ה"איכות". <u>הטסטים</u> הם בוודאי חלק מאיכות התכנה, ככל שנבדוק את התוכנה יותר, כך הטעויות יקטנו ויהיו פחות משמעותיות. <u>נוחות משתמש</u> – מהירות, זיכרון, אנחנו צריכים להחליט מה מתוך כל זה מוגדר איכות – הבנה של מה הלקוח באמת רצה בדרישות, ווידוא שלא עשינו דברים שאנחנו רק חושבים שהלקוח רצה, אך באמת הוא התכוון לדבר אחר.

הידע הדרוש למהנדס תכנה

כשאנחנו מסתכלים על עולם המדע, אנחנו יכולים לחלק לשניים – Software ו Problem Domain

Domain – אם נותנים לנו בעיה לפיתרון אנחנו צריכים לכתוב איזה תכנה מתאימה. אבל הבעיה היא

שברוב המקרים, הבעיה הניתנת למהנדס התכנה היא בעיה שאינה קשורה באופן ישיר למהנדס עצמו. אם נרצה להנדס מכונית, הבעיה שייכת לתחום המכניקה, אך איש מדעי המחשב, לא יודע כיצד לגשת לבעיה. אם רוצים להנדס שהחגורה תצפצף כל עד אינה מוכנסת, זה כבר תחום אלקטרוניקה וחשמל. כל דבר שייך לאיזה איזור בעיה אחר שהוא מדעי, ואיש מדעי המחשב צריך להפוך את המדע לתוכנה כלשהי, עליו ללמוד את השדה הרלוונטי, לפחות ברמה שהוא יוכל לעבוד וליצור תכנה. וזה קשה מאוד. אך זה האתגר הגדול להפוך כל בעיה כלשהי לתכנת מחשב על פי הדרישות

עלינו, כמתכנתים, **ללמוד מהר** את איזור הבעיה – ממה שאנו למדים **להפיק מודל אבסטרקטי** ועל גביו **לבנות את העיצוב** שמממש את המודל האבסטרקטי.

מידול Modeling

לאחר שבנינו והבנו את התכנית הכללית ואנחנו יודעים מה מצופה מאיתנו לעשות, אנחנו עוברים למידול. בנייה של המערכת בצורה אבסטרקטית. הרעיון של המודל הוא לפשט את הרעיונות אותם אנחנו רוצים לבטא / להכניס בתכנה בצורה אבסטרקטית.

חשוב לשים לב – המידול אינו מדבר על בניה של כל המערכת בכללותה, אלא על האלמנטים השונים ממנה מורכבת המערכת – עושים זום אין לכל החלקים שהגדרנו שצריכים להיות ומתמקדים עליהם. מה הכוונה?

דיברנו קודם על כך שכחלק ראשוני מבניית התכנית אנחנו יושבים על כל הדרישות (Requirements) של התכנה, ואת זה אנחנו עושים בצמוד למשתמש הדורש את המערכת. אם עשינו את השלב הזה נכון אנחנו יכולים כבר לדעת מהם האלמנטים השונים הדרושים לנו ומהם הפונקציות השונות שעלינו להוסיף. המודל כעת מתייחס לחלקים אלו המרכיבים את הדרישות המאוד קטנות האלו. ברגע שנבנה (באופן מופשט) את המודלים הללו, נוכל להרכיב הכל לכדי התכנית הגדולה. חשוב להזכיר שוב – כאשר נבנה את המודלים האלו אנחנו לא נתרכז בפרטים היותר-קטנים המרכיבים אותם, אלא נבנה תכנית כללית האומרת מה צריך לעשות, ורק בשלב בניית הקוד עצמו נתחיל לפרט יותר.

כמובן, שאם נבנה את המודלים האלו בצורה נכונה (כמו שעשינו עם UML, למשל) המעבר מכאן לבניית הקוד תהיה מאוד פשוטה – נתרגם את מה שאנחנו רואים לשורות קוד באופן ישיר.

לצורך ההבנה, בעצם ניתן להסתכל על זה כמו על ציור שבנוי מהמון פרטים. אם נתקרב מאוד ונסתכל על הפרטים השונים, נוכל לראות ממש את משיחות המכחול, וכל חלק הכי קטן עם הפרדת הצבעים, אך בוודאי שזה לא יהיה "הציור". התמונה השלמה תתקבל מהרכבת כל החלקים.

תהליך בניית התכנה

את כל התהליך הזה הזכרנו כבר קודם, אך כעת שהבנו את כל החלקים, יש צורך להסתכל על זה שוב ונבין הכל בצורה יותר פשוטה. כאשר אנחנו נסתכל על המודלים השונים של פיתוח תכנה (לא המודלים הקודמים של ה modeling, אלא מדולים בדוגמת agile ודומיהם), אנחנו ניקח כל מודל ומתוכו נגזור את סט הפעולות הדרושות לנו על מנת להתקדם בבניית התוכנה על פי המודל אותו נציע.

הפעולות הכלליות של כל מודל יורכבו מארבע התהליכים לקמן:

- דרישות (Specification) – דבר שעליו כבר דיברנו לא מעט. הדרישות השונות מהתכנה, מה היא אמורה לעשות ומהם ההגבלות הניתנות עליה (לא כל דבר אנחנו חייבים לעשות ויהי מה, אלא הדרישות הם נגיד יכולים להיות מבחינת זמן ביצוע, הכבדה על המערכת וכו')
- פיתוח (Development) – יצירת התכנה. לעומת שלב הדרישות שהוא די זהה בין המודלים השונים, בשלב הפיתוח עצמו מתגלים ההבדלים והפערים בין המודלים השונים.

- תיקוף (Validation) – הולידציה של התכנה עובדת בשתי רמות. האחת, לוודא שמילאנו את כל הדרישות אותם הלקוח ביקש. לוודא שבמהלך העשייה, לא איבדנו את הכיוון שנדרשנו אליו. הדבר השני, והחשוב לא פחות, הוא לוודא שהתכנית באמת עובדת ואין בה באגים (או לפחות לא באגים שאנחנו רואים במבט חטוף).
- התפתחות (Evolution) – על שלב זה פירטנו קודם, שלבי התפתחות התכנה. לכל תכנה יש "מעגל חיים" החל מהרגע הראשון שמשחררים את התכנה לאוויר העולם, וכל התיקונים הנדרשים תוך כדי העבודה האמיתית שהלקוח מבצע איתה. לאחר שמתגברים על הבאגים הראשונים שבדרך כלל מופיעים, מתחילים להוסיף פיצ'רים ודברים נוספים שבעת הלקוח מבין שהוא צריך / רוצה, ואז התכנה רצה ועובדת עד השלב בו מחליטים להפסק את התמיכה בתכנית, ולעבד על הפרוייקט הבא.

כשלונות פרויקטי תכנה

במקרים רבים, בנייה לא נכונה ועבודה שגויה על פרוייקט תוכנה גורמים בסופו של דבר לכישלון הפרוייקט. רבים ניסו להבי מהם השלבים הנכונים לביצוע הפרוייקט, וההשוואות הנפוצות ביותר הם לבניית גשר.

כאשר בונים גשר מקבלים תקציב מסוים בו צריך לעמוד – הן מבחינת תקציב כספי, והן תקציב זמן, מסגרת זמן בה הפרוייקט צריך להיות מוכן. בדרך כלל פרויקט הנדסי עומד בתקציב הזמן, הכסף, וחשוב לא פחות – עומד כמו שצריך ולא נופל. לעומת זאת, כאשר עובדים על בניית תכנה המצב שונה לגמרי, הרבה פעמים לא עומדים בזמן היעד, חורגים מהתקציב הכספי, ובזמן שחרור התכנה רואים שיש בה לא מעט באגים. אם זה בכלל עובד.

מדוע ההבדל כה גדול?

בניית גשר נעשית בדרך כלל תחת תכנית מסודרת לרמת פרטי הפרטים. הקבלן לא עומד על צד אחד של הגשר, מצביע לכיוון השני ואומר לפועלים "בואו נגיע לשם", אלא עובדים תכנון מדויק של מהנדסים ואדריכלים הלוקחים בחשבון את כל הפרטים הנצרכים – גודל, חומרים וכו' על מנת לוודא שהגשר יעמוד. כמובן, שלא ניתן באמצע העבודה להחליט שמאלתרים ומורידים דברים (וגם לא מוסיפים דברים מיותרים), התכנון עצמו הוא נוקשה ולא גמיש בכלל. דבר נוסף – במידה והגשר נופל, האירוע נחקר ומוצאים מה היתה הסיבה לנפילת הגשר, הלקחים נלמדים ומי שמסתבר שזו אשמתו, משלם את המחיר על הכישלון.

בבניית תכנה המצב קצת שונה – יותר מידי פעמים עובדים בצורה לא מסודרת ולא ברורה לגמרי. כך שבמהלך העבודה חלקים מהתכנה הולכים לאיבוד וחלקים לא נצרכים מתווספים לתכנה. בנוסף, המתכנת מתעלם מכל מיני באגים שצצים לו בעבודה והוא מתעלם מהם או עושה איזה מעקף זמני שלא פותר את הבאג, אלא רק את התקלה הספציפית. כמובן שכתוצאה מכך, טעויות כל הזמן חוזרות וקשה לומר מי היה האשם ומי אחראי על התקלה.

משבר תכנה

"בניית תכנה היא כמו בניית קתדרלה, מסיימים לבנות ומתחילים להתפלל" (סמואל רדווין, 1988)

הרבה פרוייקטים של תכנה, שנראים במבט ראשון מאוד מוצלחים, מסיימים בסוף בכישלון. דבר זה נובע ממספר סיבות שעליהם דיברנו, אך ניתן לומר באופן די גורף שמרבית הבעיות נגרמות כתוצאה מכשל אנושי. כשלון של פרוייקט מוגדר כ"**משבר תכנה**" (Software Crisis). ברגע שמגיעים לשלב של משבר

תכנה, צריך לעצור הכל ולהתחיל לבחון מחדש היכן אנחנו עומדים, ומה אנחנו יכולים לעשות על מנת לתקן את הבעיה.

מחקר שנערך בשנת 1979 בארה"ב הגיע לממצאים הנוראיים הבאים:

- 2% מהתכנות המוגמרות בלבד עובדות בזמן מסירת התכנה ללקוח.
- 3% מהתכנות עובדות אחרי מספר תיקונים.
- 45% מהפרוייקטים נמסרים ללקוח, אבל מעולם לא נעשה בהם שימוש ראוי.
- 20% עוברות שימוש, אבל רק לאחר שינויים משמעותיים.
- 30% מהפרוייקטים משולמים, אך מעולם לא מגיעות לשלב הסופי או מעולם לא נמסרות ללקוח.

כמובן שהממצאים האלה לא ממש תורמים למוניטין של המתכנתים. בעקבות המחקר ועם התפתחות יכולות התכנות, התפתחה גם המתודיקה של "הנדסת תכנה". הנדסת התכנה, כמו כל הנדסה ניסתה לבנות שיטות מוסכמות ונהלי עבודה שיובילו את המתכנת ואת כל הצוות עצמו לעבודה מסודרת שתניב תוצאות ראויות.

העקרונות העיקריים עליהם עבדו הם ארבעה:

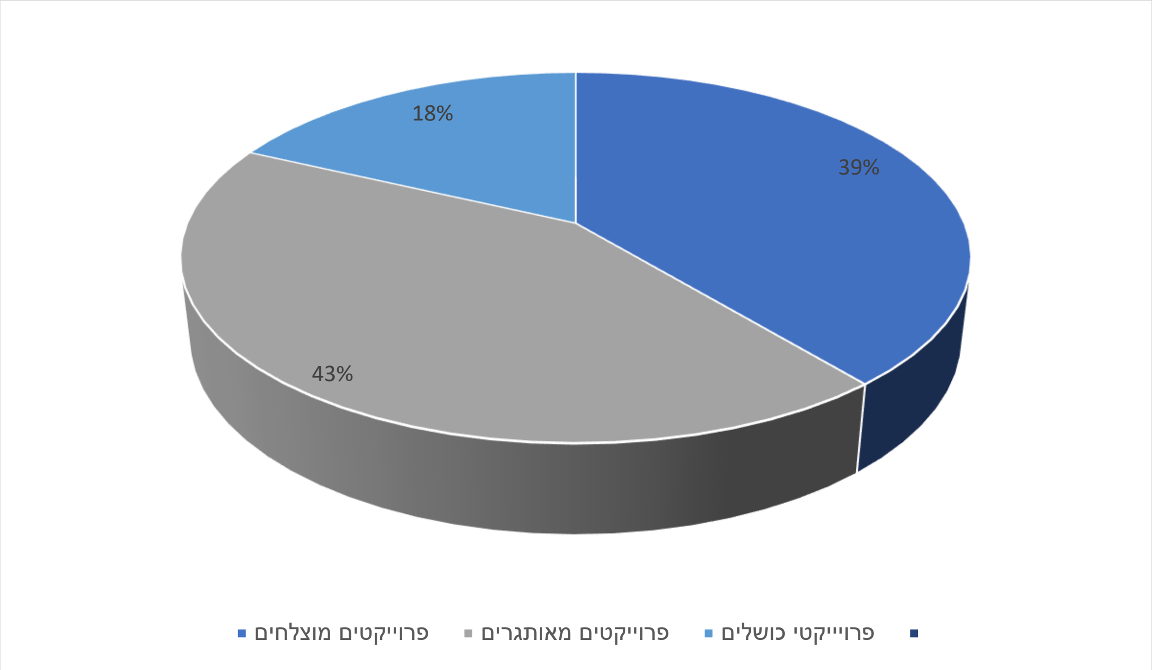
- גישה שיטתית לפיתוח תכנה.
- יישום עקרונות הנדסיים (סדר עבודה).
- אוסף נהלים, כלים ושיטות עבודה.
- הבטחה של איכות תכנה בצורה עקבית.

בשנת 1985 הוקמה קבוצת סטנדיש (Standish Group) שלמדה את כל הלקחים האלה, ולאחר יותר מידי כשלונות של פרוייקטים שכמובן הובילו לחשש של משקיעים כלפי פרוייקטי תכנה, התחילו לקטלג ולעבוד על עקרונות התכנה ועל סיווג הפרוייקטים ורמת ההצלחה שלהםץ

את כל הפרויקטים אותם הם למדו הם סיווגו לשלוש קבוצות עיקריות:

1. פרויקט מוצלח – פרוייקט שעומד בתקציב הזמן והכסף, מגיע עם כל הפיצ'רים הנדרשים על פי דרישת הלקוח, ונעשה בהם שימוש למטרה המתאימה.
2. פרוייקט מאותגר – הפרוייקט עובד ונעשה בו שימוש, אך חרג ממסגרת התקציב וההערכות, ולא מציג את כל הפיצ'רים הנדרשים.
3. פרוייקט כושל – הפרוייקט מבוטל לגמרי בנקודה מסויימת במהלך הפיתוח.

במהלך השנים הם הוציאו דוח שנתי - "דוח הכאוס" המציין את האחוזים השונים של כל הפרוייקטי שנסקרו. למשל עבור שנת 2012 הופיע הדוח הבא:



ניתן לראות, שבערך 60% מכלל הפרויקטים לא מוגדרים כפרוייקטים מוצלחים. אמנם יש לסייג, שפרוייקט מאתגר הוא פרוייקט שיכול לאחר עבודה ומאמץ להגיע להצלחה מסויימת, אך בוודאי זאת לא השאיפה שלנו.

מדדי הצלחה

האם ניתן לקחת גורמים שונים, שבזכותם אנחנו יכולים להבטיח שהפרוייקט יצלח? בשנת 2015 קבוצת סטנדיש פרסמה את מדדי ההצלחה – הפקטורים השונים בבניית תכנה, ששקלול של כל אחד מהנתונים עם הניקוד המתאים לו יכול להוביל את הפרוייקט להצלחה. כמובן שתמיד יכולים לקרות דברים שהם מעבר לשליטתנו, אך הקפדה על הפקטורים האלה תוכל למנוע או לצמצם את השגיאות האנושיות. מעניין לראות שתכנות עצמו תופס חלק קטן מכל השלול הסופי.

4 הדברים החשובים ביותר (15 נקודות כל אחד) הם:

תמיכת מנהלים – עמידה של המנהל בצורה בטוחה מאחורי החברה.

בגרות רגשית - התלהבות ומוטיבציה- אחד מעקרונות האג'ייל שנלמד בהמשך, הוא עבודה בגובה העיניים.

פנייה למעורבות המשתמש – בשיטות הקלאסיות מתייחסים למשתמש בזמן הדרישות, ומוודאים שאנחנו מבינים מה הוא רוצה, אך אז מתנתקים ועובדים לבד ללא פניה אליו, ולאחר מכן מביאים לו את התכנה הסופית, ללא כל מעורבות מצידו בתהליך הביניים. אך המחקר הראה שמעורבות ביניים דווקא תורמת לתיקונים (במידת הצורך) עם נזק קטן יותר.

אופטימיזציה –הרבה פעמים שאנחנו מפתחים יותר ממה שאנחנו צריכים – מרוב התלהבות מוסיפים פיצ'רים ואפשרויות. אך באופן טבעי, כל הוספה היא הוספה של באגים ובעיות פוטנציאליות. עלינו להביא בדיוק מה שהלקוח רוצה ולא יותר.

צוות מיומן (10 נקודות) – להבין את העסק ואת הטכנולוגיה. הדרישה מאנשי הצות להיות מקצועיים בכל ההיבטים הקשורים לפרוייקט עליו הם עובדים. לא לעשות רק פונקציות לפי הדרישה, אלא להבין מה המשמעות מאחורי כל דבר.

ארכיטקטורה סטנדרטית (8 נקודות) – שמירה על הארכיטקטורה הסטנדרטית והמוכרת תהיה יותר קלה על מנת להבין בהמשך את כל הפרטים. אם יבוא מתכנת אחר, הוא לא צריך ללמוד מחדש איזה מבנה שהומצא לצורך הפרוייקט. בניה מוגזמת ולא סטנדרטית תוביל בסוף לתכנה שבירה שעלולה לפגום בכל הפרוייקט.

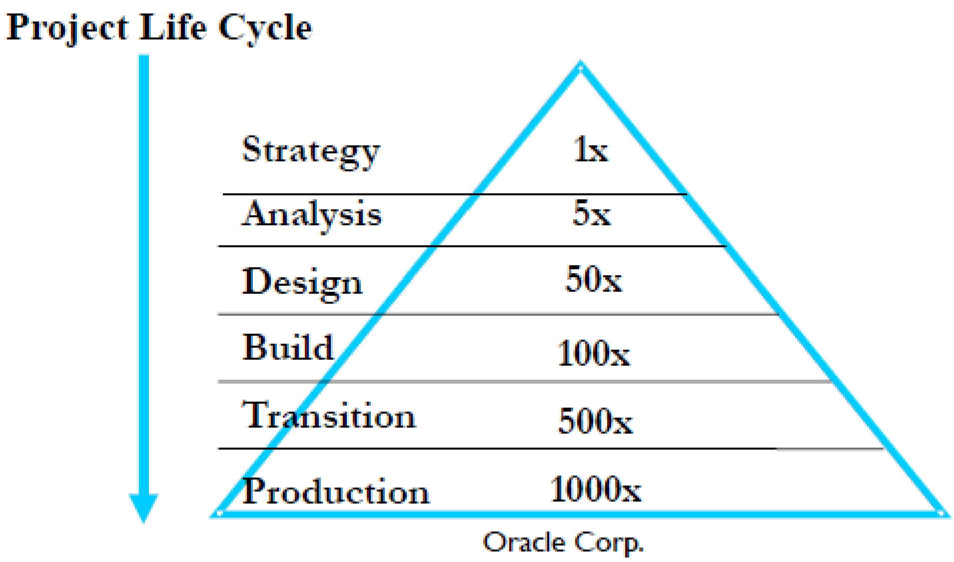
מיומנות אג'יילית (7 נקודות) – שליטה במתודולוגיה. אמנם יש לא מעט מודלים של תכנות, אך הAgile מקבל מקום של כבוד בפקטור של הצלחת התכנית.

צניעות ביצוע (6 נקודות) – עלינו ליצור תוכנה זורמת שלא תתקע את המחשב והresources, אלא תעבוד בצורה חלקה. הרכבת כל חלקי התוכנית בצורה נכונה וזורמת - אנחנו לקוחים את הרכיבים הקטנים ביותר שאנו יכול ליצור, כך שנשמור על באגים במסגרת מסוימת ומצומצמת. אם יהיה באג נוכל לאתר את הפונקציה הקטנה בה זה קרה ולא נחפש בין 2000 שורות קוד.

מומחיות בניהול פרוייקטים (5 נקודות)

הבנת הדרישות (4 נקודות) – ללא הבנת הדרישות נעשה את כל השגיאות שצוינו לעיל.

מה החשיבות לכל הפרטים האלו? מעקב נכון ושמירה על כל הפקטורים השונים, מגבירים את הסיכוי שלנו לתפוס באגים ולתקן טעויות קריטיות מוקדם מאוד, ובהתאם עלות התיקון תהיה נמוכה יחסית. על פי המחקרים, עלות תיקון תוכנה עולה בצורה פרופורציונלית בכל שמתקדמים בשלבי הפיתוח. ניתן לראות זאת בדיאגרמה הבאה:



אם התכנה כבר בשלב ההפצה, אנחנו עלולים לשלם פי אלף על התיקון, כי עלינו לעבוד על כל באג ולדאוג להפיץ את התיקון לכל מי שכבר קנה את התכנה, וההתעסקות עם כל התיקון הזה הרבה יותר משמעותי מאשר אם בשלב האסטרטגיה (המחקר) היו מגלים שחסר פיצ'ר או שאיזה משהו שהוספנו אסור לו שיהיה בצמוד לדברים אחרים.

איכות תכנה

אנחנו שואפים ליצור תוכנה שתהיה כמה שיותר איכותית. ההגדרה הפשוטה של "איכות" היא תכנה שעומד בציפיות הלקוח בצורה נאותה. מעבר לזה, על התכנה להמשיך ולהיות שימושית גם לאחר זמן, ובשביל זה עלינו לעמוד בתנאי איכות פנימיים וחיצוניים:

<u>חיצונית:</u>

- **נכונות** – עד כמה עמדנו בדרישות הלקוח, ועד כמה אי-הבנות ופער יש בדרישות בין המתכנת למשתמש.
- **שמישות** – הקלות בה המשתמש עובד עם התוכנה. אם המשתמש יכול לעבוד על התכנה ללא כל הסבר וניתן לעבוד איתה ישירות - אזי התכנה שמישה. אך אם היא מסובכת לשימוש ללא הסבר מפורט, אזי יש בעיה בשמישות.
- **יעילות** – תכנה שתשתמש בפחות זיכרון ובגישה מהירה יותר בזמן הביצוע, נחשבת יעילה יותר מבחינת הביצוע עצמו.
- **אמינות** – היכולת של המערכת לעבוד ללא כשלים – מערכת אינה תפסיק לעבוד גם אם יש טעות כלשהי, וגם אם עפים מהתכנית אז ברור לנו איפה הבעיה.
- **שלמות** – אנחנו לא רוצים לתת אפשרות למישהו לא מורשה להגיע ולשנות את התכנה.
- **יציבות** – עמידה בפני קלט לא תקין.

<u>פנימית:</u>

- **תחזוקה** – בOOP יש לנו את שלושת החלקים של התחזוקה: <u>יציבות</u> – גם בקלט לא נכון, וגם תיקון והוספת חלקים ללא קריסת מערכת. <u>מודולציות</u> – אפשרויות המחלקה שניתן לבנות ולהוסיף דברים, <u>קריאות</u> – תיעוד ונתינת שמות משמעותיים בהקשר לקונטקסט. (כפל מספרים יכול להיות שטח ויכול להיות חישוב לפונקציה מתמטית פשוטה, וצריך לדעת מה המשמעות של הכפל שאני עושה)
- **גמישות** – הרחבת פיצ'רים תוך כדי פיתוח – אם עשינו פיצ'ר מסוים, ואנחנו רוצים לעשות את אותו פיצ'ר אך שונה במקצת – למשל הדפסה בתמונה או סוג קובץ אחר – וניתן להרחיב את הפיצ'ר ללא בעיות מיוחדות.
- **ניידות** – לקחת את אותה תכנה ולנייד אותה למערכות הפעלה שונות לחלוטין. פיתוח לווינדוס ומאק במקביל.
- **שימוש חוזר** – ברגע שמפתחים את ההירכיה ואת היחסים בין כל החלקים, ניתן להשתמש בכל החלקים גם במערכות ובתוכניות אחרות.

<u>אתגרי הנדסת תכנה</u>

הנדסת תכנה שואפת לעיצוב אופטימלי, ואנחנו מתבססים לזה על שפיטה או ניסיון. מאחר ואין לנו באמת מדד מתמטי שיכול להגיד מה נכון ומה עובד. וכל הגורמים הינם בלתי מוחשיים.

ניהול פרוייקט תכנה[3]

כמו שאמרנו כבר בעבר, ועוד נחזור לומר לא מעט פעמים, את התכנה אנחנו לא כותבים לעצמנו בחושך. (בדרך כלל) אנחנו חלק מצוות גדול המכיל מספר מחלקות, וכל חלק מהצוות אחראי על חלק מסוים בפרוייקט, ועלינו מוטל להבין מראש כל שלב של העבודה, ברמת כמה זמן ייקח לנו כל דבר, כמה משאבים אנחנו נצרוך (בכלכלית), על מנת לקבל תמונת מצב אמתית ונוגעת לשטח עד כמה שאפשר.

ננסה להגדיר בעת מהי "הנדסת תכנה". דבר זה ייתן לנו את האפשרות להבין מה המצופה מאתנו.

אנחנו נתייחס לשלוש הגדרות שונות של הנדסת תכנה. מדוע לא אחת בלבד? מאחר והנושא של הנדסת תוכנה אינו מוסכם וברור היכן הוא מתחיל והיכן הוא נגמר. נראה את ההבדלים בדקויות בין ההגדרות השונות, ומזה נבין את ההסתבלויות.

1. **שיטה מסודרת לפיתוח מוצר תכנה (תעשייתי) איכותי ובעל ערך משלב הרעיון ועד לשלב הפרישה.**

הגדרה זו מתמקדת בצד התעשייתי של הנדסת התכנה. אנו באים לבנות תכנה תעשייתית, המיועדת לשימוש על ידי מאות עובדים (גם אם אין זה כך בסופו של דבר במציאות). איכותית ובעלת ערך (שיווקי). הנדסת תכנה זה מתכון עם צעדים מסודרים על מנת להגיע להצלחה. משלב הרעיון ועד לשלב הפרישה.

2. **בינון ושימוש בעקרונות הנדסיים מבוססים, כדי לקבל באופן כלכלי תוכנה אמינה, הפועלת ביעילות על מכונות אמיתיות.**

ההגדרה הבאה מופיעה בספר של פרסמן. הוא מתמקד במילה "הנדסה". המשמעות של ההנדסה בכל שאר המקצועות ההנדסיים, מדברת על השיטה הסדורה של העבודה. גם אנו כמתכנתים, לא עובדים על פי מודלים אמפיריים, אלא שואפים להתבסס על עקרונות מבוססים. בנוסף, המטרה הסופית היא להגיע לעולם האמיתי – ולכן, התוכנה שנהנדס צריכה להיות אמינה, ופועלת ביעילות על מכונות אמיתיות. לא כתרגיל או שיעורי בית, אלא שימוש בדברים מוכחים, על מנת להוציא משהו בפועל. בדומה למהנדס שבונה גשר, והוא משתמש בכל חוקי הפיזיקה והחומרים, בידיעה שאם משהו ייפול, גם האחריות עליו. כך גם אנחנו צריכים לעשות הכל בצורה מדויקת וברורה.

3.1 יישום גישה שיטתית מיוסדת על כללים ברת כימות, עבור פיתוח, תפעול ותחזוקה של תבנה
3.2 מחקר של גישות, על פי(1).

הגדרה זו של IEEE, לכאורה מדברת בצורה כללית יותר, אך מתייחסת גם היא לעובדה שאנחנו מחפשים תוצאות ברי כימות. הלכנו מנקודה A לנקודה B. כל שלב מיוסד על כללים משלב הפיתוח ועד לתחזוקה השוטפת של התכנה.

במובן שאנחנו שואפים בסופו של דבר, למעין שילוב של שלושת ההגדרות האלה. אנחנו דורשים מעצמנו להגיע עם רעיון איכותי, וביצוע איכותי ומדויק על כל השלבים השונים

[3] מצגת 2

מחזור חיים של תכנה Software Development Life Cycle (SDLC)

נגדיר בעת את ה"פרוייקט":

"פרויקט הוא מאמץ ארגוני זמני, שמטרתו היא הפקת תוצר ייחודי במגבלות הזמן והמשאבים המוקצים"

על פי הגדרה זו, ניתן לומר כי ה"פרוייקט" הוא לאו דווקא בניית כל התכנה מההתחלה ועד הסוף, אלא כל חלק שמוגדר תחת המשימות השונות. כאמור, אנחנו מנסים להפיק את התוצר שלנו תחת מגבלות. מגבלות של זמן, ומגבלות של משאבים. אנחנו לא יכולים לעבוד לנצח ולנסות לשכלל כל נקודה הכי קטנה עד אינסוף, אלא תחת המגבלות הנתונות לנו, עלינו להוציא את הטוב ביותר.

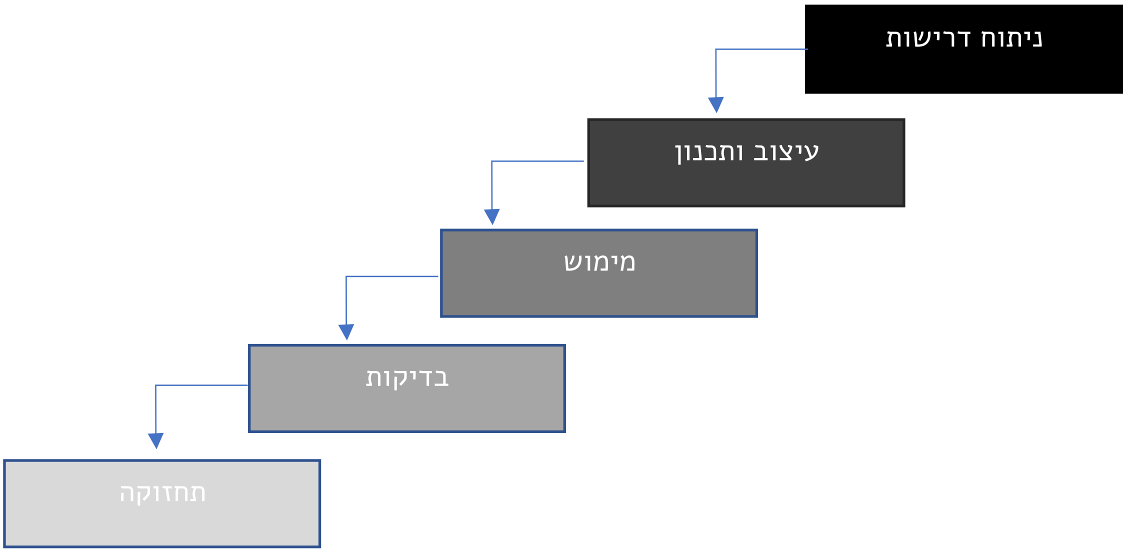
"מודל מחזור החיים" זה השיטות והצעדים שאנחנו עושים בעת בניית תכנה, ותיאור רעיוני של מה שאמור להיות בסוף.

"מחזור החיים" זה הצעדים הממשיים שעלינו לבצע על מוצר מסוים.

מודל מפל המים

מודל מפל המים, החל להתבסס בשנת 1970 לאחר מאמר שכתב וינסטון רויס[4], והתבסס כמודל העיקרי עליו התבססה בניית התכנה בעשורים לאחר מכן.

המודל נקרא כך, מאחר והמתודה העיקרית שלו היא עבודה מסודרת וקפדנית בשלבים בכיוון האחד בלבד. באופן דומה למפל שנופל מטה, ומידי פעם עוצר למלא בריכות קטנות, גם המודל הזה עובד על זרימה בכיוון מסוים, עצירה לביצוע שלב מסוים והמשך הלאה. צורת המודל הכללית נראית כך:



1. **שלב הדרישות** – לחקור את הרעיון אותו אנחנו הולכים לבצע, קבלה של דרישות הלקוח.
2. **חקירה ועיצוב** – הבנה יסודית של דרישות הלקוח. ציור מסמך המפרט. מה התכנה אמורה לעשות. לעומת השלב הראשון שהוא יחסית אמפירי ומתייחס רק למה שאנחנו סופגים מהסביבה,

[4] על פי ויקיפדיה, המודל נבנה על פרשנות מוטעית של המאמר שלו. https://he.wikipedia.org/wiki/מודל_מפל_המים

כאן עלינו ממש להבין את הדרישות השונות ברמה של לנתח כל בקשה לאן היא מגיעה ומה המימושים הקטנים ביותר שלה. בנוסף, עלינו "לעצב" את התכנה – במובן התכנותי של דרך הפעולה של התכנה וביצד היא תבצע כל דבר.

3. **מימוש** – כתיבת התכנה עצמה, תוך כדי הבדיקות במהלך הכתיבה. הטמעת התוכנה אצל הלקוח.
4. **בדיקות** – כמובן שברגע שמתקינים את התוכנה אצל הלקוחות השונים, דברים מתחילים להתפרק, ועלינו לעבור שלב הבדיקות.
5. **תחזוקה** – המשך עדכונים לתכנה על פי דרישות מערכת. אם השלבים הראשונים בוצעו היטב, אין תיקונים מרחיקי לכת, אלא רק התמודדות עם הבטגים והתקלות השוטפות, ושינויים על פי דרישות מערכת חדשות.

המטרה העיקרית של השיטה היא ללכת כמובן, רק בכיוון אחד. אין לנו מקצה שיפורים מעבר למה שאנחנו עושים בשלב המימוש. מסיבה זו, עלינו להשקיע את מירב המאמצים בשלבים הראשונים על מנת למנוע מצב ששכחנו איזו דרישה, או לא הבנו משהו עד הסוף, ובעת עלינו לחזור אחורה ולהתחיל מחדש. כמו שכבר אמרנו קודם, כל שינוי של שלב מוקדם מתייקר ככל שאנחנו מתקדמים הלאה בבניית התכנה.

יתרונות השיטה

מודל פשוט, וקל להבנה ולשימוש.

קשיחות המודל תורמת לניהול יותר קל – מאחר וכל שלב בו מוגדר בצורה מאוד ברורה היכן מתחיל ונגמר כל דבר, קל יותר לדעת שאנחנו מבצעים דברים בצורה נכונה.

השלבים אינם חופפים – ולכן אין דילוגים.

חסרונות השיטה

זמן הפיתוח ארוך יותר - התמקדות בחקירה במקום בעבודה עצמה.

נתק בין המפתחים והמשתמשים – מלבד השלב הראשון בו עובדים מול הלקוח, אנחנו לא רואים אותו עד לשלב ההטמעה. כך שאם טעינו באחד השלבים הראשונים, התיקון הרה יותר מסובך.

אם באמצע הדרך מגלים שיש צורך לעשות שינויים, עלינו להתחיל את כל התהליך מהשלב הראשון שלו.

אין תוכנה עובדת (אפילו לא בצורה מינימלית) עד לשלבים האחרונים של הפרוייקט.

כמויות גדול של סיכונים ואי וודאויות.

מתי נשתמש בשיטה ומתי לא

מכל האמור לעיל, ניתן לסכם ולראות שהשיטה הזאת אינה מתאימה לפרוייקטים גדולים ורחבים, בעלי סיכון גבוה. אם נעבוד בשיטה זאת בפרוייקטים כאלו, ישנה סבירות גבוהה שהמימוש לא ייצלח.

לעומת זאת, אם יש לנו פרוייקט קטן, שהדרישות שלו מאוד ברורות, אז קל יותר לממש את השיטה הזאת, בידיעה שגם אם יהיה עלינו לחזור ולתקן, הנזק לא יהיה גבוה.

החל מחלק זה, לא היה בחומר של שנת תשע"ט

ניתוח מערכות מידע[5]

נניח שאספנו כבר את כל הדרישות ועכשיו אנחנו רוצים להתחיל ולנתח את המערכת. הניתוח אומר לקחת את הפרוייקט שלנו, ולנתח אותו מעבר למושג המחשבים. האנליסטים שעשו את כל הניתוחים עד היום לא היו אנשי מחשבים, אלא אנשים שבל תפקידם היה לנתח את המערכת מהבחינה השימושית. וגם בעת אנחנו, כמנתחים את המערכת צריכים להגדיר את המערכת בצורה פשוטה ולא במושגי תכנות.

אנחנו צריכים עכשיו לראות בדיוק את כל הפונקציונליות הנדרשת מהמערכת. כמובן שמבחינת הניתוח אנחנו מדברים רק על מה שנכנס ומה שיוצא מהפונקציה (ולא על החישוב), אנחנו מכניסים קלט מסויים, ומצפים לפלט שיגיע לידיים הנכונות.

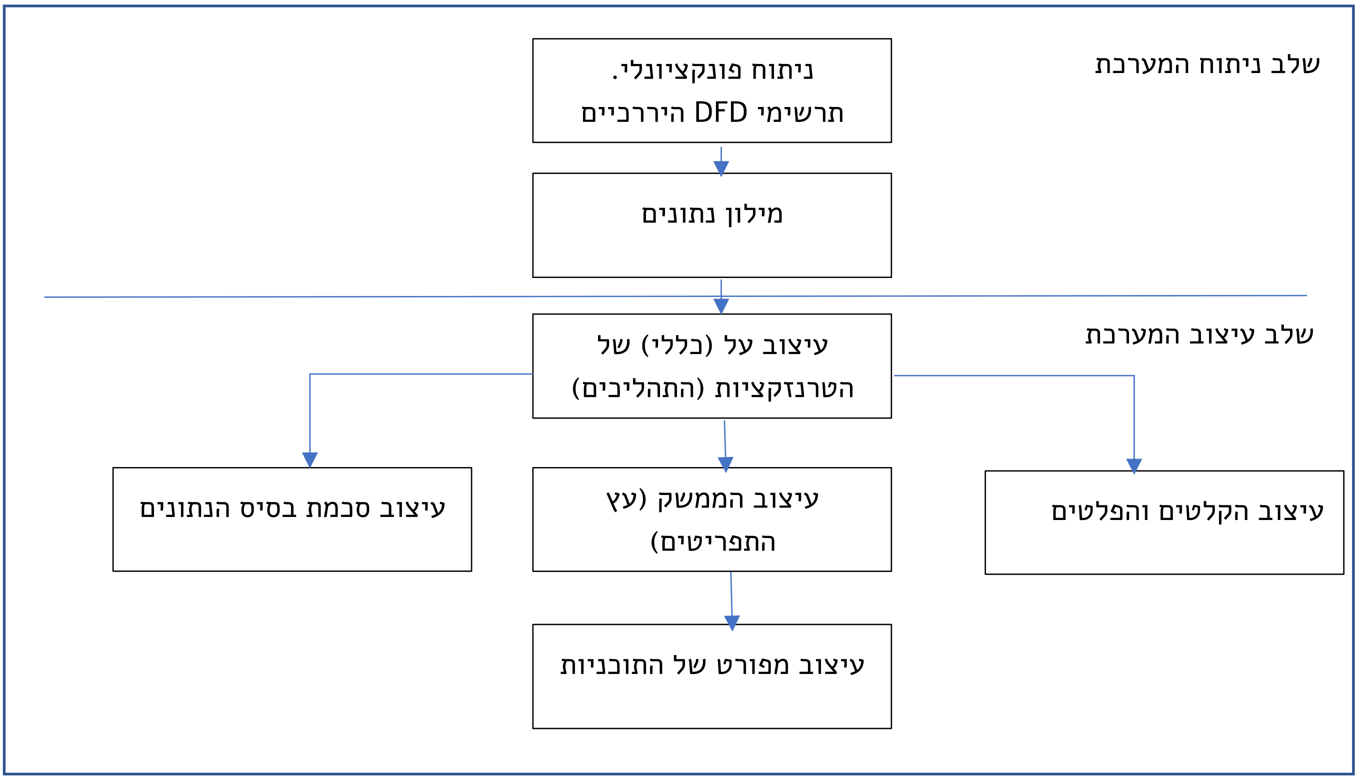
על מנת לוודא שכל זה יבוצע באופן נכון, נלמד שיטות לניתוח מערכות מידע – וראשית, כיצד לתאר את המערכת בצורה נוחה ונכונה.

שיטת ADISSA

Architectural Design of Information System based on Structure Analysis

שיטה לניתוח ועיצוב מערכות מידע, שפותחה על ידי פרופ' פרץ שובל, הנגזרת משיטת SSA שהייתה מיושנת ומסורבלת והתמקדה בעיקר בצד החומרתי של התכנה, ולכן גם פחות שימושית בקנה מידה רחב, ועל שיטה זו, בנה פרופ' פרץ את הקומה הנוספת שתומכת באופן תוכניתי יותר. ההתמקדות במודל הינה על המעבר הטבעי בין השלבים השונים

מבנה המודל:



[5] מצגת 3

החלק ראשון – ניתוח פונקציונלי – אנחנו בונים תרשים DFD (Data Flow Diagram) היררכי המתייחס לפונקציות השונות. וברגע שנראה את התרשימים האלה, נבין גם את החלק של מילון הנתונים.

תרשים DFD

אמצעי גרפי לתיאור פעולות – כל קו וקו מגיע עם חוקים המתארים פונקציונליות מסויימת. המטרה היא לקבל את המידע הזורם. אין בו לוגיקת ביצוע או מה מוביל למה, אלא תמונה סטטית של הדרך בה אנחנו מקבלים ולא אנחנו מוצאים את המידע. בסופו של דבר, ההירכיה של המודל הזה לא אומר שהפונקציות העליונות יותר יישארו למעלה גם בחלק התכנותי.

יש הרבה שיטות של DFD, אנחנו נעבוד עם גישת **דה-מארקו**, שתומכת ביצירת תרשימים היררכיים ולא רק שטוחים. מה ההבדל בין מודל היררכי לשטוח?

למשל: נרצה לצייר את כל רחובות העולם. אפשר להתחיל בלצייר את היבשות, אחרי זה לקחת יבשת ולהתחיל לצייר את כל הארצות היושבות עליה, ולהמשיך לרדת לרזולוציות שונות של ערים ושכונות עד שנגיע לכל הרחובות הדרושים.

באופן דומה אם אני רוצה לנתח מערכת, נתחיל מהרעיונות הכלליים יותר של המערכת, ונרד מטה עד הפונקציות הכי הקטנות. אך אם נעשה בכה (שיטה זאת מיושמת על ידי המודל של **גיין-סרסון**) יהי לנו הרבה בלגאן ולא נוכל לראות דברים כמו שצריך. שיטה זאת מובילה אותנו לתרשים אחד ויחיד ששולט בכל המידע, אך שליפת המידע וריבוי הפרטים המופיעים על התרשים הזה, גרומים לו להיות בלתי מובן ולא נח.

לעומת שיטה זו, דה מארקו, מדבר על הירכיה של אבות ובנים. אנחנו לא נידרש להתחיל מלמעלה, אלא ליצור אבחנה היררכית שמתפרסת על פני כמה תרשימים, כאשר לכל אחד מהם יש איזה יחס ברור בשאלה מי מוביל מעל מי. אנחנו נתחיל מתרשים שיוגדר להיות תרשים הבסיס, נחלק אותו למספר בנים, ואז נתמקד בבן אחד. כאשר נרצה להגיע לאחד הבנים הקטנים יוצר, הדרך שלנו להגיע אליו תהיה ברורה ומסודרת.

בפועל - שורש התרשים יהיה DFD-0, שיכיל את הפונקציות העיקריות לבניית המערכת . איך נמצא את הפונקציות האלו? נסתכל על המחלקות השונות במערכת, ומתוך זה נגזור את הפונקציות הכללית השונות ואת המחלקות השונות שאמורות להיות ביחסים ביניהם. למשל: אם נרצה לנתח את המכללה, נעבור בין המשרדים השונים, ונבדוק עבור כל משרד, לאיזה מחלקה הוא שייך. בסופו של יום, נקבל ראייה כללית על כל המחקות השונות במכללה, ואותם נגדיר כבנים של ה DFD-0. בתור "ניהול לוגיסטיקה", "ניהול אקדמי" וכו', כך שאם מישהו יחפש משהו ספציפי יהיה ברור לו אם הוא צריך ללכת ללוגיסטיקה או אקדמיה. בנוסף, נמספר את כל הפונקציות בתוך הDFD באופן מסוים. אין צורך שזה יהיה דווקא מספור בסדר עולה או כל דבר אחר, אלא שמספרים לא יחזרו על עצמם, או במילים אחרות שהמספור יהיה חח"ע.

את הפונקציות בתוך המסגרת של הDFD נסמן בעיגולים, כאשר אם נראה פונקציה שאמורה להתפרק הלאה נסמן אותה בעיגול כפול. כמובן, שבשלב ההתחלתי יהיה לנו בעיקר פונקציות כפולות, אבל לאט לאט נגיע בעומק לסוף הפונקציות שיוגדרו על ידי עיגול בודד.

למשל: ניקח את תרשים הDFD-0 הזה שייצג לנו את ניהול המכללה. ברור לנו שלא יהיה כאן אף אחת מהפונקציות שהיא

סופית. כמו כן, לא פירטנו פה עדיין את הישויות החיצוניות ואת זרימת המידע בין הפונקציות השונות. עשיו ניקח כל אחד מהעיגולים וננסה להבין למה ניתן לפרק אותו הלאה – למשל "ניהול סטודנטים" – בוודאי שיתווסף אליו הוספת סטודנט, מחיקת סטודנט, רישום לקורסים וכו'. לאחר שאנחנו בטוחים שרשמנו את כל הפונקציות הנוספות, אנחנו בונים את תרשים המידע הבא שייקרא DFD-2. למה 2? כי זה היה מספר הפונקציה בתרשים האב. כעת, כל פעם שנרצה להגיע אל התפריט של ניהול הסטודנטים נחפש את 0.2.

עכשיו נמשיך ונחקור את הפונקציות השונות באופן כללי יותר, ונמשיך לרדת עבור כל פונקציה עד שנגיע לשלב הסופי.

ישנם שלושה כללי אצבע, עבורם נוכל להגיד שבכל הנראה הגענו לפונקציה יסודית ואין לנו הרחבות נוספות אליה:

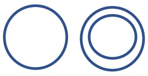
- אם ניתן לתאר בצורה פשוטה ומילולית מה הפונקציה עושה.
- אם אי אפשר לפרק את הפונקציה ליותר משלוש פונקציות משנה.

כמובן, שאנחנו לא שואפים להגיע לעץ מאוזן או משהו בסגנון, אלא לפרישה רחבה וממצה בכל האפשר.

תרשים DFD מרכיביו וחוקיו

יש לנו תרשים בודד, ויש לנו את הקשר בין התרשימים השונים. תנחיל בבניית תרשים בודד, ואז נעבור הלאה לקשרים בין האבא לבן, ובין כל המערכות השונות

המרכיבים השונים תורמים גם לפונקציונליות וגם לזרימת המידע.

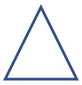


עיגול – פונקציית מערכת. מתאפיין על ידי עיגול אחד או שניים. מי שמציין את העיגולים זה מספר העיגול שצריך להיות חח"ע אך לאו דווקא עוקב (זה המפתח של הפוקנציה). פונקציה מורכבת תסומן על ידי שני עיגולים. כל פונקציה גם תתואר בצורה פשוטה מה היא עושה – קלט למשל, בלי פירוט רב מידי, וזה גם יהיה ה"שם" שלה – כמו שאמרנו, המפתח של הפונקציה הוא המספר ולא השם, ולכן יכול להיות שבמחלקות השונות יהיה את אותו שם לפונקציה.

מלבן – ישות חיצונית/ משתמש במערכת - כל האנשים (בדרך כלל אנושיים, אך לאו דווקא) שנמצאים מחוץ למערכת, המספקים קלט או מקבלים את הפלט שהמערכת מספקת. גם היישויות יסומנו במספרים בתיאור של היישות עצמה ומה היא מייצגת. (מופיע מחוץ למסגרת הDFD ומקושר על ידי חץ מידע). יכול להיות שאותו יישות גם יכניס קלט וגם יקבל פלט. אנחנו לא מדברים על אדם ספציפי אלא על יישות כללית, אם בסופו של דבר זה יהיה שני אנשים שונים זה לא משנה לנו. אנחנו נסמן את זה כאותו סוג יישות. יש לשים לב – היישות אינה מי שמפעיל את המערכת אלא מי שמספק את המידע – אם יש לנו מערכת של ניהול סחורה בסופר, ויש עובד אחד שאחראי להכניס את כל החוסרים למערכת, הוא אינו היישות אליה אנחנו מייחסים. אנחנו מדברים דווקא על העובד שמסדר את המדף של השתייה ורואה שחסר שם בקבוקים של סופר דרינק.

מלבן מאורך (פתוח בצד אחד) מאגר מידע – כל פעם שכותבים מידע זה מתווסף למאגר מידע שמוגדר לאיזה סוג מידע הוא שומר. המאגר יופיע בתוך הDFD והפונקציות השונות יקושרו אליו עם חיצים על מנת לתאר מי מכניס אליו מידע ומי קורא ממנו. כמובן שהרבה פונקציות יעברו

למאגרי מידע, ייקחו מהם קלט ויפלטו אליו בחזרה. יש לציין, שברמת התכנון זה פחות משנה לנו איזה סוג של מאגר מידע אנחנו משתמשים. אם זה טבלה או רשימה או כל דבר אחר, אין לזה שום משמעות מבחינתנו. על מנת לדעת מה כל מאגר נתונים מחזיק, אנחנו שוב נמספר כל אחד ונכתוב בצורה תמציתית מה מאגר המידע הזה מכיל.



משולש – יחידת זמן - ישויות שונות לגמרי ממה שהכרנו עד כה. תחת המושג הכללי של יחידות זמן יש לנו שני סוגים – T – ישות זמן שנמצאת מחוץ לתרשים מצד שמאל, והיא הנציג של כל הפעולות שמתבצעות בצורה אוטומטית. למשל: דו"ח יומי, דו"ח חודשי וכו' או כל מיני תהליכי אצווה שאמורים להתבצע בזמן שהמחשב פחות בשימוש. החץ הנמשך מאותה יחידת זמן לעבר הפונקציה המתאימה יציין את מרווח הזמן הדרוש לאותה משימה, או תיאור הזמן הספציפי.

זמן אמת – לא מדובר במערכות לזמן אמת ולזמן קריטי שאנחנו רגילים לדבר עליהם בדרך כלל. הכוונה כאן היא יותר לעניין חומרתי, כמו סנסורים או דברים אחרים שנותנים מידע למערכת. יישות זו באה לידי שימוש הרבה פעמים כאשר יש לנו מערכת שהיא בעיקר חומרה אך מנוהלת על ידי מחשב כמו מערכת השקייה אוטומטית. יש לנו סנסורים שבודקים לחות בקרקע, ורק כאשר הנתונים מתאימים, המחשב יכול לעבוד הלאה ולתרום את ההשקייה הדרושה.

חץ – זרם מידע בין פונקציות ורכיבים אחרים. החץ יסומן בכיוון זרימת המידע, אם מדובר בקלט לתוך פונקציה, אזי החץ יצא ממאגר הנתונים או היישות המעניקה את המידע לתוך הפונקציה, וכן להיפך. יכול להיווצר מצב, שיש יישות שגם מביאה מידע וגם צריכה לקבל מידע אחר בחזרה. במקרה כזה היישות יכולה להופיע פעמיים – פעם בצד הנותן ופעם בצד המקבל. כמו כן, נשתדל מאוד שלא להכניס לפונקציות יותר מזרם מידע אחד, בשביל לא לסבך את המידע, אבל כמובן שהפלט כול לצאת לכל מי שדורש אותו. המידע על החץ יציין מה בדיוק האינפורמציה המועברת מצד לצד. בדרך כלל נשים מצד שמאל את הקלט ומצד ימין את הפלט.

מאחר והתרשים בסופו של דבר הוא מופשט, יכול להיות שמספר תרשימים יהיו שקולים עבור מספר תרחישים שונים. משיכת כסף מכספומט ובדיקת נתונים של סטודנט יכולים שניהם להיות תחת אותו תרשים בדיוק.

מסיבה זו, חייבים להסביר את התרשים תוך כדי הכתיבה שלו.

בעת נעבור כל חלק בנפרד ונציין בדיוק את כל החוקים שלו. חלק גדול מהדברים נאמרו כבר קודם, אך בעת נכתוב זאת בצורה מסודרת. ברגע שנבין את כל החוקים, נוכל לראות בוודאות מה אנחנו עושים, וכיצד לכתוב כל דבר.

הסבר חוקי המרכיב פונקציה

1. מסמלת פונקציה שהמערכת מבצעת – יכולה להיות מורכבת או פשוטה.

אם אנחנו רוצים ללכת לבדוק איך ניתן למצוא את הפונקציות, ואילו פונקציות דרושות מאיתנו על מנת לבנות את המערכת, הולכים למפעל או למקום עליו רוצים לבצע את הניתוח, ובודקים שם את המחלקות השונות ולפי זה נחלק את הפונקציות השונות. כל "דלת" תהיה פונקציה מורכבת או פשוטה, על פי הניתוח הסופי.

2. איתור ותיאור פונקציות של מערכת מידע הוא תהליך המלווה ב"פירוק הירכי" של פונקציות החל בפונקציות כלליות וכלה בפונקציות פשוטות, יסודיות. אחרי שנאסוף אץ כל הפונקציות שהחלטנו שנצטרך, נתחיל לחלק הכל לתתי פונקציות, עד שנחליט את העומק הדרוש לפונקציות.
3. לכל פונקציה בDFD צריך לתת מספור חח"ע שהוא יהיה המפתח שלו. נועד לזיהוי ולא לסדר ביצוע. אין חשיבות לסדר, אלא רק שאם יש לנו 0.6.1, שיהיה הפונקציה היחידה שתגיע לאותו מקום.
4. תהיה כותרת המתארת את הפונקציה בצורה מילולית. כאמור, השם של הפונקציה יכול לחזור על עצמו בכמה מקומות שונים ואין לנו בעיה עם זה. בדרך כלל, ככל שהפונקציה יותר כללית, היחס לשם הוא יותר כשם עצם ("ניהול סטודנטים"), וככל שיורדים לכיוון הפונקציות המפורטות יותר, השמות הם יותר משפטי ציווי ("עדכון סטודנט").
5. כמה פונקציות יופיעו בכל תרשים? אנחנו לא רוצים להעמיס 100 פונקציות בתוך תרשים אחד, כי בוודאי שזה יהיה לא נוח. אדם יכול להתסכל בו זמנית על בין 5 ל12 פונקציות בלי להתבלבל. פחות מזה, הפונקציה לא דורשת פירוק – אם יש לנו רק שלוש פונקציות, הן יהיו פונקציות סופיות ולא נמשיך לפרק אותן. אם יותר מזה, חייבים לפרק למספר קבוצות בשביל שניתן יהיה לעקוב בצורה נוחה.
6. לכל פונקציה חייב להיות זרם מידע אחד או יותר שנכנס, והיא חייבת לפלוט מידע הלאה. המידע יכול להגיע ממאגר או מישות אחרת חיצונית לתרשים.
7. מתי תהליך הפירוק נעצר? כמה רמות של היררכיה צריך? ברגע שמתחילים לדבר בכיוון של לוגיקה תכנותית, ניתן לעצור ולפשט. אם אנחנו מדברים על פונקציה שלוקחת שני מספרים ומחזירה את הכפל ביניהם, אז אין לנו לאן להמשיך.
8. אסור שיהיה פירוט יתר – אין צורך לרדת עד פרטי הפרטים הקטנים ביותר של התהליכים כמו "עדכון כתובת של סטודנט", אלא להסתפק ב"עדכון פרטי סטודנט". בנוסף, לנו כמתכנתים, ברור שכל פעולת עדכון או שינוי במידע חייבת להגיע גם עם חיפוש, שיבדוק אם בכלל המידע הדרוש לעדכון נמצא במאגר המידע. אבל בזמן שאנחנו עושים את התרשים, אנחנו לא יורדים לרזולוציה הזאת של כל השלבים, אלא פשוט מדברים על עדכון, ומכוונים להוציא ידי חובה את כל שלבי הביניים.
9. אין לוגיקה בDFD, אין סדר שמתחיל פעולות. התרשים עצמו, כמו שכבר אמרנו קודם הוא סטטי, אך בנוסף לכל האמור, יש רק שני מקרים בהם יש לוגיקה – חץ – מוביל ממקום למקום. אם פונקציה שולחת חץ לפונקציה אחרת, אז השניה לא תרוץ ללא הראשונה.
10. אסור לעשות פונקציה שמזינה את עצמה. דבר כזה הוא יותר מידי "תכנותי", ונראה כמו לולאה, אותה כבר אמרנו שאנחנו לא רוצים לראות בתרשים.

הסבר חוקי המרכיב ישות[6]

1. המשתמשים יכולים להיות עובדי הארגון או גורמים מחוץ לארגון שיש להם קשר עם המערכת.
2. משתמש אינו חלק מהתוכנה! יופיע מחוץ למסגרת התרשים. המיקום שלו ייקבע על פי הדרישות (יוסבר בהמשך).
3. המשתמש אינו מפעיל המערכת אלא הגורם המבצע ונזקק לסיוע המערכת. אמרנו את זה כבר קודם – אם אנחנו לוקחים מידע מספק או עובד בחנות, הם המשתמשים של התוכנית ולא מי שיושב ומקליד את הנתונים.
4. שם הישות אינה אדם, אלא תפקיד. זה לא מנשה מהנהלת חשבונות, אלא הנהלת חשבונות.

---

[6] מצגת 4

5. ישות חיצונית היא מספקת נתונים. לכן יהיה צורך לתכנת ממשקים בין המשתמש לתבנה עצמה, כל ממשק ייוצג על יד חץ.
6. ישות חיצונית למערכת מזוהה על ידי האות E ומספר ייחודי. כמובן שאין חשיבות לסדר המספרים כל עוד אנחנו מדברים על מספור חח"ע.
7. ישויות נמצאות משני צידי התרשים לפי התפקידים שלם כמקבלות / פולטות מידע.
8. אפשר לקבץ ישויות בעלי מבנה משותף לישות כללית ולפרט בהמשך את ישויות המשנה. למשל: עבור ניהול הלקוחות נשתמש בהתחלה בישות הראשית "לקוח", וכאשר נרד למטה ונתחיל לפרט על הטבות שונות נכניס שם את ה"לקוח VIP", ובן על זה הדרך.

הסבר חוקי המרכיב מאגר מידע

1. מקום אחסון של נתונים המצטברים, ומעריכים כי יהיה בהם צורך בהמשך. בסופו של דבר נקרא ונשנה בכל מאגרי המידע השונים, והם לא שם רק בשביל הפנים היפות שלהם.
2. כל מאגר מידע מזוהה על ידי האות D שאליה צמוד מספר זיהוי ייחודי, כמובן חח"ע.
3. מאגר מידע חייב להיות קשור לפחות לפונקציה אחת שקוראת ממנו, ואחת שמעדכנת בו נתונים. בפעם הראשונה שהוא יופיע, בדרך כלל בתרשים 0, או נמוך קצת יותר אם הוא פנימי, הוא חייב לפחות פונקציה אחת שולפת מידע ואחת מזינה, ולא יכול להיות רק אחד מהם. בהמשך ניתן להתפשר על פחות מזה.
4. פעולות על מאגר: עדכון / שליפה. כמו שצויין כבר קודם – אנחנו לא מתחילים לעשות חיפושים במאגר, לפחות לא בצורה גלויה. אנחנו נסתפק בשתי הפעולות הללו, ונתבסס על כך שבוודאי כל מה שקורה ברקע, נשאר ברקע.
5. מאגר מידע יכול להיות קשור אך ורק לפונקציות. ישות לא מעדכנת מאגר מידע, מאגר מידע לא מעדכן ישירות לאחד אחר, אלא משתמשים בפונקציה מגשרת בשביל לעשות זאת.
6. מאגר מידע שמשומש יותר מפעם אחת באותו תרשים מסומן בקו אלכסון.
7. מאגר מידע חיצוני, בדוגמת אינטרנט, יצייר מחוץ למסגר בצד ימין או שמאל, על פי הפעולה הנעשית בו (קריאה – ימין. עדכון – שמאל). אי נדע מה בפנים ומה בחוץ? אם מאגר המידע כבר קיים, ואנחנו רק מתמשקים אליו, הוא חיצוני למערכת.

הסבר חוקי המרכיב יישות זמן

1. ביצוע פעולות באצווה - פעולות שמתבצעות ביוזמת המערכת ולא ביוזמת המשתמש, במועד שנקבע מראש. יש לרשום על החץ את הזמן המוגדר. אם מדובר על פונקציה כללית לא ממספרים את ישות הזמן, אך רושמים את מועד ההפעלה. אם רושמים בצורה כללית, אחרי זה רושמים את המימושים הספציפיים. בדומה לצורה שסימנו את הישות הכללית, כך נעשה גם במרביב הזמן.
2. הזמן הוא טריגר שמפעיל פונקציה ללא שום התערבות מצד המשתמש
3. יזם של זמן נמצא אך ורק משמאל למערכת, כי הוא מפעיל את המערכת ולא מופעל על ידה.

הסבר חוקי המרכיב יישות זמן אמת

1. מערכות שקשורות למכשירים ולאמצעים מיוחדים, כגון סנסורים, מכשירי מדידה שונים וכו'.
2. מסמנים עם משולש עם האות R ובצמוד לו מספר זיהוי (כן, חח"ע).
3. יכול להופיע מצד שמאל ומצד ימין, על פי זרימת המידע.

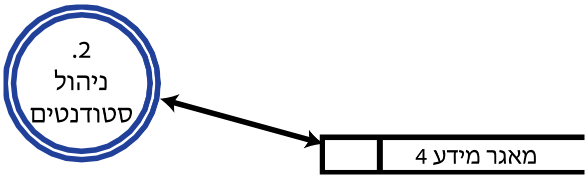
הסבר חוקי המרכיב זרם מידע

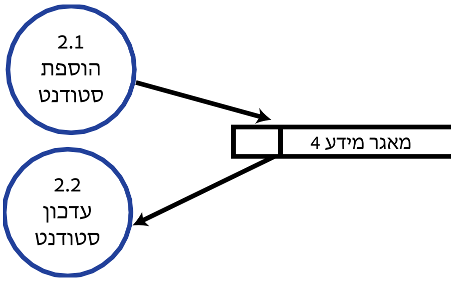
1. על זרם המידע יופיע המידע המועבר מקצה לקצה, כאשר אם אנחנו רוצים למשל להוציא מידע מפונקציה של נתוני דו"ח או משהו דומה, המכיל מספר פרטים, אנחנו לא נפרט על החץ על כל הפרטים השונים הקשורים לזרם המידע, אלא רק נרשום באופן כללי את "נתוני הדו"ח". את שאר המידע היותר מפורט נכתוב במילון הנתונים.
2. אחד מקצוות החץ הוא עיגול של פונקציה. יכול להיות שמידע יועבר בין שתי פונקציות שונות, ושני הקצוות יהיו עיגולי פונקציה, אך אין מצב בו מידע יעבור ממסד נתונים ליישות או משהו דומה ללא פונקציה מתווכת.
3. לזרם מידע יש שם. השם אינו ייחודי ולא אמור להיות חח"ע. הייחוד של הזרם נעשה (כמו שציינו קודם) באמצעות הרכיבים של הזרם בקצוות , למשל: זרם מידע בין פונקציה 2.2 ליישות 4. המספרים האלה ייחודיים ולכן ניתן להשתמש בהם.
4. כמו שעשינו הבחנה בן פונקציות ייסודיות לכלליות (כאלה שמתפרקות למספר פונקציות ייסודיות), כך נעשה גם הבחנה בין <u>זרם מידע יסודי</u> שלפחות באחד מהקצוות יש פונקציה יסודית, לבין <u>זרם מידע כללי</u> העובר בפונקציות כלליות.

   כמובן, שאם זרם עובר בין פונקציה ייסודית לכללית, הזרם ייחשב כללי – משמעות ההבחנה היא תשומת הלב לפירוט בהמשך, שנצטרך לכתוב את הזרם ברמות הנמוכות יותר.
5. החיצים במהותם הם חד-כיווניים, וכבר דיברנו על זה בחלקים הקודמים. אך יש לציין שזה רק כשמדובר על חצים שמובילים לפונקציה ייסודית (או יוצאים מאחת כזאת).

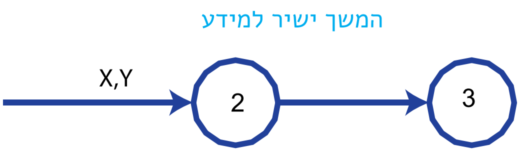
   ישנם מקרים של פונקציה הזרם יצוייר באופן דו כיווני – למשל כשאנחנו רוצים לבטא מעבר של מידע לפונקציה כללית. יכול להיות שנרצה לקרוא מידע, לעדכן אותו ולהחזיר בחזרה למאגר המידע. במקרה כזה, הפונקציה הכללית תסומן בחץ דו כיווני, וכשנרד מזה ברמות השונות של ה-DFD נוסיף לפונקציות הייסודיות את החיצים החד כיוונים.

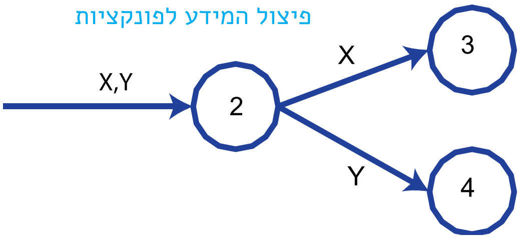
   לדוג'





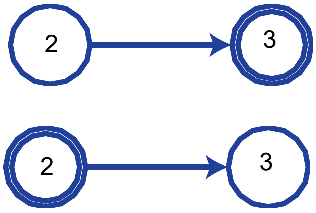
   בטבלה של הס-DFD נייצג חץ דו כיווני, שיבטא שיש מעבר של מידע לשני הכיוונים, אך בפונקציות היסודיות יותר, נוודא שיהיה רשום לנו בדיוק מי קורא מידע ומי כותב מידע למאגר.
6. מידע יכול לעבור בדרך ישירה בין שתי פונקציות. אם כל המידע עובר בשלמותו (גם אם נעשה בו שינוי בדרך), אין צורך לכתוב מחדש על החץ את השם של המידע שעובר. אם אנחנו מפצלים את המידע לפונקציות שונות, יש צורך לציין מי מקבל איזה חלק של המידע.





7. על מנת לחבר בין שני תרשימי DFD שונים, אנחנו משתמשים בקונקטורים. לפי כל מה שאמרנו בסעיפים הקודמים, אין שום מניעה שמידע יעבור בין שתי פונקציות כלליות. הבעיה מתעוררת כאשר אנחנו רוצים לבטא את מעבר המידע בין הפונקציות עצמן. אין לנו שום אפשרות למתוח קו שיצא מפונקציה 2.3 לפונקציה 10.7. כיצד נבטא זאת? נסמן אליפסה קטנה המקשרת בין הפונקציות השונות – כל פעם בצד המתאים לקבלה או העברה של מידע, ובתוכה נסמן את הפונקציה אליה אנחנו מקשרים.

במובן שזה פשוט כאשר מדובר בשתי פונקציות כלליות, אך מה קורה אם אחת מהפונקציות היא יסודית ואיזו היא כללית? במקרה כזה, יש לשים לב לכיוון זרם המידע. צריך לזכור, שברגע שאנחנו מדברים על העברת מידע מ/אל פונקציה כללית, הכוונה היא שיש אי שם פונקציה יסודית אחרת שהיא זאת שמטפלת במידע הזה, אבל אנחנו לא יודעים מי, אז היכן נשים את הקונקטור? **הכלל הוא שהקונקטור יופיע בצד של הפונקציה הכללית**. מאחר ובצד היסודי ברור לנו מה קורה – המידע נשלח/מתקבל למקום מאוד מדויק, אך דווקא בצד השני אנחנו לא יודעים מי מטפל במידע. ולכן -



אם פונקציה יסודית מעבירה מידע לפונקציה מורכבת באופן הבא: אז פונקציה 2 שולחת את המידע ל-3. כאשר בתוך DFD-3 הפונקציה המתאימה תקבל את הקונקטור שיתחבר אליה היישר מפונקציה 2.

לעומת זאת, במקרה ההפוך:

DFD-2 יוציא את הקונקטור (מצד ימין של יציאת המידע), כאשר הקונקטור יחובר לפונקציה היסודית המתאימה ויהיה כוה עליו 3 לציון הפונקציה אליו הוא מתחבר.

8. קשר לוגי or/and – אמרנו כבר שתרשים הDFD בא לתאר לנו מצב סטטי. בתור שכזה, אין לנו יכולת לדעת מי ראשון ומי אחרון. אך היוצא מן הכלל הם הפונקציות המסומנות ביניהם בתרשים זרם מידע. כי הרי ברור שפונקציה שמקבלת מידע לא יכולה להתחיל את הפעולה ללא המידע שיועבר אליה.

בנוסף, ניתן להשתמש בהרחבה נוספת של קשרי AND וקשרי OR

קשר AND – חיבור של שתי פונקציות (או יותר) לפונקציה אחרת. לא משנה אם מדובר בקשר שמגיע לפני או אחרי פונקציה, הוא מסמן על כך שכל הפונקציות המקושרות חייבות לעבוד, בין אם מדובר בתנאי מסוים שחייב לחול, או כמה כאלה בשביל להפעיל פונקציה מסוימת, ובין אם פעולה של פונקציה אחת מפעילה כמה אחרות, קשר ה-AND דואג שכולם יפעלו.

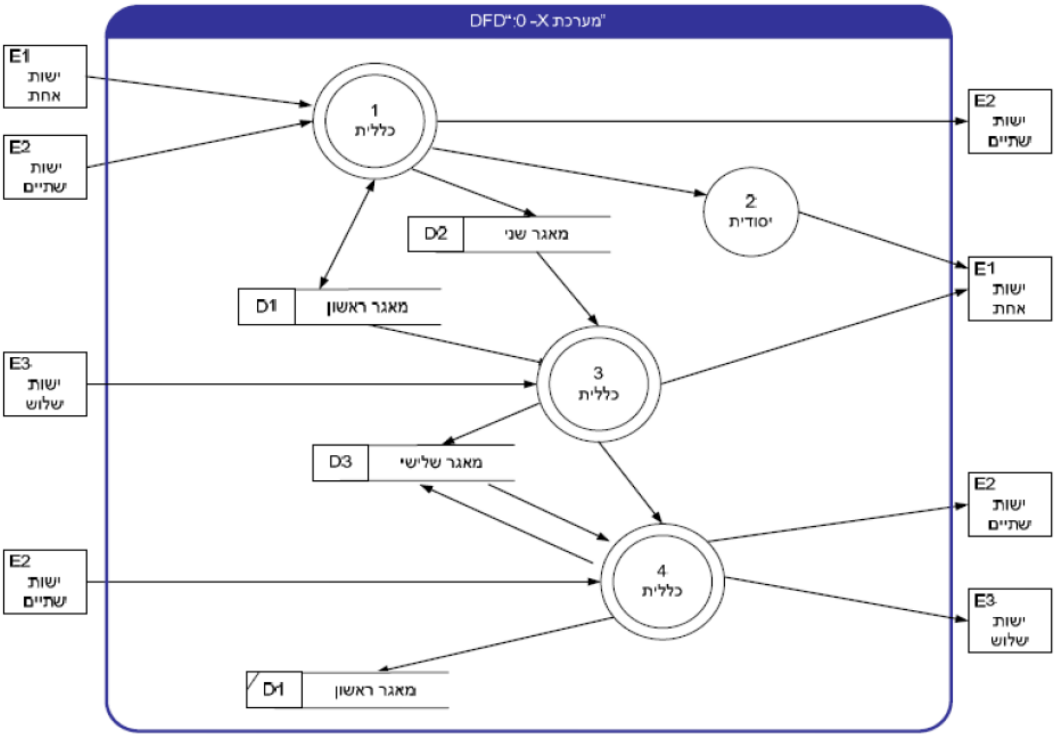
בדרך כלל אם לא מסומן אם מדובר בקשר OR או AND נהוג לומר שהקשר הוא AND, אבל יותר פשוט זה לשאול למה הכוונה של הקשר.

קשר OR – מדבר על הפעלה של פונקציה **אחת בלבד** מכל הקישורים לאותה פונקציה. חשוב להדגיש את זה כי לא מדובר פה על קשר OR לוגי שראינו עד עכשיו, אלא הוא יותר נכון לתיאור ב XOR. שוב – מדובר על פונקציה את, לא אחת או יותר.

חוקי הקשר בין תרשים האב לתרשים הבן

חלק זה לא ייתן לנו נקודות סדורות כמו בחלקים שעסקנו בהם עד כה, אלא הוא מעין סיכום של כל החוקים, שבעצם מוביל אותנו לחוקים של הפירוק של תרשים האב לתרשימי- הבן הממשיכים אותו. התהליך של המעבר מהרישום הכללי של תרשים האב לבנים השונים נקרא "פירוק". ועיקרו הוא פשוט התייחסות לחוקים בירידה מטה.

לדוגמא ניקח את המערכת X. המתוארת באופן הבא:



תרשים זה הוא הDFD-0 של המערכת, וניתן לראות בו את הפריטים הבאים:

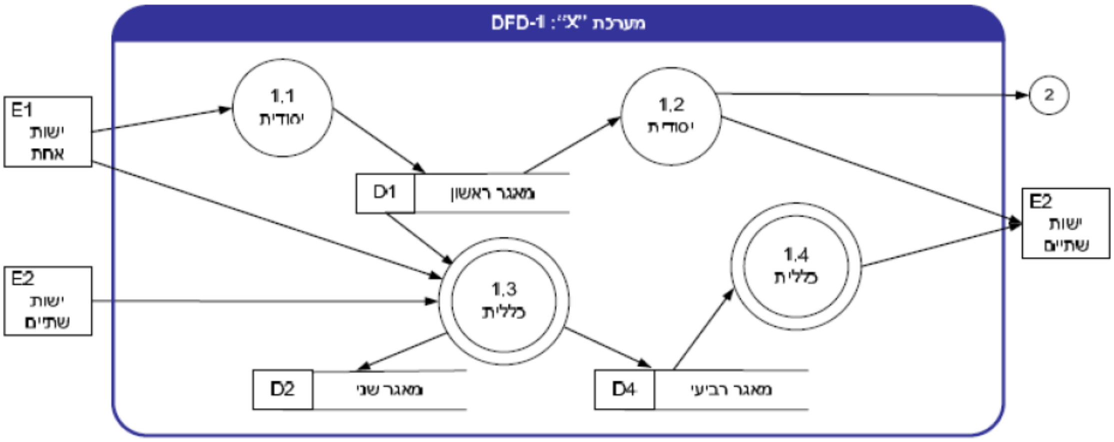
<u>פונקציות:</u> 4 פונקציות שונות, מתוכן פונקציה 2 יסודית, והשאר כלליות

<u>יישויות:</u> 3 יישויות שונות המקבלות ונותנות מידע לפונקציות השונות. יישות 2 לדוגמה, מעבירה מידע לפונקציה 1, ואחר כך מקבלת מידע (מעודכן/ערוך) בחזרה, ועובדת באופן דומה גם עם פונקציה 4.

<u>מאגרי מידע:</u> 3 מאגרי מידע. כולם מקבלים מידע ומעבירים מידע לפונקציות השונות. כאשר D1 שעובד גם עם פונקציות 1 ו-3, מקבל מידע לאחר מכן (ניתן לראות את הסימון של מידע מעובד) מפונקציה 4.

<u>זרם מידע:</u> מספר זרמים שעוברים בין הפונקציות השונות. ניתן לראות שהזרם בין פונקציה 1 לD1 הוא דו כיווני, וכל השאר הם חד כיווניים.

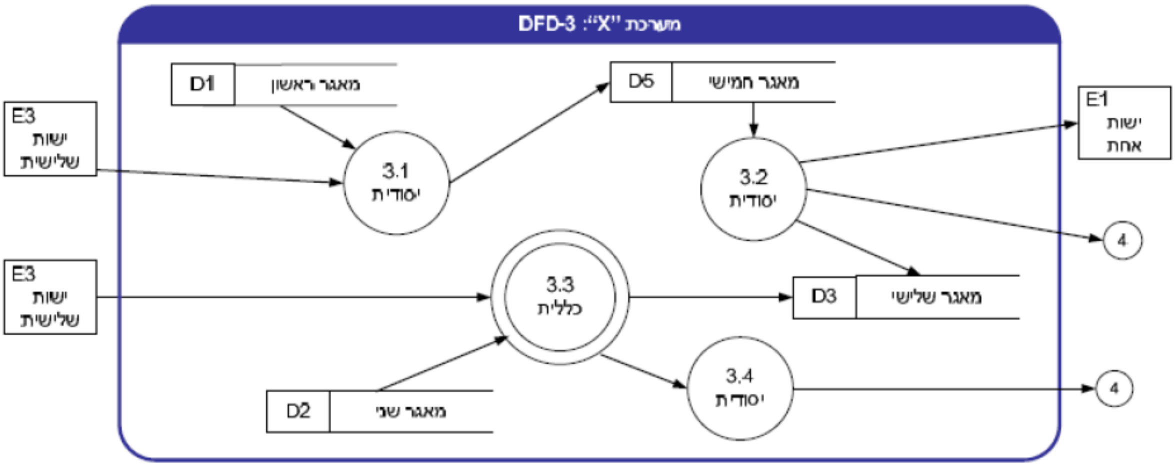
כבר עכשיו אנחנו יכולים ללמוד הרבה ולקבל מושג כללי כיצד ייראו הבנים, אך נסתכל על הפירוט של שלושת הבנים (מאחר שפונקציה 2 היא יסודית, לא נבנה לה כמובן DFD).



עכשיו כבר ניתן לראות ביותר פירוט את מעבר המידע בתוך הפונקציה. למשל יישות 2 שראינו שהיא גם נותנת מידע לפונקציה 1 וגם מקבלת, למעשה מעבירה את המידע לפונקציה 1.3, שמעבירה למאגר מידע D4 ורק אז עובר ל1.4, ויוצא בחזרה ליישות 2.

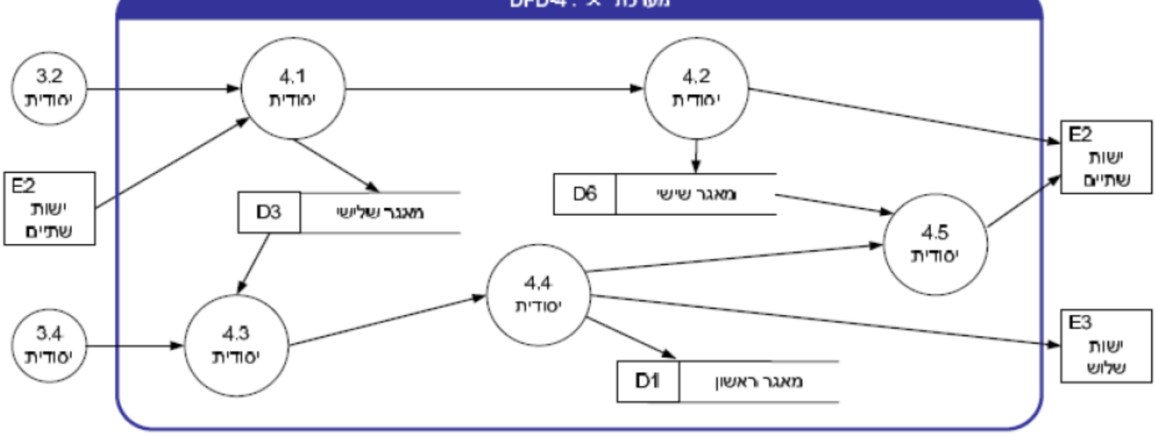
מאגר מידע 4 שמופיע פה, לא מופיע בתרשים האב, כמו שאמרנו בחוקים שלו שמותר שיהיה מאגרי מידע פנימיים לפונקציות הפועלות רק בתוך אותה פונקציה, דבר זה אפשרי בתנאי שאנחנו עושים גם קריאה וגם כתיבה לאותו מאגר מידע.

בנוסף, שימו לב לקונקטור – בתרשים האב ראינו שיש מעבר מידע מפונקציה 1 (הכללית) לפונקציה 2 (היסודית). ובהתאם לכללים שאמרנו קודם, מאחר ו-2 היא פונקציה כללית, לא יהיה תרשים מיוחד בשבילה. ולכן כאן אנחנו מציירים את הקונקטור היוצא מ-2.1 ומקושר ל-2.



פונקציה 2 היתה יסודית, ולכן אנחנו מדלגים ל-DFD3.

אין פה הרבה דברים מיוחדים לשים לב אליהם, 3E מעביר מידע גם ל-3.1 וגם ל-3.3, והוא מצוייר פעמיים. מבחינת הקונקטורים, קודם היה פשוט זרם מ-3 ל-4, אך פה אנחנו רואים שיש שתי פונקציות יסודיות שמעבירות ל4, ולכן יש לנו 2 קונקטורים ל4. שימו לב, שלא מצוין בדיוק לאן ב-4 המידע נכנס, אלא רק באופן כללי (למרות שלפי איך שהבנתי מהמצגת, זה אמור להיות כמו בתרשים DFD4 למטה, שמצויין שם בדיוק מאיזה פונקציה זה מגיע).



מודלים לפיתוח מערכות תכנה[7]

ישנם מספר מודלים (שיטות) שונות בגישה להנדסת תכנה. אנחנו דיברנו כבר בקורס המבוא על הAgile, והזכרנו במצגות הקודמות את מודל "מפל המים", ועכשיו נתרכז בעוד כמה שיטות.

השיטות הבאות מבוססות ברובן על מודל מפל המים, אך נראה את ההבדלים בכל גישה והמאפיינים השונים. בנוסף, עבור כל שיטה נתמקד במספר נקודות חשובות:

- הרעיון של המודל – מה הרציונל העומד מאחוריו.
- הסבר המודל – שיטת העבודה בפרוטרוט.
- יתרונות המודל.
- חסרונות המודל.
- שימוש במודל – לאיזה סוג פרויקטים נעדיף את המודל הזה ובאילו נעדיף שלא להשתמש בו.

לאחר שנדע את כל המרכיבים של כל מודל, נוכל לדעת באיזה מהם לבחור עבור כל פרויקט נתון.

מודל מפל המים

על מודל זה התחלנו לדבר בעבר, נדגיש את הנקודות העיקריות-

מדובר במודל קשיח, המתמקד בתכנון של כל שלב עד לפרטי הפרטים לפני שעוברים לשלב הבא.

מודל זה מתאים לעבודה בעיקר על פרוייקטים שאינם גדולים במיוחד, מאחר וככל שהפרויקט גדל, כך גדל גם הסיכוי לטעות באחד מהשלבים. הבעיה העיקרית במודל זה, שקשה לשנות בו דברים, מה שגורם לכך שאם מגלים טעות בתהליך העבודה, העלויות של התיקון עלולות להיות מאוד גדולות.

מודל "בנה ותקן"

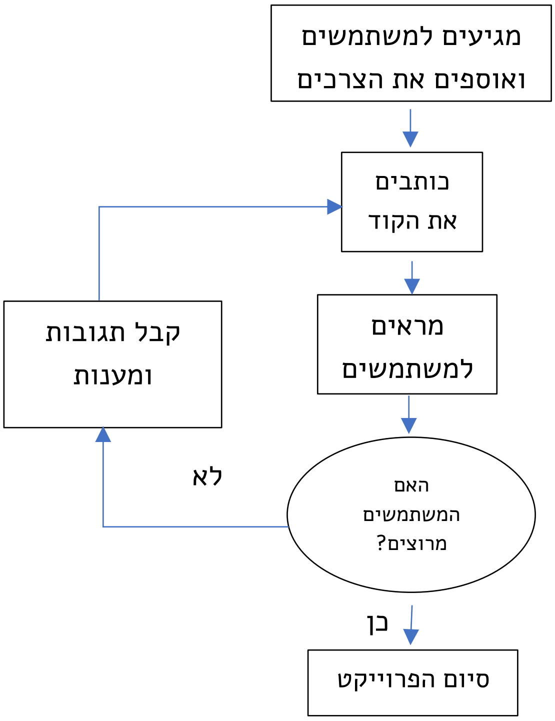
מודל זה הוא הכבשה השחורה של כל המודלים. בעוד כל השיטות מתמקדות בסדר עבודה מסוים (אפילו האג'ייל, שנותן לנו את האפשרת לעבוד באופן מהיר יחסית), מודל זה פשוט מתמקד בעשיה עצמה. לא סתם אחד הכינויים למודל זה (ע"פ ויקיפדיה) הוא "מהר ומלוכלך".

אחרי הקדמה שבזו, בוודאי נחשוב שלא משתמשים בו כמעט, אבל בעצם אנחנו (כסטודנטים) משתמשים בו **כל הזמן**.

השיטה פשוט אומרת – "קיבלת משימה, תבצע אותה. תהיה בעיה, תקן אותה".

[7] מצגת 5

הסבר המודל



פשוט עושים עד שזה עובד וסבבו על כולם.

יתרונות המודל

- אין ניירת מיותרת – לא מתעסקים עם דו"חות או פרוטוקולים שצריך לעקוב אחריהם.
- עבודה מהירה

חסרונות השיטה

- ויתור על שלבים חשובים – אמנם זה הרבה יותר נח לא להתעסק עם הניירת, אבל בסופו של דבר, היא נמצאת שם מסיבה מסויימת. אם אנחנו לא מתעסקים בלחשוב על מה שאנחנו עושים, יש סיכוי סביר שנעשה אותו לא נכון.
- לא ברור מתי לבצע משימות שונות – מאחר ואין סדר לדברים, דברים חשובים וגדולים עלולים להידחק לסוף הרשימה ללא סיבה.
- לא מתאים לקבוצות גדולות – נוצר מספיק בלאגן כאשר בן אדם אחד עובד בצורה חופשית. אם שני אנשים או יותר יעבדו בצורה לא מסודרת וללא תיאום ביניהם, ככל הנראה התוצאות לא יהיו מוצלחות.
- קשה להעריך תוצרים – אם לא קבעת יעדים, איך תדע שעמדת בהם?
- עלות תיקון בעיה עולה בכל שהיא מתגלה מאוחר יותר – דיברנו על זה בהקשר של מפל המים, אך גם בשיטה זו, שהיא ההיפך הגמור ממפל המים, אנחנו עלולים לגלות שפיצ'ר קטן שהוספנו ללא מחשבה תקע לנו את כל התוכנה לאחר זמן מה.

שימוש במודל

אחרי כל המילים הטובות שהרעפנו על המודל הזה, מתי בכל זאת כן נשתמש בו? כאשר אנחנו רוצים לבצע שינוי קטן במערכת ספציפית בתוך מערכת גדולה יותר. לחילופין, אם אנחנו עובדים על תכנה שתחום ההגדרה שלה הוא יחסית מצומצם: משתלה, מטפלות וכדו', ניתן לעבוד בשיטה זאת.

אם נעבוד על מערכות שהם גדולות ומורכבות כמו מערכות לניהול כספים, הזמנת טיסות, או דברים דומים בהם אנחנו דורשים כמה שיותר דיוק, וכמה שפחות באגים בהפתעה, נשאיר את השיטה הזאת בצד.

מדוע בכל זאת השיטה הזאת נפוצה? ברור לכולם שהשיטה הזאת היא לא ממש מתודה עובדת, אלא "קריאת שם" לתופעה גדולה שהיא לא חיובית בעיקרה. לדבר זה יש מספר סיבות:

הגורם הפסיכולוגי – בתחום החברתי, תכונות האופי של המתכנתים לא מחמיאות לנו. מדובר על אנשים הישגיים, הרוצים לראות תוצאות בשטח, לקבוע עובדות ותצפיות ולדעת בדיוק מה קורה. בנוסף, חלק גדול מהמתכנתים הם יהירים וחסרי סבלנות להסביר למשתמשים וללקוחות את תהליך העבודה שהם עושים. עקב כך, הם מעדיפים פשוט לזנוח את החלק הזה ולהתמקד ב"חלק החשוב" שזה כתיבת הקוד עצמו. אחרון ומשמעותי לא פחות – עם כל ה"פרפקציוניזם" המתכנת הוא טיפוס עצלן שאוהב להשליך את הידע שלו מתחומים אחרים אותם הוא מכיר, לתחום בו הוא נמצא עכשיו. דבר זה גורר הטמעות של פיצרים לא נצרכים והתמקדות בטפל.

גורם חינוכי – המתכנתים לא מכירים היטב את החומר הדרוש ולא מגדירים כמו שצריך את התרשימים השונים.

אופי תכנה – אנחנו מודעים לכך שיכולים להיות שינויים. עקב כך, אנחנו מקבלים את זה כחלק משלבי בניית התכנית ואפילו לא מנסים להוציא תוכנית עובדת.

כיצד נמנע מכל החסרונות של שיטה זאת?

עובדים בתהליך שחושב מראש על התהליך אותו אנחנו עוברים. מכינים את עצמנו מראש לשינויים שצצים בדרך כלל במהלך העבודה – לא הולכים ראש בקיר, ולא כותבים דברים שאי אפשר לשנות או לערוך אחרי הכתיבה, ובגדול תשומת לב לפרטים.

מודל וי V

מודל ה-וי התפתח למעשה ממודל מפל המים, אך הוא הוסיף לו נדבך. בעוד שמודל מפל המים עובד בצורה ליניארית בכיוון אחד, מודל וי עובד הלוך וחזור. אך לא כמו במודל הבנה ותקן שבל הזמן חוזרים ומתקנים עד שהתכנה עובדת, כאן מתייחסים לכל שלב וכותבים את הבדיקות המתאימות לו, על מנת לוודא שאנחנו עוברים על כל השלבי בצורה נכונה.

המודל קיבל את השם שלו, על אופי העבודה שיורד מטה (כמו במפל המים) ואז חוזר בחזרה בשלב הבדיקות.

הסבר המודל

1. ביצוע שלב הדרישות – איסוף המידע מהמשתמשים/לקוחות.
2. כתיבת הValidation Tests – כתיבת הבדיקות המוציאות לנו את התשובה האם יישמנו נכון את כל מה שנדרש מאיתנו.
3. ביצוע שלב הניתוח – מה נדרש מאיתנו ברמת התכנה
4. כתיבת System tests – סט בדיקות כללי לתכנה, המוודא שהיא עובדת על פי ההגדרות השונות.
5. ביצוע שלב העיצוב – תכנון המחלקות השונות, והמרכיבים השונים שלהם.
6. כתיבת הIntergration Tests – הבדיקות עבור כל מחלקה בנפרד, כל מחלקה מטפלת באיזור אחר של המיע שצריך לעבור, ולכן יש לוודא שהכל עובד כשורה בכל איזור.
7. ביצוע כתיבת הקוד – לאחר שברור לנו מה כל המרכיבים השונים, אנחנו יכולים להתפנות לקוד עצמו.
8. כתיבת הUnit Testing- בדיקות עבור כל פונקציה וכל חלק הכי קטן של הקוד, שמעבד את המידע בצורה הנכונה.
9. הרצת Unit Testing – לאחר שכתבנו את כל הנדרש, ואנחנו מוכנים עם כל הבדיקות שאנחנו אמורים לעשות, אנחנו מתחילים את הבדיקות ברזולוציות הקטנות ביותר, בפונקציות עצמן, כך שאם פה נראה בעיה, יהיה לנו יותר פשוט להתמודד איתה ולא נצטרך לחזור יותר מידי אחורה.
10. הרצת הIntegration Tests – הבדיקות על המחלקות השונות.
11. הרצת הSystem Units – בדיקת התוכנית עצמה
12. הרצת Validation Tests – וידוא שדרישות הלקוח מוצו ובאו על סיפוקן.

יתרונות המודל

- פשוט וקל לשימוש – אמנם הוא מורבב מהרבה חלקים, אך אם אנחנו מתמקדים בכל פעם בשלב שאנחנו נמצאים בו בצורה מלאה, הרבה יותר פשוט להעמיק בו.
- פעילויות בדיקת התכנה קורות הרבה לפני הקידוד - אמנם אנחנו לא יכולים לבדוק באמת קוד שעדיין לא נכתב, אך עצם ההתמקדות בשאלות של מה אמור להיות ומה לא, ובחינה של הטסטים שאמורים להיות בהמשך יכול לעזור לנו מלעשות טעויות שנצטרך לשלם עליהם בהמשך.
- טעויות מתגלות בשלבים מוקדמים- אם מוודאים עד הסוף את הדרישות, רוב הסיכויים שלא נצטרך לחזור אליהם בהמשך.
- עובד היטב על פרויקטים קטנים וברורים – רמות הסיבוך לא גדולות מידי.

חסרונות המודל

- מודל נוקשה מאוד – המון עבודה וכתיבה שהיא מעבר לתוכנית עמה.
- כותבים תוכנה עובדת רק בשלב הקידוד – עד שלב 7 לא כתבנו אפילו שורת קוד אחת.
- קשה לעשות שינויים בדרישות – עומק החפירות על כל שלב, גורם לכך שכל שינוי שצריך להיעשות גורר חזרה ארוכה מאוד אחורה.

מתי נשתמש בשיטה

- כאשר יש לנו פרוייקטים קטנים/בינוניים שבהם הדרישות מוגדרות היטב ויציבות – אנחנו לא חוששים שנעבוד בכיוון מסוים, ולאחר זמן נגלה לתדהמתנו שינויים במבנה התוכנית ובדרישות
- יש לנו אמון גבוה בלקוח – גם מצד זה שלא יהיו שינויים, כאמור. אך גם מהסיבה ששלב הקידוד והתכנה הפועלת מגיעים בשלבים מאוחרים יחסית, אם הלקוח לא יאמין לנו שאנחנו עובדים היטב זה לא יהיה טוב לאף אחד.

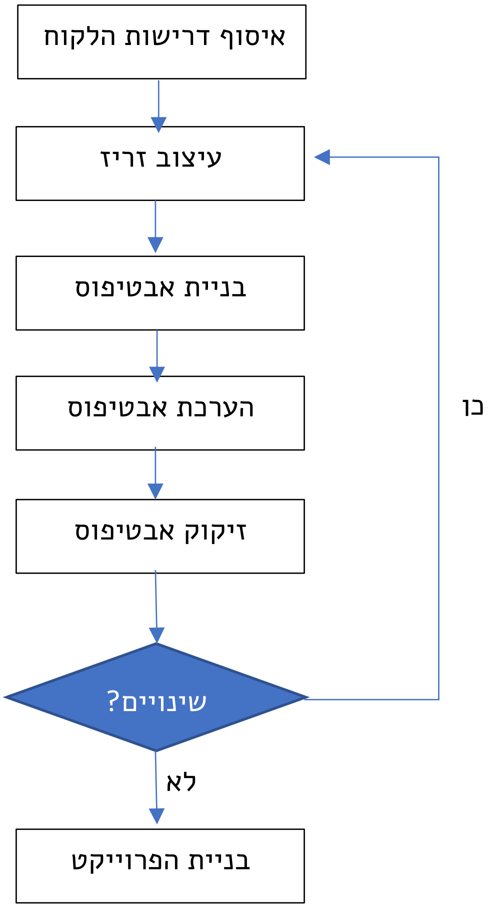
מודל האב-טיפוס

מודל אב-טיפוס, אינו ממש מודל לעיצוב התוכנה כמו כל השאר, אלא מגיע כנדבך נוסף על גבי מודל מפעל המים. דיברנו על כך שישנה סכנה כאשר הלקוח רואה את התכנה בפעם הראשונה רק כאשר היא עשויה ומוגמרת, מטרת המודל/גישה הזאת היא להעניק ללקוח איזה "הצצה" על מה שאמור לקרות עוד לפני העבודה

הסבר המודל

לאחר איסוף המידע והדרישות מהלקוח, אנחנו ניגשים אליו עם מודל בסיסי מאוד של מה שאנחנו הולכים לעשות. כמה בסיסי צריך להיות המודל? ברמה של מצגת פאוור פוינט או אפילו סקיצה על נייר. השאיפה שלנו שאנחנו נוכל לקבל אישור עקרוני מהלקוח שאנחנו עובדים באותו ראש ולעבוד בצורה בטוחה יחסית.

תרשים המודל דומה יחסית לשיטת הבנה ותקן, ונראית כך:



בכל פעם מראים את המודל הראשוני ללקוח, עד שמוודאים שאין לו עוד שינויים ואז ממשיכים לעבוד בלי שהלקוח יושב לנו על הווריד.

ישנם שני מודלים עיקריים של האב-טיפוס:

1. **מודל אבטיפוס מתפתח** – עובדים על אב-טיפוס ומראים ללקוח. אם הוא מרוצה – מה טוב. אחרת – מתקנים רק את החלקים הדרושים תיקון. וחוזר חלילה. כך ממשיכים ובונים בכל סבב תיקונים על בסיס מה שנבנה עד עכשיו.
2. **מודל אבטיפוס לזריקה** – בכל פעם שהלקוח לא מרוצה מהתוצאה המוצעת, אנחנו זורקים את כל אב הטיפוס שעשינו ומתחילים מהתחלה. העבודה נראית אמנם מייגעת ומוגזמת, אך היא מתאימה עבור פרוייקטים מסויימים – למשל כאלו שקשורים בביטחון, או בעלי חשיבות גבוהה, אנחנו רוצים להקטין כמה שאפשר את הסבירות לטעויות קטנות. ולכן, רק ברגע שכל אב הטיפוס מאושר מתחילתו ועד סופו, אנחנו יכולים להמשיך הלאה.

יתרונות המודל

המשתמשים מעורבים באופן פעיל בפיתוח – אין טעויות שנובעות מאי-הבנות מול הלקוח, אלא כל דבר קטן וכל אישור עובר דרכו.

המשתמשים מקבלים הבנה טובה יותר של המערכת שרוצים לפתח – משתמש שמעורב בשלבי הפיתוח המוקדמים, גם מבין יותר טוב את המוצר שלו בעצמו, וככה גורם גם לטעויות נמוכות יותר, מאחר שהוא מצליח לזקק בעצמו את הדרישות שלו, וגם הפתרונות יותר מותאמים ללקוח ברגע שהוא רואה את כל מה שנעשה.

ניתן לזהות בקלות פונקציונליות חסרה – כל המעבר אל מול הלקוח, יכול להפיל את ההבנה שחסר משהו בתכנה עצמה, ואולי אנחנו הולכים בכיוונים טובים, אך לא מספיק מדויקים.

חסרונות המודל

סכנה שנרד לצורת עבודה של בנה ותקן – כל החזרה הלא-מאורגנת על העבודה, עלולה להוביל אותנו בסוף לעבודה לא נכונה שאנחנו בכל פעם נבנה דבר קטן ונחזור ונתקן אותו, לא תהיה לנו צורת עבודה מסודרת.

ניתוח בעיה חלקית או לא מספקת – מאחר ואין איזה שיטה מסודרת לאיך אנחנו עובדים ומה מתקנים, אנחנו עלולים לפספס נקודות שיתגלו רק אחר כך בנקודות קריטיות.

יישום לא שלם, עלול לגרום שלא נשתמש במערכת – אם אנחנו כל הזמן נעבוד על לשפץ את אב-הטיפוס, יקח לנו המון זמן להגיע לעבודה עצמה והיא עלולה להתמסמס ואולי אפילו להתבטל לגמרי.

שימוש במודל

כאשר בונים מערכת שאמורה להתעסק עם הרבה משתמשי קצה – יש לנו חשיבות רבה לפידבק מהמשתמשים, ולכן ננסה כמה שאפשר כן ליצור איזה שהו מודל תכנוני.

ניהול סיכונים[8]

באותו שבוע שהועבר השיעור, ד"ר דיין הסב את תשומת ליבנו לאירוע חדשותי שממש קשור לעניין – חברת "המימד החמישי", שבני גנץ עמד בראשה, הבריזה על סגירת החברה ופיטור העובדים אחרי שכבר גייסו לא מעט כסף, והמוצר שלהם היה יחסית מצליח, אך בסוף 40 מיליון דולר הלכו קאפוט, וזה אפילו בלי לדבר על כל הכסף המצטבר שעבר שם, שעות עבודה וכו'.

ידוע שחלק גדול מהסטארט-אפים שקמים בסופו של דבר מפסידים כסף ונסגרים, ונשאלת השאלה, מדוע? אם יש חברה שקמה על תשתית של צרכים ודרישות, איך קורה שפתאום דברים מפסיקים לעבוד?

חלק גדול מהעניין הוא הדבר הבא שנלמד – ניהול סיכונים.

עד לתקופה האחרונה, האנשים שהתעסקו עם התכנה לא נגעו בכסף, אלא רק במה שקשור אליהם וזהו. מהר מאוד הגיעו למסקנה שדבר כזה מאוד בעייתי, כי בסופו של דבר, כמו כל עסק כלכלי אחר שדורש כסף, אנחנו צריכים גם לדעת כיצד להתנהל איתו, ואנחנו צריכים לשים לב גם לסיכונים. אחת הבעיות הקשות בפיתוח תכנה חדשה, זה מי שאין לו השכלה כלכלית נתקל מהר מאוד בבעיות ולא מצליח להגיע למוצר סופי, על אף היכולות המרשימות שיש לו.

לכן עלינו תמיד לחשב סיכונים. בכל דבר אחר שאנחנו עושים בחיים - חתונה, משכנתא וכו' אנחנו מתעסקים הרבה ומתייעצים. ואם על סכומים כאלה של מיליון אנחנו חושבים הרבה, אז על עסקים של מיליוני ועשרות מיליוני שקלים ברור שאנחנו לא יכולים לסמוך על "יהיה בסדר".

נגדיר כעת – "סיכון: הוא האפשרות של הפסד" – כל הפסד (כרגע אנחנו מתעסקים יותר בצד הכלכלי), שעלול לקרות לנו, יש דברים שאנחנו יכולים לחשוב על אפשרות שיקרו, ויש דברים שקורים ללא התראה מראש, אך אנחנו בכל חייבים להיות ערוכים עד כמה שאפשר. כל סיכון כרוך באי-ודאות לגבי האם הוא יתרחש, ואם ומתי הוא יתרחש, גם אז צריך להבין מה אותו הפסד גורר.

מחשבה על סיכונים שונים שעלולים לקרות, בעצם יגררו אחריהם מספר נזקים, אותם אנחנו צריכים לבדוק – <u>תוספת זמן</u> – כל נזק שייגרם ונצטרך לתקן אותו, יוסיף לנו לזמן הפיתוח, וזמן כידוע – שווה כסף. כל עיכוב וחזרה לאחור, גורמים לנו נזק רטרואקטיבי (על כל העבודה שנעשתה לשווא), וגם קדימה עבור זמן התיקון והעיכוב במסירת המוצר. <u>איכות המוצר ירודה</u> – נזק יכול לגרום למסירה של מוצר ירוד, עם הרבה באגים ובעיות שונות, גם אם נרצה לתקן, העיכוב עלול לגרום לפזיזות במסירת המוצר, ששוב לא יהיה מתוקן עד הסוף. <u>כישלון הפרוייקט</u> – הדבר הגרוע ביותר שעלול לקרות. אם הנזק יהיה גדול מידי, משקיעים עלולים למשוך את ידיהם מהפרוייקט, או פשוט לא לממן הלאה, ולבי כסף אין הרבה מה לעשות.

הרציונל שלנו מבקש למפות כמה שיותר סיכונים באלה שעלולים לקרות, על מנת להקדים תרופה למכה, עבור כל בעיה וסיכון שעלולים להתעורר, צריכים להיות מוכנים עם מניעה ותוכנית כיצד לצאת מהבור אליו עלולים להיקלע.

גישת ניהול סיכונים:

בארי בם, שכתב את ה"קוקומו" להערכת מאמץ בפיתוח תכנה, הכניס גם את ניהול הסיכונים למימד החישוב, והמודל הספירלי שנלמד בהמשך מתבסס על ניהול הסיכונים. הגישה לניהול סיכונים היא

[8] מצגת מס' 6

איטרטיבית (ממוענת חזרות) ומונה חמישה שלבים עיקריים. נראה אותם כאחד, ואז נגדיר כל אחד מהם בנפרד:

1. זיהוי סיכונים – מה נחשב "סיכון".
2. הערכת הסיכונים – כמה נזק זה יעשה ומה ההסתברות להופעתה.
3. תעדוף הסיכונים – יצירת רשימה מסודרת וממויינת לפי רמת הסיכון והנזק.
4. מניעה – לאחר התעדוף, החלטה של במה לטפל ואיך עוד לפני שנגיע לשלב הנזק.
5. בקרה - מעקב אחרי התכנה – תכנית אופרטיבית לבדיקת התכנה בזמנים קצובים, על מנת לבדוק שלא סטינו מהדרוש, ולתפוס נזקים עוד כשהם קטנים.

זיהוי הסיכונים

כמובן שאנחנו מדברים על סיכונים של מדעי המחשב, ונמצא הרבה כאלה ולא באמת יהיה לנו שפוט לכתוב את הכל ולהתייחס לכל הסיכונים בצורה שווה. על מנת שיהיה לנו קל לכתוב את כל הסיכונים, נמיין אותם למספר משפחות עיקריות, וכל משפחה מאגדת בתוכה הרבה סיכונים מסוגים שונים, וכך נוכל להתייחס לכל קבוצה בפני עצמה בצורה מלאה-

מתחילים בגודל המוצר PS – Product Size – סיכונים אלו מתחשבים בגודל התכנה אותה אנחנו הולכים לכתוב – אין דין תכנה של 20 שורות קוד, לתכנה של 200, אלף או מיליון. ככל שיש יותר שורות קוד, יש יותר סיכוי לטעויות. עד כמה הקוד משתמש בעצמו שימוש חוזר- יש הבדל בין תכנה שחוזרת על אותם שורות הרבה פעמים, לבין תכנה שרצה בצורה ליניארית ללא חזרות. האם מדובר על תכנה רבת משתמשים? – ככל שיש יותר משתמשים, ככה הנזק יורגש מהר יותר, ונוכל לתקן אותו במהירות ובקבלת פידבק מיידי מאנשים שמשתמשים בתכנה. עד כמה הדרישות עלולות להשתנות? אם אנחנו בונים דבר חדש, אנחנו עלולים להביא אותו ללקוח, ורק אז הוא יבין באמת מה הוא רוצה. הדרישות עלולות להשתנות לאחר מחשבה נוספת.

השפעות עסקיות - BU – Business – לנו כמתכנתים, אין ראייה וקשר לעסקים, לפרסום ושאר מרעין בישין. סיכונים כלכליים ועסקיים של המוצר על הכנסות החברה, תכנה שתממן את עצמה תועיל לחברה, תכנה שייקח הרבה זמן עד שתכניס כסף, צריכים לקחת זאת בחשבון ולא לנוח על זרי הדפנה. האם התאריך הוא הגיוני למסירה – עבודה בלחץ כאשר מגלים שיש עוד שבוע לדד-ליין לא תוביל לתכנה מהודקת אלא לבלאגן. איבוד מקורות מימון – לפעמים לאחר סבב הגיוסים, משקיעים שהיו בעבר קשורים בחברה, עלולים לאבד את מקורות המימון שלהם, ובסופו של דבר אנחנו עלולים למצוא את עצמנו ללא מקורות מימון. כמות הלקוחות – אם עובדים עבור פיתוח מוצר לחברה אחת או לשוק רחב יותר, זה משפיע על דרך ההתנהלות של החברה ואילוצי השיווק וההשקעה בו. אילוצי ממשלה – פיתחת מוצר נפלא, אבל פתאום גילית שהממשלה לא ממש מתלהבת מכל הסיפור של לחזות פשעים ולעצור אנשים עוד לפני שהם ביצעו את הפשע, איך תגיב עכשיו?

מאפייני המשתמשים - CU – Costumer characteristics – סיכונים הקשורים ליכולת הלקוח לעבוד עם המוצר, והיכולת שלנו כחברה, לגשת אל הלקוחות. אם אנחנו עובדים על מוצר עבור לקוחות מבוגרים, עלינו לוודא שהמוצר בנוי בצורה שהם יוכלו להשתמש בזה על כל המגבלות הטכנולוגיות של פער גיל, יכולות שימוש וכו'. כל לקוח שונה צריך לדעת כיצד לגשת אליו, וכיצד הוא יוכל ליצור קשר עם החברה- האם מדובר בקשר ישיר, מעבר דרך הספק של המוצר, כל החלטה כזאת יכולה להיות הרת גורל, אנחנו צריכים להקים מערך של אנשים שישרתו את הלקוח לכל הפניות שלו, אם נבנה מוצר מעולה, אבל קשה מאוד להשתמש בו, אנשים פשוט לא ישתמשו בו.
בנוסף, אנחנו שוב מדברים פה על לקוח שהזמין עבודה – האם הוא יודע באמת מה דרוש? האם הוא יכול ליצור איתנו קשר בצורה ברורה, ולהקדיש זמן לפגישות? שיטת "שגר ושכח" יכולה להישמע מפתה, עד

שלאחר תקופה, מזמין העבודה בא לביקורת ומגלה שעובדים על משהו אחר ממה שהוא התכוון. האם הלקוח מבין את שיטת פיתוח התכנה – מה ניתן לעשות מה לא ניתן לעשות.

מאפייני משתמשים. סיכונים הקשורים לתחכום הלקוחות והיכולות של מפתחים לתקשר עם הלקוח במועד.

<u>תהליך פיתוח</u>- PR – Process definition – הבנה של השיטה שעובדים איתה, התהליך עליו עובדים, ועד כמה הצוות העובד מכיר את כל התהליך. אם לא ברורה הדרך, אז יש סיכוי שהתהליך ייתקע או לא ילך לכיוונים הרצויים.

<u>סביבת פיתוח</u> - DE – Development environment – עד כמה המפתחים מיומנים בכלים של סיבת הפיתוח – האם מדובר על כלים חדשים שלא התנסו בהם בעבר, או על פיתוח על בסיס דברים מוכרים וידועים? האם יש את כלי התכנות המתאימות לפיתוח המסוים הזה, או שעובדים עם דברים ש"אמורים" להוציא את אותו דבר, אבל לא בצורה מספיק מובטחת. האם כל המפתחים יושבים באותו מקום – ככל שיש ריכוז של הפיתוח תחת אותו גג, ככה יותר קל ליצור קשר אחד עם השני, להיגב לכל אחד ולהנחות את מי שצריך.

<u>טכנולוגיה מפותחת</u> - TE – Technology to be built – הבנת הטכנולוגיה החדשה - האם היא מוכרת לחברה, יכול להיות שעובדים על בסיס טכנולוגיה שמוכרת בעולם, אבל אם זה פעם ראשונה שהחברה מתמודדת עם פיתוח בטכנולוגיה הזאת, יש הרבה זמן של למידה והסתגלות. האם מדובר על הטמעה של דברים ישנים או כתיבה חדשה מאפס – הטמעה של משהו קיים זה מאוד נוח, אבל צריך לוודא שבאמת עושים דברים נכון, ולא רק לוקחים משהו קיים ומנסים בכח לדחוף אותו לכיוון שאנחנו רוצים גם כשהוא לא מסתדר.

<u>גודל צוות ונסיונו</u> - ST – Staff size and experience –קשה מאוד לקחת קבוצה של אנשים שונים ולהגיד להם לעבוד ביחד, ללא שום רקע מקדים. גם אם כל אחד הוא המוביל בתחומו, לכל אחד יש את הדרך שהוא רגיל לעבוד בה, ולכל אחד יש מיומנויות שונות והתייחסות שונה לעבודה בצוות. עלינו לוודא שהצוות יודע ומסוגל לעבוד ביחד, כמה צוות אנחנו מחזיקים, והאם הם באמת מחוייבים לפרוייקט, ולא שם רק להעביר את הזמן. זה אמנם נראה זוטות, אבל אלו דברים שמשפיעים מאוד על איכות העבודה והסיכוי לנזק.

הערכת (אומדן) סיכונים

איך מעריכים את הסיכונים? ישנם ארבעה מרכיבים עיקריים שאנחנו מגדירים ב"סיכון" לפיתוח תבנה:

1. <u>סיכון ביצועים</u>- האם ביצועי המערכת יפגעו כתוצאה מהסיכון? המערכת תעבוד אבל לא באופן אופטימלי – תבנה שדורשת הרבה מעבד ומשאבי מערכת, עלולה לא לרוץ היטב אצל הלקוחות.
2. <u>סיכון עלות</u> – טעויות בחישוב עלות המוצר והפיתוח. אם לאחר זמן מגלים שחסר לנו כסף לפיתוח, או שהחישוב הראשוני בכלל לא היה נכון, והעלות היא הרבה יותר גדולה ממה שנדרש מלכתחילה.
3. <u>סיכון סיוע</u> – סיכון שמתייחס לתבנה עצמה אצל הלקוח – האם נגלה פתאום איזו בעיה שלא נוכל לטפל בה? או שנוכל לתקן אותה אבל בצורה לא נוחה, שעלולה לגרור עבודה נוספת ותיקונים חוזרים.
4. <u>סיכון לוחות זמנים</u> – אין הרבה מה להרחיב פה, כ סיכון גורר אחריו זמן עבודה מחודש. בסופו של דבר הזמן הזה עלול להוביל לנו לסיכון חדש.

השפעות סיכון

מיון הסיכונים השונים מחולק בדרך כלל לארבעה רמות סיכון. ככל שהסיכון יותר גבוה, הוא גורר אחריו גם זמן תיקון גבוה יותר, גם עלות תיקון גבוהה יותר. רמות הסיכון מחולקות ל"קודים", שעבור כל קוד יש השפעה שונה:

1. קטסטרופי Catastrophic – כישלון המשימה ו/או תוספת עלות של K500 $. למעשה, הרמה הקשה ביותר- ישנם עלויות שאנחנו לא יכולים לספוג, ואם נגלה סיכון שעלול להשבית את כל העבודה, עלינו לסווג אותו כ"אסון".
2. קריטי Critical – קרוב לכישלון ו/או תוספת עלות של 100K-500K $ - דבר כזה זה נזק עצום, אבל אפשר לצאת ממנו בכוחות מרובים, ובכל זאת נעדיף להמנע ממצב כזה.
3. שולי/גבולי Marginal – ירידה בביצועים ו/או תוספת עלות של 1K-100K $ - אמנם זה יחסית שולי למיליונים שמושקעים במוצר, אבל לאחר כמה התקלויות "שוליות" של המוצר, אולי יש פה משהו קצת עמוק יותר.
4. זניח Negligible – אי נוחות בשימוש במערכת ו/או תוספת עלות עד 1000 $ - מכה קלה בכנף של הפיתוח, אבל עדיין כדאי להתייחס גם לסיכונים כאלה.

כל בעיה נמיין לפי המשפחה וסוג הסיכון שעלול לקרות, כאשר כל סוג סיכון צריך גם להיות ממותג ברמות הסיכון המתאימות לו.

טבלת סיכונים

לאחר שאנחנו אוספים את כל המידע שדיברנו עליו עד עכשיו, אנחנו מתחילים למלא את טבלת הסיכונים, המחולקת לחמישה עמודות:

Risks	Category	Probabilty	Impact	RMMM
מה הסיכון? לא עושים את זה לפי סדר מסוים, אלא פשוט דוחפים כל מה שעולה לראש	לאיזה משפחת סיכונים הוא משתייך – נכתב בראשי התיבות המתאימות	ההסתברות שהסיכון יקרה – באגים בקוד באים בסבירות גבוהה יותר לדינוזאור שיהרוס את כל המחשבים במשרד	רמת ההשפעה – מדד השפעות הסיכון – נכתב רק על ידי מספר הקוד של רמת ההשפעה (פה הדינוזאור דווקא יהיה יותר משמעותי)	נעבור על זה בהמשך

נצרף למטה איך אמורה להיראות טבלת הסיכונים בשלב הזה. יש לשים לב, כלום לא ממוין עדיין, ודברים מופיעים בערבוביה, אבל זה השלב הראשוני של הטבלה:

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

לאחר שיש לנו את כל הסיכונים הרלוונטיים, אנחנו מתחילים במיון. מתחילים קודם כל מיון על פי הסתברות. ולאחר מכן, ממיינים על פי רמות ההשפעה. מה שזה ייתן לנו, זה שחשוב לנו לטפל בדברים שסבירות גבוהה שיקרו, גם אם רמת ההשפעה קצת קטנה יותר, מאחר שעבודה על דברים קטנים שחוזרים על עצמם כל הזמן, יסיחו אותנו מהעבודה החשובה שאותה אנחנו אמורים לעשות.

לאחר המיון – מנהל הפרוייקט מותח קו – לא נטפל **בכל** הסיכונים הקיימים, כי צריך להישאר ריאליים, אבל כן יהיה איזה שלב שנגדיר עליו מה ראוי לעבודה ומה על אף רמת הסיכון הגבוהה, הוא בעל הסתברות נמוכה מכדי להתעסק איתו (דינוזאור ואפוקליפסת זומבים כנראה יישארו בחוץ).

ניסוח תוכנית פעולה

אחרי שממיינים את כל הסיכונים ומחליטים מה רלוונטי ובמה נטפל, כותבים את ה-RMMM - Risk Mitigation Monitoring and Management. שזו תוכנית העבודה לנו להתנהלות אל מול הסיכונים הקיימים בעבודה. כל M מתעסק באספקט שונה אליו אנחנו מתייחסים-

צמצום מסוכנות Risk Mitigation – דברים רעים קורים, כי הלילה אפל ומלא זוועות, אבל אם נהייה ערוכים, נוכל לצמצם את הסיכון. ביצד? קודם כל צמצום המסוכנות עצמה – לוודא שאם תהיה פגיעה, היא תהיה פחות משמעותית וחזקה ממה שהיא יכולה להיות בכוחה המלא. צמצום השביחות – הקטנת האפשרות שבכלל תהיה פגיעה, ללכת ולעבוד עם ראש בקיר, זה נחמד, אבל זה כנראה לא יקטין לנו את הסיכון.
שימוש באמצעים משפטיים, ארגוניים וכו' – על מנת לוודא שאנחנו לא עוברים על חוקים שייפגעו בנו, או מתנהלים בצורה חסרת אחריות.

בקרת הסיכון Risk Monitoring – מעקב אחרי התוכנית ועדכונה. תפיסה של סיכון בשלבים מוקדמים יועיל להקטנת הסיכון עצמו ולרמת הפגיעה בחברה.

ניהול סיכונים Risk Management – מה עושים כשהסיכון באמת קורה? איך אנחנו מתמודדים אותו, וכמה מהר אנחנו מצליחים להימנע משקיעה בבוץ, ולהתרומם כמה שיותר מהר.

ה-RMMM הוא לא רק המלצה יפה, אלא ממש חוברת שכל חברה צריכה להגיש (או לפחות לדאוג לחבר), בה יהיו מפורטים כל הסיכונים השונים, ותבוניות עבודה מתאימות.

נראה דוגמה לתוכן עניינים של חוברת RMMM שכזו –

I. Introduction
  1. Scope and Purpose of Document
  2. Overview of major risks
  3. Responsibilities
    a. Management
    b. Technical staff
II. Project Risk Table
  1. Description of all risks above cut-off
  2. Factors influencing probability and impact
III. Risk Mitigation, Monitoring, Management
  n. Risk # n
    a. Mitigation
      i. General strategy
      ii. Specific steps to mitigate the risk
    b. Monitoring
      i. Factors to be monitored
      ii. Monitoring approach
    c. Management
      i. Contingency plan
      ii. Special considerations
IV. RMMM Plan Iteration Schedule
V. Summary

ניתן לראות ,שאם מדובר על דברים באוויר- אלא אם אנחנו מדברים על צמצום פגיעה, אנחנו ממש מסתכלים על אסטרטגיה מתאימה, וצעדים קונקרטיים לצמצום של כל סיכון.

החלק הרביעי של הRMMM מתעסק בלוחות זמנים – כל כמה זמן אנחנו מבקרים את עצמו, איך אנחנו עוקבים אחרי הסיכונים, על מה מסתכלים יותר ועל מה פחות.

החלק החמישי – הסיכום – חייב להופיע והוא מתייחס לסקירה חוזרת של כל דרבי הפעולה, ומה עושים בכל שלב.

המודל הספירלי

אמנם את כל המודלים עברנו בחלק הקודם וזה נראה קצת לא שייך, אולם מודל זה מתבסס על ניהול הסיכונים, ומה שנגזר מהם, ולכן נדחה דווקא לעכשיו.

מודל זה שהוצע על ידי בוהם ב-1988, מתבסס על מודל מפל המים, אך מתחשב בניהול הסיכונים שלמדנו זה עתה. הדבר המיוחד שהמודל הזה הוסיף, הוא האיטרציה. לקחנו את המודל הכללי של מפל המים, ואנחנו מיישמים אותו ב"נאגלות" – אנחנו לא עובדים מההתחלה עד הסוף, אלא עוברים שלב, מיישמים מה שניתן על פי ההגדרה, וממשים חלק מהדרישות של מפל המים, כך שבסוף כל האיטרציות נקבל יישום מלא.

בגלל הסבבים ההולכים וחוזרים, מודל זה נראה כמו ספירלה ומכאן שמו.

הסבר המודל

האיטרציות השונות לוקחות את סט-הדרישות ומפעיל עליו את ניהול הסיכונים מההתחלה ועד הסוף, ורק אז מתחילים לדבר על מפל המים

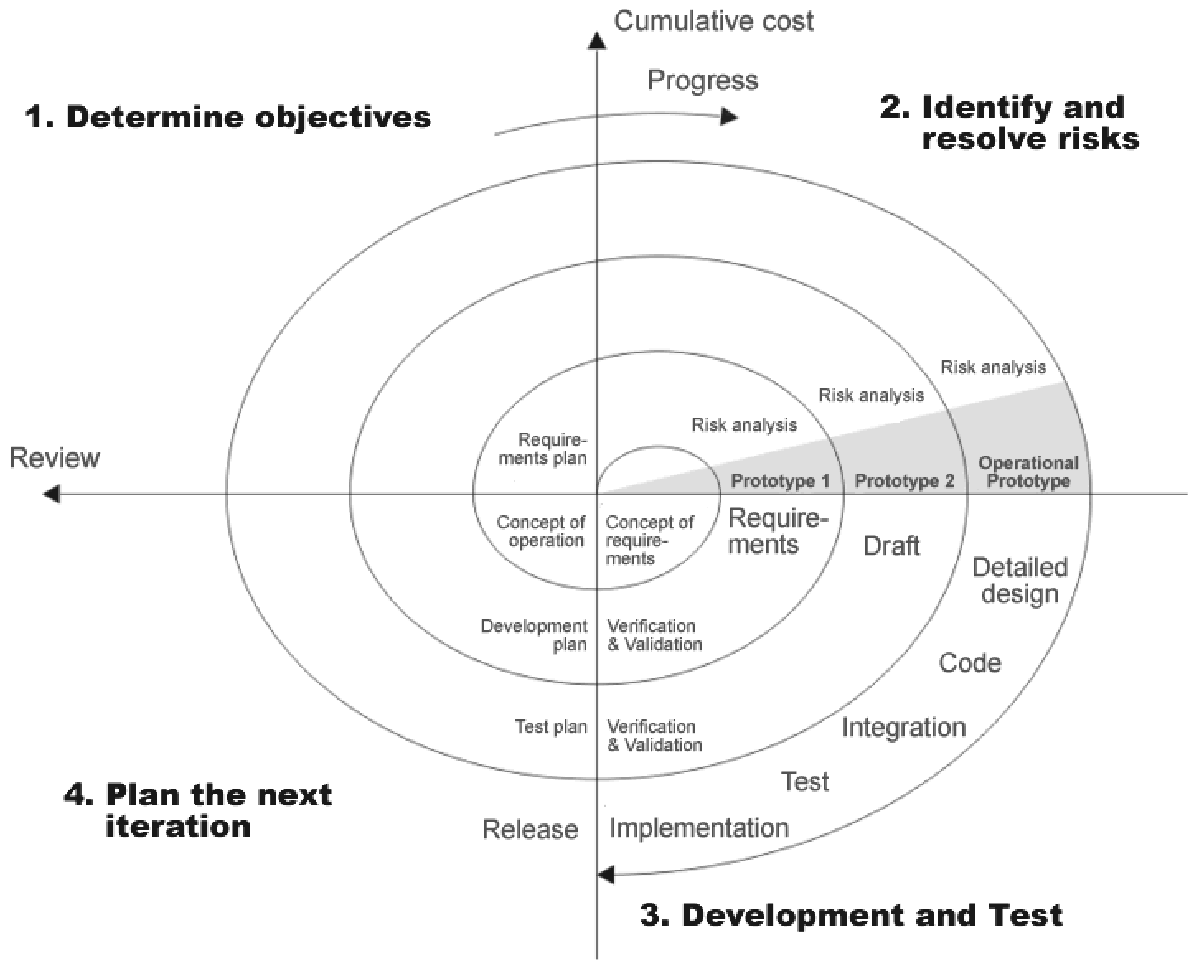
בכל סבב של הספירלה, עושים את הדברים הבאים –

1. בוחרים בחלק מהדרישות – שוב, לא כל הדרישות – חלק.
2. מורידים את הסיכונים הטמונים בדרישות שבחרנו – על פי ניהול הסיכונים מנסים להפעיל את הנוהל של RMMM על מה ששייך לדרישות אלו בלבד.

3. בונים אב-טיפוס – על פי מה שלמדנו בפרק הקודם. אב-הטיפוס מתייחס לכל מה שעשינו עד עכשיו, וכן לשלב הנוכחי בו אנו נמצאים
4. לפי הערות הלקוח, מתקנים את הדרוש – כמה זמן זה ייקח תלוי במודל אב הטיפוס בו אנחנו משתמשים.
5. מפל מים: ניתוח, עיצוב וכתיבת קוד – את כל הדרישות כבר אספנו קודם, ובעת נותר לנו השלבים האלו.

ברגע שמסיימים את כל השלבים האלה, לוקחים רשימת דרישות חדשה, ומתחילים שוב. בסופו של דבר, המוצר גדל לאיטו עד שאנחנו מגיעים לתוצאה הרצויה.

תרשים המודל:



מודל הספירלה מחלק את העבודה לארבעה רביעים, כאשר בכל מעבר על הספירלה, אנחנו עוברים בכל אחד מארבעת החלקים:

1. תכנון / קביעת מטרות – הגדרת המטרות, החלופות והאילוצים.
2. ניתוח וניהול סיכונים – בכל רמה אנחנו עוברים על הסיכונים הרלונטיים.
3. פיתוח הנדסי ובדיקות – פיתוח המערכת ובדיקתה.
4. הערכת המשתמשים ותכנון האיטרציה הבאה– הצגת התוכנה למשתמשים, וקבלת פידבק.

כל המודל מוכון הורדת סיכונים – בין אם באיטרציות ההולכות וחוזרות על ניהול הסיכונים, ובין אם בפניות החוזרות ללקוח, על מנת לוודא שאנחנו עובדים בכיוון הנכון.

ציר ה-X נקרא Go\No-Go. למעשה, בכל פעם שאנחנו מגיעים לשלב שאנחנו אמורים לחצות את הציר, יש לנו אפשרות לבחון את התכנה ואת המוצע לנו, על מנת להחליט האם אנחנו ממשיכים בפיתוח, או נאלצים למשוך את ידינו ולהכריז שמיצינו, ואין סיבה להמשיך. ניתן לראות כי יש שני מקרים בהם אנחנו מגיעים לציר ה-X. לאחר הניהול סיכונים – אנחנו עלולים להבין שיש לנו יותר מידי סיכונים, ועדיף שנעצור לפני שיהיה מאוחר מידי. בפעם השניה – לאחר שאנחנו מראים ללקוח, הוא יוכל פשוט למשוך ידיים, ולהגיד שהוא לא רוצה להמשיך את העבודה. מה עושים במקרה כזה, זה עניין של חוזים וכאלה, אבל בטוח לא ממשיכים לעבוד בלי שהלקוח מעוניין בזה.

ציר ה-Y מבטא את העלות המצטברת – ככל שאנחנו עוברים עליו יותר פעמים, כך העלות הכללית של העבודה תגדל. אנחנו שואפים לחצות את הקו הזה כמה שפחות פעמים.

ההבדל הגדול בין מפל המים למודל הספירלי, הוא שבמפל המים אנחנו חושבים עד הסוף, מרביצים עבודה, ורק בסוף מראים ללקוח. בספירלה אנחנו עובדים על איטרציות הרבה יותר נוחות ומעודנות, הלקוח מעורב בכל השלבים, והאב טיפוס הוא חלק אינטגרלי ממערך המודל.

יתרונות המודל

הרבה מסירות קוד קטנות ליזם, גורר שאנחנו לא עובדים לחינם, כל שלב הולך ונבדק אריח על גבי לבנה. המודל מתאים למציאות, ישנה התקדמות זהירה ולא חסרת מחשבה. מעורבות הלקוח גוררת תוכנה עובדת נכון יותר, ומדויקת יותר.

השיטה מתאימה לפרוייקט גדול, שקשה מאוד לעבור עליו בפעם אחת, ולכן אפשר בכל חלק לקחת שלבים חלקיים מהדרישות בפרוייקט.

מתאים גם לפרוייקט שהדרישות לא ברורות בו עד הסוף – אם במודל מפל המים היינו סגורים לשינויים, כאן אנחנו יכולים לקבל לקוח, עוד לפני שהוא חידד לעצמו בדיוק מה הוא רצה, ועם החזרות השונות אנחנו נבין ביחד איתו לאן הוא שואף.

חסרונות המודל

על מנת שהמודל יעבוד טוב אנחנו חייבים מפתחים מנוסים. כל החזרות וניהול הסיכונים זה דבר שקשה מאוד למפתחים שאינם מנוסים לעבוד איתם, ובעיקר כאשר כל הזמן חוזרים לאותם מקומות. לא כל מי שיודע לכתוב קוד, יצליח להחזיק את כל האיטרציות האלו.

ריבוי איטרציות גורר ריבוי שינויים, לכן אנחנו עלולים לחזור מספר פעמים על קוד שכבר כתבנו ולחזור ולשכתב אותו.

קשה לכתוב חוזים – חוזה בדרך כלל הולך באופן ישר לכמות העבודה, אבל מאחר שאנחנו לא יודעים מראש כמה איטרציות נעבור עד הסוף, אנחנו עלולים לדרוש סכומים נמוכים בהרבה מהדרוש, או לחילופין לבקש סכום גבוה במחשבה שנעמוד בו, אך להפסיד כך לקוחות. מאחר והכמות לא ידועה גם קשה לקבוע על בסיס איטרציות וכדו'.

היום כבר פחות מקובל להשתמש במודל זה – בעיקר בגלל ענייני החוזים וחיזוי מספר האיטרציות, או שמנסים לדחוס איטרציות בצורה מסובכת.

UP – Unified Process[9]

החל משנות ה-80 החלו להשתמש בתכנות מונחה עצמים (OOP). הבעיה היא שהיו לא מעט שיטות, וכל אחד לקח רק את החלק המתאים לו ולהיה מזה בלאגן שלם. בשנת 1996 התאגדו שלושה חוקרים וייסדו את ה-UML שזו שיטת מידול המתאימה לעבודה עם OOP. הרעיון היה ליצור מודל עבור כל שלב בנייה, ולעשות הכל בצורה תהליכית, ומכאן השם "תהליך מאוחד" Unifies Process.

עקרונות UP

1. איטרטיבי ואינקרמנטלי – על מנת להתרחק כמה שאפשר ממודל מפל המים, אנחנו עושים בכל פעם מספר איטרציות, כאשר כל איטרציה בונה ומוסיפה על גבי הקודמת. צורת העבודה נקראת workflow על שם הזרימה שלה שבכל פעם מוסיפים, גם אם התוספת לא מורגשת בעין. זה מרגיש קצת כמו שיטת הספירלה, רק שבספירלה אנחנו לא עובדים בצורה של בניה, אלא פשוט לוקחים בכל שלב מספר דרישות ועובדים עליהם, כאן יש משמעות לסדר ולרמות השונות.
2. מונע ע"י Use-Case – אנחנו אוספים את הדרישות מהלקוח ובונים מהם use-case שונים, כאשר כל מקרה שימוש כזה נובע ישירות מהדרישות, ועליו אנחנו מפעילים את ה-UML המתאים.
3. מרוכז לפי ארכיטקטורה – שלבי הבנייה מרוכזים על פי סדר מאוד מסוים שמוסיף רבדים שונים, ולא סתם לפי איך שבא לנו.
4. ממוקד סיכון – בהנחה שאנחנו עובדים בצורה נכונה, אנחנו מפחיתים משמעותית את הסיכון ליפול על טעויות, כל דבר נבדק שוב ושוב ומצמצם את האפשרויות לטעויות.

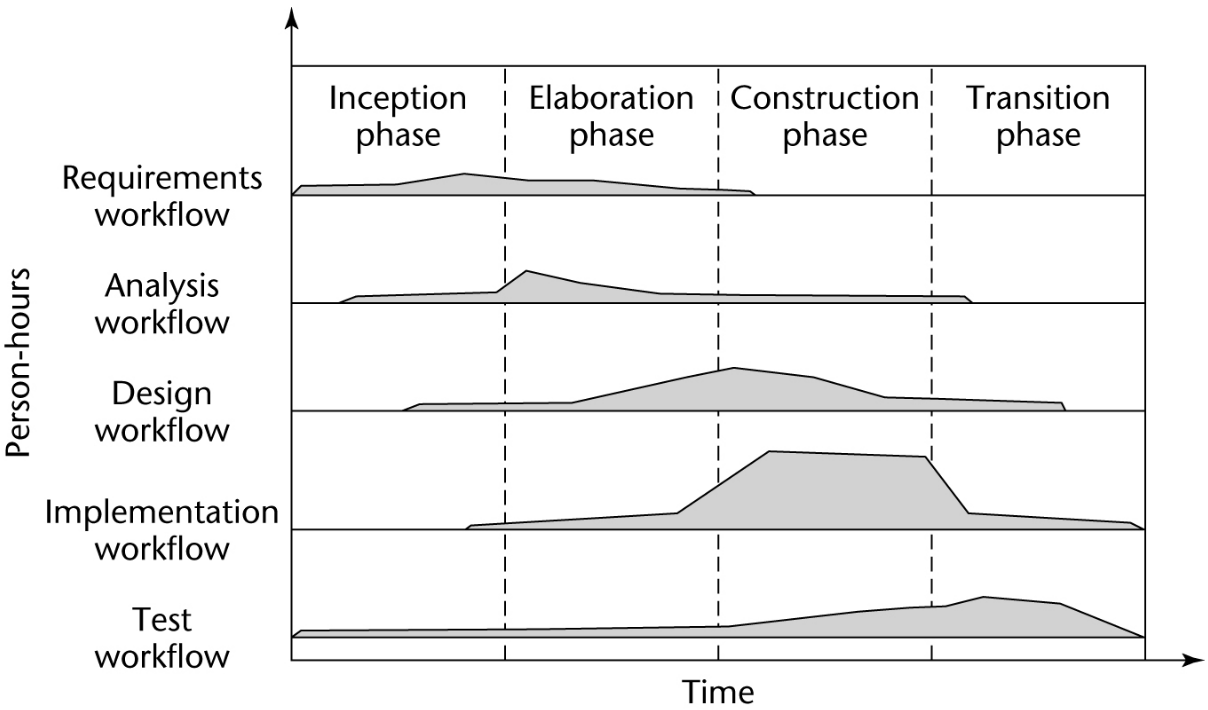
תהליך הפיתוח בעצמו מחולק לארבע פאזות עיקריות – אם במפל המים היו לנו 7 שלבים, כאן יש רק ארבעה, אבל אנחנו דואגים להבליע את כל השלבים הרצויים בזמן התהליך. ארבעת השלבים הם:

1. **שלב ההתחלה Inception Phase** – הצורה הטבעית ביותר לדבר על הפרוייקט, הדברים הסובבים שמאפשרים לנו לפתור את הבעיה. מי שרוצה לצבוע את הבית צריך בכלל לבדוק שיש לו אפשרות להביא את הצבע. השלב הזה נובע בעקר מאיסוף הדרישות.
2. **ביסוס הפרוייקט Elaboration Phase** – הרחבה של הפרוייקט והנחת היסודות של כל מה שאנחנו רוצים עבוד עליו, איחוד של דרישות תחת גג מסוים וכו'.
3. **בנייה Construction Phase** – הבנייה והמימוש עצמו של הפרוייקט.
4. **מעבר Transition Phase** – התקנת הגרסה הראשונה אצל הלקוח והמתנה לפידבק.

הפאזות מתבצעות אחת אחרי השניה, כאשר בכל פאזה אנחנו מכילים את כל שלבי זרימת העבודה השונים. כמובן שאנחנו לא כותבים את כל הקידוד מההתחלה בשלב הראשון, אבל אנחנו משתדלים לתת נגיעות כבר מההתחלה ברמה מסוימת.

ניתן לראות בדיאגרמה בהמשך שבכל שלב מכיל אחוזים מסויימים של ה-workflow השונים. ניתן גם לראות שהשלבים העיקריים ממוקדים באלכסון שנראה דומה לאלכסון של מפל המים, אך בעצם זה שאנחנו עושים את זה בצורה מתמשכת וחוזרת על עצמה, אנחנו דואגים שהעבודה תזרום בכל הרמות.

[9] מצגת 7



שלב ההתחלה Inception Phase

בשלב ההתחלה אנחנו אוספים את כל המידע הדרוש לנו על מנת להתחיל לעבוד, החל משלב הדרישות, וכלה בהבנה של התחום עליו אנחנו הולכים לעבוד – אם אנחנו עושים תכנה שקשורה לתחום עבודה מסוים, אנחנו צריכים להבין בור ברמה המינימלית כדי שתהיה לנו שפה משותפת.

הפעולות העיקריות בשלב זה- (גם פה יש רשימה, עוד מעט זה יסתיים)-

1. הגדרת חזון המוצר – מה המקום של המוצר בשוק, למי הוא ממוען, כמה אנשים הולכים להשתמש בו. האם מדובר במוצר שמוכרים לשוק הרחב, או למוצר עבור מקום מסוים. כל שאלה כזאת מגדירה לנו את המוצר בצורה יותר נבונה.
2. חשיפה ראשונית של דרישות הלקוח – אנחנו לא לוקחים על ההתחלה את כל הדרישות, אלא בונים משהו מאוד ראשוני, ואחר כך רואים איך זה מתיישב עם שאר הדרישות של הלקוח.
3. הגדרת תחומי הפרוייקט – מה התכנה תעשה לטובת המשתמש. בונים UC של הדרישות השונות ומנסים לבנות איזה תרחיש בשביל לראות איך כל דבר זז ממקום למקום.
4. מילון מונחים – הרחבה של המושגים עליהם אנחנו מדברים. אם אנחנו מדברים על אמנות, אנחנו צריכים שתהיה לנו שפה משותפת עם הלקוח, ולהכיר את עולם הדיון שלו. אנחנו לא נמציא את כל המושגים מחדש, אלא נבסס על מה שנמד ממנו.
5. מודל עסקי – חישוב עלויות מול הרווח הצפוי. הבנה של העלויות הכלליות שעלולות להתווסף לתוכנה ברמת של משכורות וכו'. יכול להיות שלאחר שלב זה נבין שזה בכלל לא רווחי לנו ליצור את כל התכנה הזאת, ונוותר לפני שיהיה מאוחר מידי.
6. ניהול סיכונים ראשוני.
7. יצירת חוזה ראשוני – יצרת חוזה מינימילי בינינו לבין הלקוח, החוזה אמור להיות פתוח לשינויים בהמשך, אבל לפחות לכתוב "ראשי פרקים".

Use-Case

תרחיש או רישום של מתווה שימוש בפרוייקט. בדומה לפונקציה מורכבת שהיתה לנו ב-DFD, הUC מסביר לנו איזה תהליך או פעולה שצריכים להתבצע מההתחלה ועד הסוף. למשל אם אנחנו עשינו תהליך של "קבלת עובד" או משהו בסגנון אנחנו מתחילים ביצירת העובד והתחנות השונות אותן הוא אמור לעבור וכו'.

על מנת להבין את כל התהליכים, נשתמש בדוגמה של אוסברט אוגלבי. זה מעין מקרה לימודי (Study-case) שמתאר את כל שלבי הבניה מההתחלה ועד הסוף.

מקרה-בוחן אוסברט אוגלבי

מביאים בפנינו מקרה בוחן שיעזור לנו להבין את כל השלבים הדרושים לבניית דיאגרמות UML נכונות.

אוסברט אוגלבי הוא סוחר אומנות. הוא מעוניין ביצירת תכנה שתותאם עבורו, ותעזור לו בניהול השוטף של ענייני המסחר. בשלב הראשון עלינו להבין את התחום בו אנחנו דנים. לצורך זה, אנחנו מראיינים את אוסברט אוגלבי, ומנסים לדלות ממנו מידע על מנת שנוכל להכניס אותו למילון המונחים.

מילון המונחים שהוצאנו הוא זה:

Landscape: a painting of a scene in nature
Masterpiece: a painting of undoubted excellence
Masterwork: an inferior painting by an artist who previously or subsequently has painted a masterpiece
Medium: a classification criterion; the material with which an artwork is painted; *see also* oil, watercolor
Oil: a medium; abbreviation for "oil-based paint"
Other painting: a painting that is neither a masterpiece nor a masterwork
Portrait: a painting of one or more people
Quality: a classification criterion; a painting is classified as a masterpiece, masterwork, or other painting, depending on its quality
Still life: a painting of inanimate objects
Subject: a classification criterion; subjects include landscape, portrait, and still life
Watercolor: a medium; abbreviation for "water-based paint"

אפשר לראות שהמונחים פה די מעורבבים מבחינת הסדר, יש לנו בהתחלה ציור נוף, באמצע פורטרט, ובין לבין מיונים של סוג העבודה, חומרים שמשתמשים ביצירה ועוד, אבל אנחנו יכולים כבר לקבל איזה מושג כללי, ולראות מה קשור למה ומה תחום הדיון שלנו.

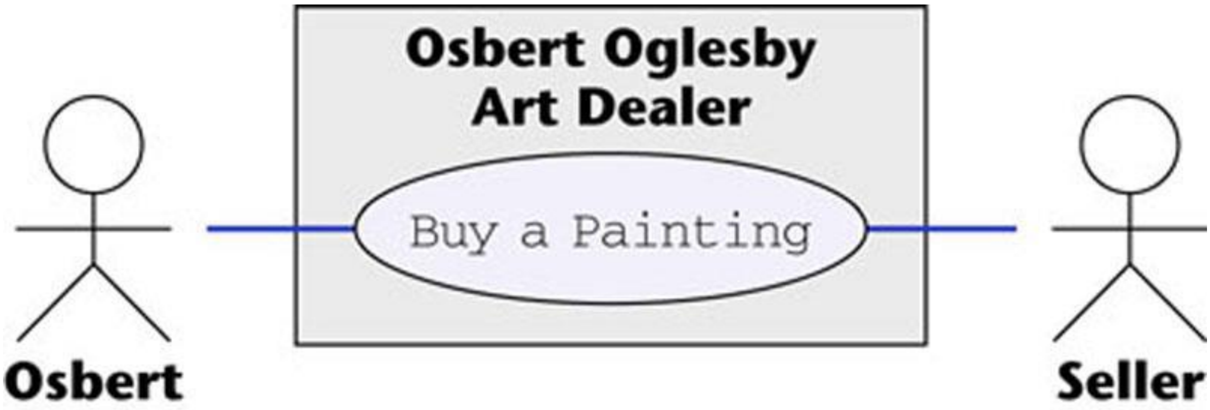
בנוסף, אנחנו מקבלים ממנו רשימות דרישת מהתכנה:

מבחינת קניית ציור – על התכנה להעריך מה המחיר המקסימלי עליו לשלם עבור יצירה נתונה.
זיהוי טרנדים ברכישת יצירות, מוקדם בכל האפשר – אם יש התעוררות של רכישות עבור אמן מסוים או משהו בסגנון, על התכנה לחזות את זה כמה שאפשר על מנת שאוסברט יצליח להיערך בהתאם – לרכוש מה שאפשר ועוד.
בנוסף ,על התכנה לשמור רישום של כל הרכישות והמכירות השונים.

מבחינת פעולות עסקיות, אוסברט עושה 3 פעולות באלה באופן שוטף-

1. קונה יצירות אומנות.
2. מוכר יצירות אמנות.
3. מפיק דוחות מכירוה/קניה.

כל מה שתיארנו עד עכשיו הם הUC הבסיסיים עליהם נעבוד.



UseCase בסיסי שנעבוד עליו מורכב באופן הבא – שחקנים (Actor) שייצגו את היישויות השונות העובדות על המערכת ושהמקרים מתייחסים אליהם – יש לשים לב – אין חשיבות אמיתית למיקום בו מניחים את השחקן, שלא כמו בDFD שעשינו, שם היתה לנו ישות שמכניסה מידע משמאל, ויישות שמוציאה מידע, כאן המקרה שונה – שניהם יכולים להיות בשני הצדדים.

המקרה עצמו אליו אנחנו מתייחסים, יוקף באליפסה ובתוכה כתוב את הפעולה המתבצעת. במקרה זה "רכישת יצירה". הקווים הנמתחים בין השחקנים יבטאו כמובן, את השחקנים הרלוונטיים.

מאחר ויכולים להיות כמה פונקציות שונות תחת אותו רעיון כללי, אנחנו מגדירים את מסגרת-העבודה בריבוע, וכל הפונקציות נמצאות בתוכו.
להמשך הפיתוח של הדיאגרמה, על פי הדרישות, אנחנו נוסיף את שתי הפעולות הבאות:

ניתן לראות שיש פה הבדל בין שחקן שקונה/מוכר. אנחנו לא מדברים פה על יישויות(?) כלליות, אלא על מקרים ספציפיים, ולכן יש לפרט על מה מדברים.

למעשה, כל פונקציה בזאת היא Use Case בפני עצמו, ועכשיו אנחנו צריכים להתחיל להכנס טיפה יותר עמוק. עבור כל מקרה שבזה, אנחנו נכתוב פירוט קצר המסביר את המקרה המדובר –

קניית ציור– מאפשר ללקוח (אוסברט) לרכוש ציור.

מכירת ציור – מאפשר ללקוח למכור ציור.

יצירת דו"חות – מאפשר ללקוח לגשת לרשומות על ציורים שהוא מכר או קנה במשך השנה האחרונה.

את המידע אנחנו מכניסים בטבלה המתאימה, כאשר את התיאור המפורט יותר (שמסביר ממש מה הוא עושה ולאן הוא הולך) אנחנו משאירים לשלב יותר מאוחר.

מתוך כל מה שבירננו עם אוסברט, אנחנו בודקים מה באמת רלוונטי וחשוב ויכול להחשב כ"דרישה" של התכנה. במקרה שלנו, שלושת המקרים הם באמת הדרישות של התכנה, ועכשיו נתחיל לפתח אותם.

שלב הביסוס Elabortion Phase

בשלב זה אנחנו נמשיך ונספק מידע בסיסי עבור כל use case. נוסיף את התיאור הקשר של צעד-אחר-צעד. נבנה את המודל הרעיוני (פה אנחנו קצת נכנסים לרמות של OOP) של המרכיבים השונים והמידע שהתכנה תכיל. ונבנה גם אבות טיפוס ראשוניים לתהליכי העבודה.

ראשית, עבור כל מקרה אנחנו נוסיף את הצעדים הדרושים לקיומו – למשל, עבור רכישת ציור:

Brief Description

The `Buy a Painting` use case enables Osbert Oglesby to buy a painting.

Step-by-Step Description

1. Osbert inputs details of the painting he is considering buying.
2. The software product responds with the maximum purchase price he should offer.
3. If the seller accepts Osbert's offer to buy the painting, Osbert enters further details.

Note: Details of the algorithm for determining the maximum price will be obtained later.

אוסברט מחפש פרטים (נראה המשך באילו פרטים אנחנו מתחשבים) על הציור אותו הוא רוצה לקנות. המערכת עושה את החישובים שלה, ופולטת את המחיר הגבוה ביותר, שעל אוסברט להציע. אם המוכר מקבל את ההצעה של אוסברט, הפעולות ממשיכות הלאה, כשהם סוגרים ביניהם את הקניה.
אנחנו מוסיפים גם בהערה, שחישוב המחיר המקסימלי יתווסף בהמשך.

בהתאמה, נוסיף את הצעדים הדרושים למכירת ציור, ולהוצאת דו"חות.

Brief Description
The `Sell a Painting` use case enables Osbert Oglesby to sell a painting.
Step-by-Step Description
1. Osbert inputs details of the painting he has sold.

Brief Description
The `Produce a Report` use case enables Osbert Oglesby to obtain information on paintings he bought or sold in the past year or to detect new trends in the art market.
Step-by-Step Description
1. Osbert requests a report of the type he needs. The report is printed.

מילאנו את הצעדים (צעד..) של הפונקציות, אבל עדיין לא ממש התקדמנו יותר מידי לקראת הבנה של מה אנחנו עושים. כל מה שרשום הוא יחסית מאוד מעורפל, אבל פרטי האלגוריתמים פחות חשובים לנו, בשלבים הראשוניים, אנחנו לא מתעסקים ב"איך" אלא ב"מה", מסבירי את הדברים שאנחנו עושים וזהו, את הפרטים היותר ספציפיים אנחנו דוחים כמה שאפשר.

מודל רעיוני Conceptual Model

המודל הרעיוני מרכז את כל המושגים החשובים עליהם אנחנו מדברים בתכנה. מודל זה מכיל מעבר ליצירת המחלקות השונות ואת הקשרים שאנחנו מפעילים בין אחד לשני. דבר זה חשוב מאוד, מאחר וברגע שיש לנו את המודל הזה, הלקוח יכול לראות שאנחנו באמת אוחזים במציאות ומבינים על מה הוא מדבר. הוא גם מוסיף לנו בהתאמה, את יסודות ההבנה של העסק, ובהמשך את הבסיס לעיצוב התכנה.

את הקונספט אנחנו ממדלים בצורת מחלקות, והמודל יכיל – כל המחלקות (על פי המושגים השונים) והקשרים ביניהם, בעבור כל מחלקה, נוסיף את רשימת התכונות האופייניות לו.

עכשיו אנחנו עוברים בצורה איטרטיבית, ובכל פעם אנחנו מוסיפים עוד פרטים ומחדדים יותר נקודות.

<u>איטרציה ראשונה</u>

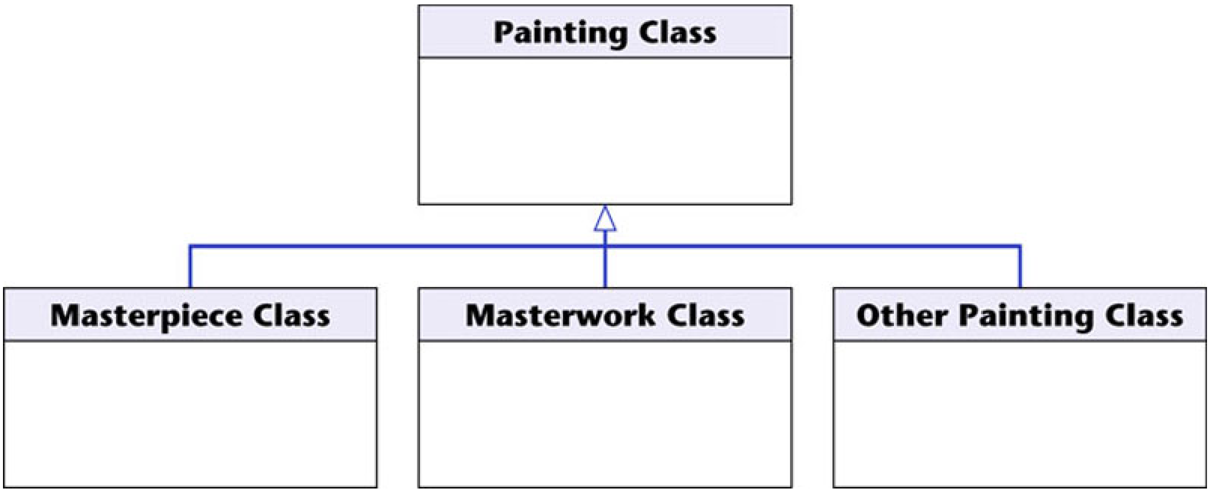
יש לנו ארבע מחלקות עיקריות –

1. Painting Class – ציור.
2. Masterpiece Class – יצירת מופת.
3. Masterwork Class – יצירה של אמן חשוב, אך נמוכה בדרגתה מיצירת-מופת/

4. Other Painting Class – יצירות ללא דירוג בעל חשיבות.

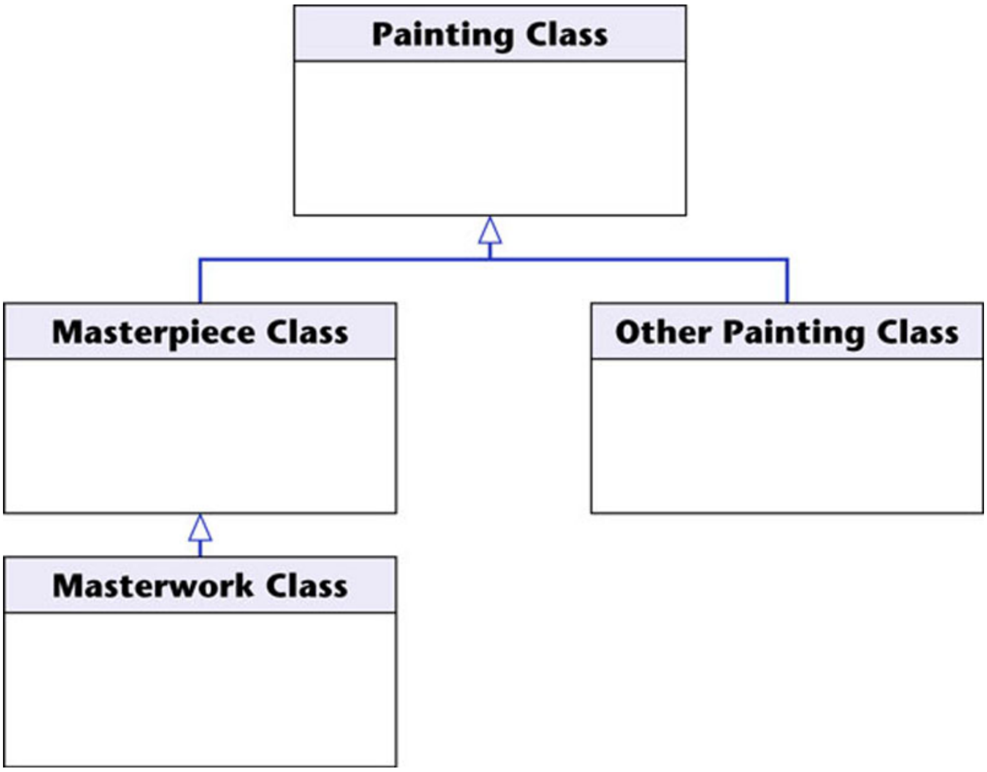
איטרציה שניה

עכשיו אנחנו מסדרים את המחלקות לפי היחסים ביניהם. מי מוכל במי. כמובן, שהמחלקה ציור שהיא כללית יותר תהיה העיקרית, וכולם יירשו ממנה.



איטרציה שלישית

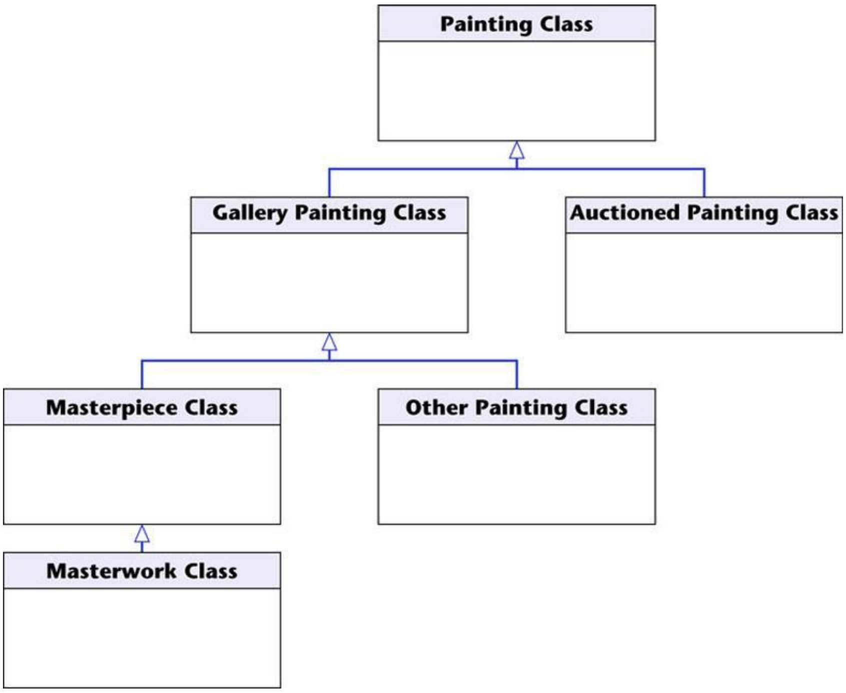
עכשיו אנחנו קבענו באופן כללי את היחס. אבל מאחר ואנחנו יודעים שבהמשך נצטרך להתייחס למחירים שונים, אנחנו רוצים לשקף את זה באופן ברור יותר. למשל – אנחנו מגדירים כי המחיר המקסימלי לMasterwork, יהיה המחיר עבור Masterpiece של אותו יוצר. על מנת שלא נתמחר ציור בצורה שהיא גבוהה יותר ממה ששולם על יצירות אחרות.
בגלל תוספת המידע הזאת, אנחנו מזיזים את המאסטרוורק, שתירש את כל האפשרויות של המאסטרפיס, וכבה אנחנו נוכל לבדוק מה יהיה המחיר המקסימלי לאותו פריט.



<u>איטרציה רביעית</u>

עכשיו אנחנו רוצים קצת יותר להכנס לענייני המסחר של התמונות. מאחר ואנחנו מחפשים למכור תמונות שיש ברשותנו, אנחנו רוצים לחפש רישומים שמחפשים את הציורים שיש לנו, ואותם אנחנו מעוניינים למכור.

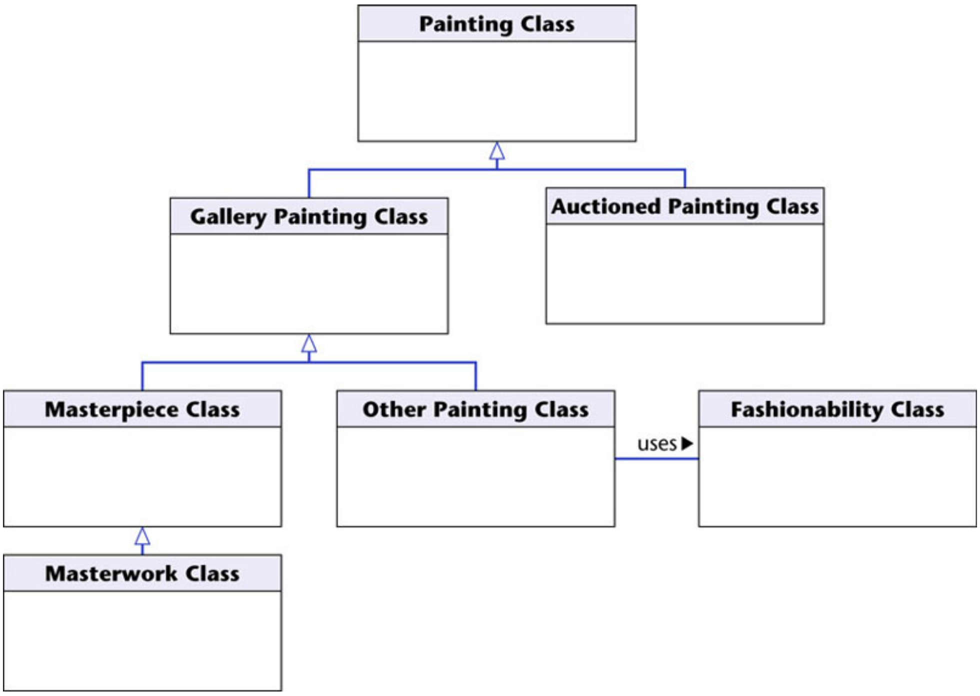
נוסיף למחלקת השנות, עוד מחלקת ביניים, של יצירות שעוברות מכרז. ככה נוכל להשוות בין היצירות הקיימות אצל אוסברט, לבין שאר היצירות. בסופו של דבר, אמנם יש לאוסברט את המלאי המוגבל שלו, אבל כשהוא משווה את המחירים זה בוודאי מול מספר יצירות רחב הרבה יותר, אבל אנחנו לא צריכים להתחיל לקטלג יצירות שאנחנו משווים מולם לפי קטגוריות של חשיבות עבודה וכו', אלא רק ישירות אל מול היצירה עצמה. לכן נפריד מעט את המחלקות ליצירות בין היצירות הקיימות אצל אוסברט בגלריה, שימויינו לפי חשיבות, לבין היצירות החיצוניות שפשוט יהיו "קיימות" כמופעים בודדים.



<u>איטרציה חמישית</u>

דבר נוסף שאנחנו צריכים להתחשב בו (ברמת תמחור היצירה) הוא מדד אופנתיות. יכול להיות שיצירה של אמן מסוים תהיה שווה סכום מסוים, אך בעבור זמן, עקב אירועים שונים פתאום הוא יהיה פופולרי הרבה יותר, ומחירי היצירות שלו יעלו. במקרה זה נצטרך לחשב איזה מדד אופנתיות, שיהוו איזה קבוע שאותו נוכל להכפיל במחיר של היצירה (בדרך כלל הוא יהיה 1, אבל אם פתאום מדד האופנתיות יעלה, אז המחיר יעלה בהתאם).

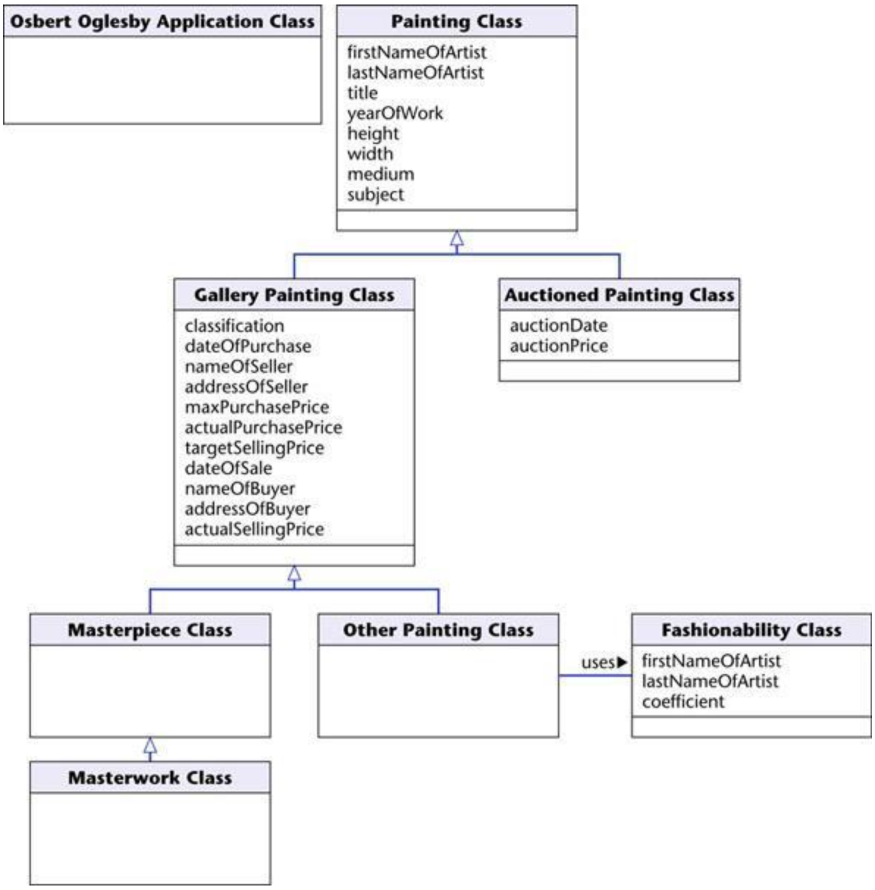
לצורך זה אנחנו נוסיף מחלקה של אופנתיות. עכשיו כאשר אוסברט ירצה להציע מחיר עבור ציור שהוא שייך למחלקה של ציור "אחר" (לא מאסטרקלאס/מאסטרוורק), ניתן יהיה לחשב בעזרת זה את המחיר.



כדאי לשים לב, שהמחלקה של האופנתיות היא מחלקה שמשתמשים בה, ולא קיימת תחת עץ המשפחה של הציורים השונים.

איטרציה שישית

אחרי שהמודל הזה מוצא חן בעינינו, ואנחנו חושבי שכיסינו את האפשרויות הגלומות בו, אנחנו מתחילים להרחיב בתוך כל מחלקה בעצמה. עבור כל מחלקה נוסיף את האטריביוטים הרלוונטים אליה. כמו שם, שנה, אמן וכו' . בנוסף אחרי שיש לנו איזה מודל שאנחנו חושבים שאנחנו יכולים להתחיל לעבוד איתו, נוסיף מחלקה כללית שמפעילה לנו את כל המחלקות לפי הדרוש –



עכשיו אנחנו רוצים להמיר את המחלקות לתצורה יותר מוגדרת – כלומר, אנחנו רוצים לראות את ההבדל בין סוגי המחלקות, לא ברמת השם והירושה, אלא התפקיד הכלל בתוך המערכת. אנחנו ממיינים את המחלקות לשלושה סוגים, ורושמים כל מחלקה עם הסימון המתאים לה –

שלשות הסוגים הם – מחלקת **יישות** – כשמה, דואגת ליישויות השונות בתוך המערכת – במקרה שלנו, מחלקת הציור היא בוודאי מחלקת יישות. מחלקות כאלו מכילות מידע שימושי שאמור להישמר לאורך זמן.

מחלקת **תחום** (גבולות) - מחלקה שמקשרת בין היישויות השונות לתוכנה, המחלקה הזאת בדרך כלל מקושרת לקלט/פלט. למשל מחלקות שקשורות לדוחות שונים של המערכת, וכו.

מחלקת **שליטה** – המחלקה האחראית על החישובים השונים והאלגוריתמיקה. חישוב עלויות מוצרים, מעקב אחרי שינויים בעקבות אירועים שונים, וכו'.

המחלקות יסומנו בדיאגרמה בצורה הבאה-

איך מסדרים את המחלקות בקטגוריות השונות? דרך פשוטה אחת, היא להגדיר את התוכנה בפסקה אחת בודדת. הרעיון מאחורי זה, הוא שברגע שנגדיר את הכל בפסקה אחת, בהכרח נכניס את כל הפעולות

הדרושות וכל ההתמשקויות של המערכת עם עצמה. מתוך הפסקה הזאת אנחנו נוכל לדוג אחר כך את המחלקות השונות לסוגיהן.

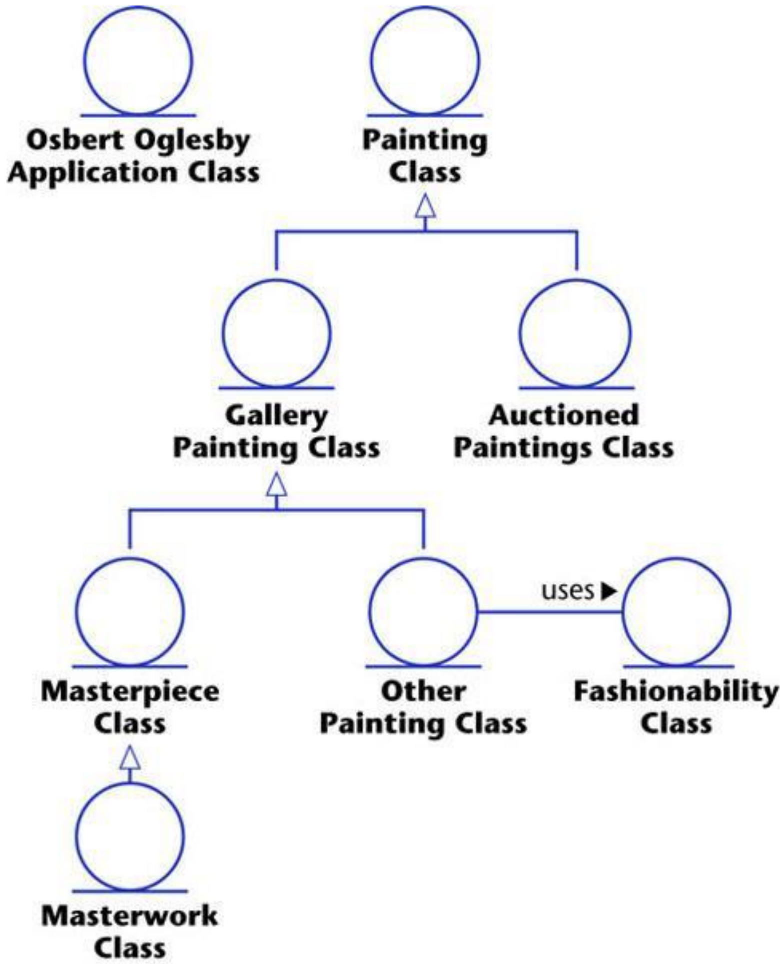
הפסקה שתתאר את התכנה שלנו, תהיה כדלקמן:

התכנה תפיק דו"חות במטרה לשפר את היעילות של תהליך קבלת ההחלטות בעת קניית עבודות אמנות. הדו"ח יכיל מידע על קניה ומכירה של התמונות, המסווגות על פי מאסטר-פיס, מאסטר-וורקס, וציורים אחרים.

המילים עם הקו מתחתם, הם אלו הקשורים לשמות עצם ולפעולות עליהם. עכשיו צריך לנקות את הפעלים עצמם, ולנסות להישאר עם שמות העצם הבסיסיים ביותר. מה שיישאר לנו בסוף, זה רק שמות העצם הקשורים למחלקות הסיווג של הציורים – מחלקת ציור, מאסטר-פיס, מאסטר-וורק, וציורים אחרים.

איטרציה שישית – חלק ב

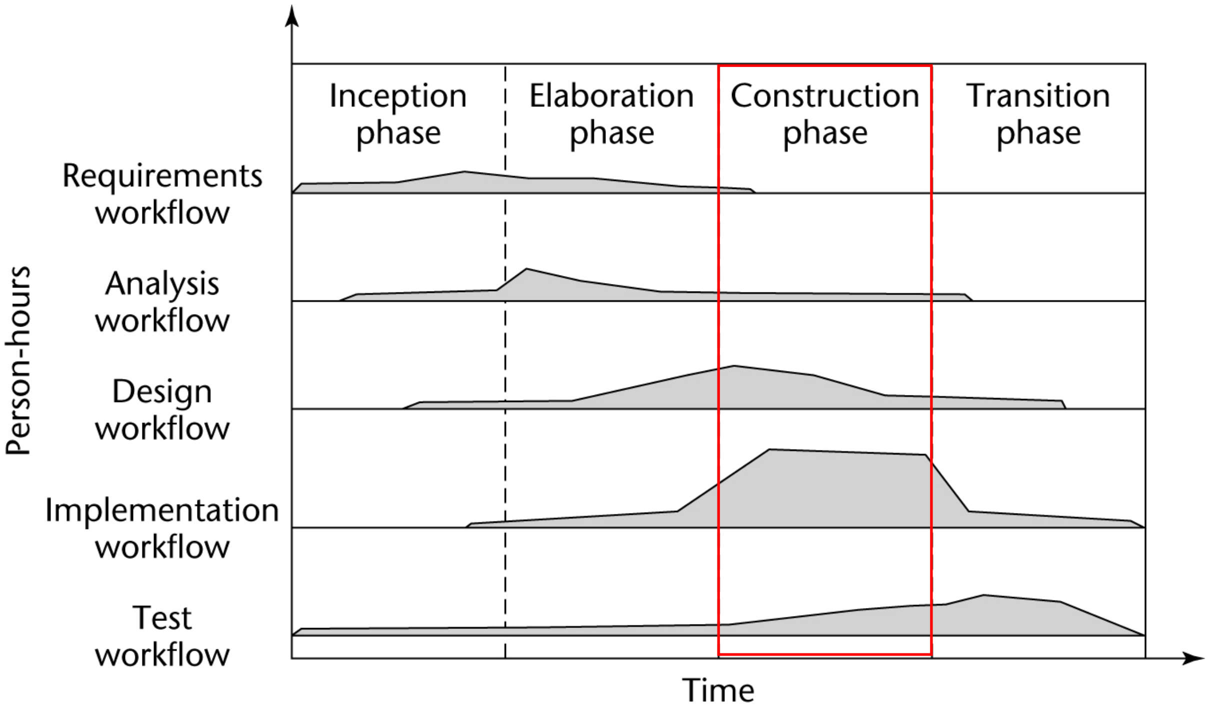
בנייה מחודשת של הדיאגרמה בהתחשבות בסימנים החדשים של מחלקות. למעשה כרגע, כל שיש לנו הם רק היישויות, בשלבים הבאים אנחנו נוסיף את שאר המחלקות הרלוונטיות, והדיאגרמה תקבל גם את שאר הסמלים. ברגע זה ייראה כך –



לסיכום: אספנו את הדרישות, ובנינו מערך ראשוני של התכנה. בכל איטרציה הוספנו נדבך נוסף של הבנה ביצד המחלקות השונות אמורות להיראות ואת הקשרים השונים בין המחלקות. עכשיו יש לנו את הרעיון הכללי של כל התכנה, ואת הבסיס הנדרש על מנת להבין את החלורה שלה. מה שנשאר הוא להתחיל את הבניה, ולראות ביצד התכנה זורמת בעבודה שלב אחרי שלב.

שלב הבניה Construction Phase

אם נחזור רגע לדיאגרמה של השלבים השונים, נוכל לראות כי מבחינת הדרישות (Requierment), והניתוח (Analysis), של התכנה עברנו כבר את ה"פיק", ושלב הבניה אמור להתעסק ברובו בביסוס העיצוב של התכנה, ובבנייה הממשית (Implemnetation) של התכנית. אבל לפני שנגיע לזה, אנחנו צריכים קודם כל לסיים את הדרישות (על מנת שנוכל לעבוד על פי הדרישות האמיתיות), ולהמשיך בניתות.



דרישות וניתוח של שלב הבניה

בשלב זה, אנחנו מרחיבים את כל הuse case שעשינו עד לפה. ראשית, אנחנו מוודאים שבאמת יש לנו את כל הפעולות הנצרכות, ולחדד כל מקרה עד הסוף – אילו מאפיינים ומידע אני אמור לקבל ולמסור ברגע שאנחנו רוצים למכור/לקנות ציור? כל מידע שנחליט שחייב להופיע, אנחנו צריכים להחליט באיזה שלב הוא ייכנס. דיברנו עד עכשיו על אלגוריתם שמחליט לנו על מחיר מקסימלי – עלינו להחליט מהו מחיר מקסימלי, ואיך לחשב אותו. חידוד ההבדלים בין Masterpiece ל-Masterwork – כאשר נגדיר את זה בצורה מדויקת, כל יצירה שתהיה לנו תוכל להיות ממויינת ביתר קלות. אופנתיות – להוסיף רשימה של אמנים שבעבור כל אחד מהם, נוכל להוסיף איזה מדד קבוע F שיהיה האינדקיציה המתאימה לחישובים השונים. עלינו לשקול גם אפשרויות נוספות – למשל, מחירים והצעות שהיו במכרזים בעבר – נגדיר זמן של 25 שנה, ונוכל לחשב לפיו את השינוי במחירים ומכירות, איזה אמנים נמכרו יותר, ובאילו מחירים.

בעת ננסה להרחיב את כל המקרים השונים עד כמה שאפשר. למשל עבור מכירת תמונה, רשמנו את הצעד הבא:

Brief Description The `Sell a Painting` use case enables Osbert Oglesby to sell a painting.
Step-by-Step Description 1. Osbert inputs details of the painting he has sold.

התייחסנו לכך שאוסברט צריך להכניס את הפרטים של הציור אותם הוא רוצה למכור. בשלב זה, אנחנו כבר יודעים באיזה פרטים מדובר, אחרי שעשינו את כל החקירה שלנו, אנחנו יכולים להרחיב את הצעד הבודד הזה, לעוד מספר תתי-צעדים:

Brief Description
The `Sell a Painting` use case enables Osbert Oglesby to sell a painting.

Step-by-Step Description
1. Osbert inputs details of the painting he has sold. These are
   Date of sale
   Name of buyer
   Address of buyer
   Actual selling price

אנחנו נצא מנקודת הנחה, שאנחנו לא צריכים את המייל של המוכר והגיל שלו, אלא רק את הפרטים הרלוונטיים למכירה עצמה. לעומת זאת, קניית ציור תצטרך להכיל הרבה יותר פרטים. בצעדים המקוריים היו לנו 3 צעדים פשוטים – חיפוש היצירה, החזרת המחיר, ובדיקה האם המוכר מרוצה מהמחיר המוצע.

עכשיו אפשר לראות שהמצב כבר שונה לגמרי – קודם כל הוספנו שלב רביעי, אבל כל שלב אחר נוסף לו עוד פרטים ותתי סעיפים, למשל:

השלב הראשון (אדום) – מהם הפרטים שהתכנה צריכה לקבל עבור החישוב עצמו- כל מאפיין של הציור כמו הצייר עצמו, החומרים, הגודל, הקטגוריה של חשיבות הציור, ברור לנו שמדובר על פרטים חשובים, תמונה שהיא בגודל 50*70 ס"מ ותמונה של 3 מטר יחושבו באופן שונה.

השלב השני (כתום) מפרט את החישובים השונים אל פי הקטגוריות – כל קטגוריה מקבלת חישוב שונה, המאסטרפיס יחושב בהשוואה ליצירות מופת אחרות. המאסטרוורק יחושב על פי עבודות של אותו אמן, כל שאר היצירות יחושבו בהצמדה למדדי האופנתיות.

השלב השלישי (תכלת) אם אוסבורט אכן קונה את הציור, הוא מכניס את

Brief Description
The `Buy a Painting` use case enables Osbert Oglesby to buy a painting.

Step-by-Step Description

1. Osbert inputs the details of the painting he is considering buying. These are
   First name of artist
   Last name of artist
   Title of work
   Year of work
   Classification (masterpiece, masterwork, other)
   Height
   Width
   Subject (portrait, still life, landscape, other)
2. The software product responds with the maximum purchase price Osbert should offer.

2.1 For a masterpiece, the software product computes the coefficient of similarity between each painting by that artist for which there is an auction record and the painting under consideration for purchase. Question marks in the first or last name of the artist, or in the title or date of the work are to be ignored.
   The software product scores 1 for a match on medium, otherwise 0.
   The software product scores 1 for a match on subject, otherwise 0.
   It adds these two numbers, multiplies by the area of the smaller of the two paintings, and divides by the area of the larger of the two.
   The resulting number is the coefficient of similarity.
   The software product finds the auctioned painting with the largest nonzero coefficient of similarity. If there is no such painting, Osbert will not buy the painting under consideration.
   The software product computes the maximum purchase price by adding to the auction price of the most similar work 8.5%, compounded annually, for each year since that auction.

2.2 For a masterwork, the software product first computes the maximum purchase price as if the painting were a masterpiece by the same artist. Then, if the picture was painted in the 21st century, it multiplies that figure by 0.25, otherwise it multiplies it by $(21 - c)/(22 - c)$, where c is the century in which the work was painted $(12 < c < 21)$.

2.3 For any other painting, the software product computes the maximum purchase price from the formula $F \times A$, where F is a constant for that artist (fashionability coefficient) and A is the area of the canvas in square centimeters. If there is no fashionability coefficient for that artist, Osbert will not buy the painting under consideration.

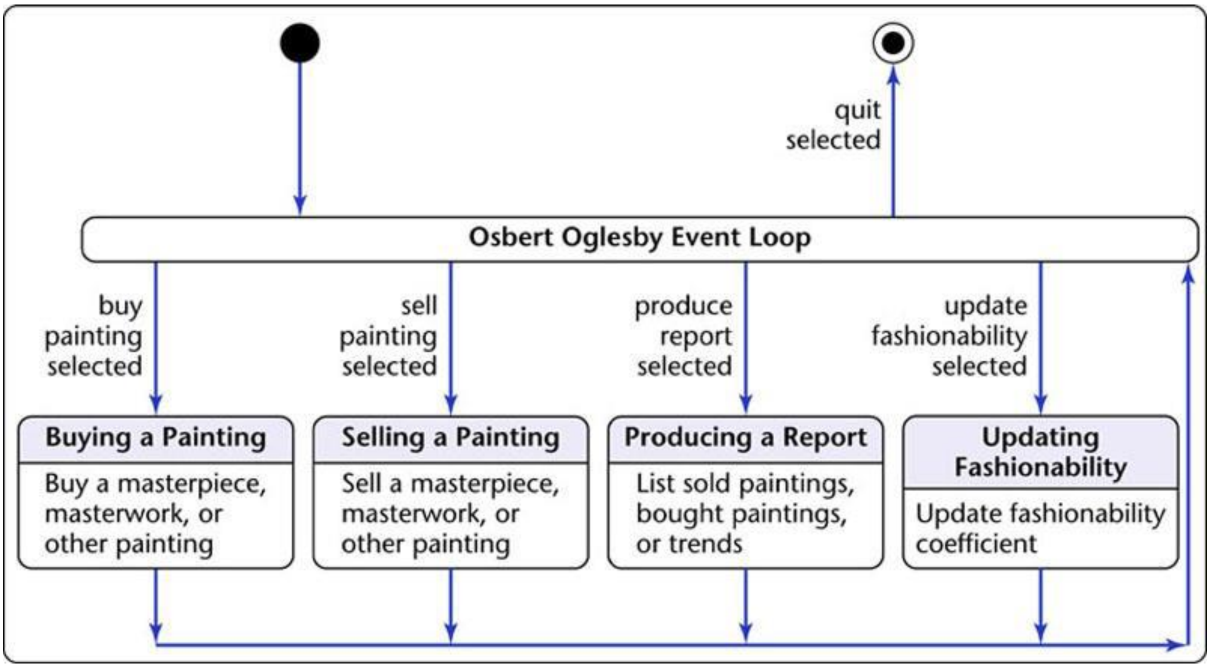
3. If Osbert buys the painting, he enters further details. These are
   Date of purchase
   Name of seller
   Address of seller
   Actual purchase price
4. The software product then records the following:
   Maximum purchase price determined by the algorithm
   Target selling price

פרטי המכירה, באופן דומה לפרטים שהוא מכניס בזמן שהוא מוכר את הציור (המידע הזה יישמר לטובת דוח"ות וחישובים עתידיים.

השלב הרביעי האחרון (ירוק) מתמקד בשמירת המידע הרלוונטי לרכישות הבאות – מחיר המקסימום, והמחיר המוצע לטובת היצירה.

המודל הדינאמי

עד עכשיו המחלקות והמודלים הראשוני היו די סטטיים. במחלקות לא היה רשום לנו את סדר הפעולות מלבד בתיאור, וגם הUC לא ממש הראו רצף של פעולות. עכשיו אנחנו מתחילים לדבר על הרצף. ניצור לנו דיאגרמה דינאמית ראשונית, שתביל "לולאה" של פעולות – תחת מעטה גג של "פעולות שצריך לעשות" נפרט את הUC השונים, לרמת הפונקציות והתיאור הכללי שלהם, באופן הבא:



למעשה, יש פה מעין תפריט, ממנו אוסברט בוחר כל פעם את הפעולה, עושה את הדרוש, וחוזר אחורה.

<u>חילוץ של מחלקות התחום</u>

עכשיו אנחנו יכולים להגדיר מחלקת Boundary, על ידי זה שנגדיר את כל הלולאה הזאת תחת תחום אחד, את המחלקה הזאת אנחנו נגדיר במחלקה שקשורה ל-UI, ובה יופיע התפריט של הפעולות:

כל תפריט, יפתח לנו איזה תחום הגדרה נוסף, אבל הוא יהיה תחת מחלקת היישות, או המחלקות החישוביות יותר.

תחומי ההגדרה שכן נתחשב בהם, יהיו הפקת הדוחות, מאחר שכל דו"ח פותח לנו עולם דיון אחר לגמרי.

המחלקות הנוספות יהיו:
דו"ח רכישות.
דו"ח מכירות.
דו"ח טרנדים עתידיים.

ניתן לראות בקלות, שאמנם שני הדוחות הראשונים קשורים אחד לשני באופן מסוים, אבל המכירות והרכישות בסוף מגיעים למקומות

שנים לגמרי. דו"ח הטרנדים, לעומתם, בכלל לא מתייחס למידע מקומי בתוך הגלריה, אלא למידע שנאסף על בסיס אופנתיות, אירועים שונים וכו'. יש לשים לב, שאת מחלקות התחומים האלה אנחנו ממיינים לקלט/פלט -כאשר מחלקת תחום הUI תהיה הקלט, והמחלקות החדשות עבור הדו"חות יהיו, כמובן, פלט.

<u>חילוץ של מחלקות השליטה</u>

גם את מחלקות החילוץ קל מאוד לזהות, אלו כל המקומות שדיברנו בהם על חישוב שהוא לא טריוויאלי – אנחנו לא צריכים להגדיר מחלקת שליטה עבור כל add קטן שאנחנו עושים, אבל כמו שראינו בUC השונים יש לנו חישובים שמתייחסים שונה לכל סוג יצירה, לא דין מאסטרפיס כדין יצירה פשוטה. לכן אנחנו נוציא את החישוב הזה לשלוש מחלקות שליטה שונות. בנוסף, יש לנו את הפונקציה שמעדכנת את הטרנדים העתידיים, גם היא, מן הסתם, לא חישוב סטנדרטי. ולכן נגדיר את ארבעת המחלקות הבאות:

מחלקת חישוב מחיר מקסימום ל-Masterpiece.

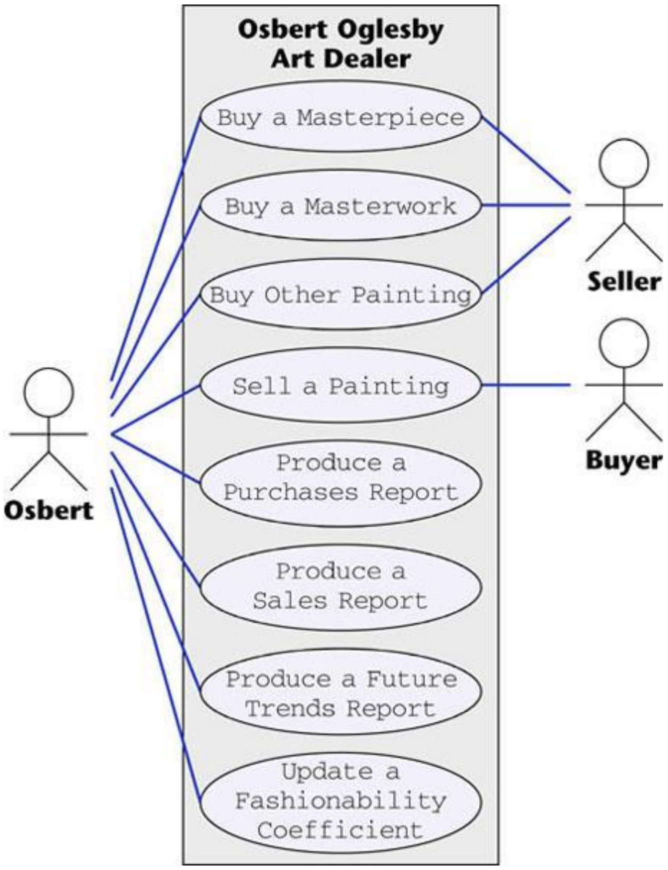
מחלקת חישוב מחיר מקסימום ל-Masterwork.

מחלקת חישוב מחיר מקסימום ליצירות אחרות.

מחלקת חישוב טרנדים עתידיים.

<u>דיון מחודש ב-Use Case</u>

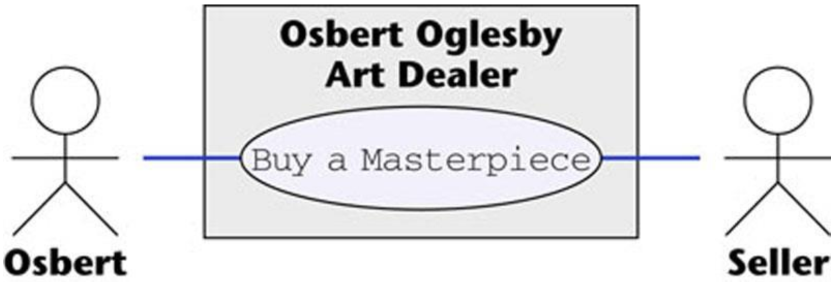
לאחר שחידדנו את המחלקות של ניהול והחישוב, אנחנו מבינים שרכישת יצירה, היא לא מקרה בודד של UC, אלא שלוש מקרים שאינם חופפים – עבור כל יצירה במיקום של החשיבות שלה, אנחנו נפעיל UC אחר. באופן דומה, גם את הפקת הדוחות אנחנו נצטרך להפריד למחלקות השונות שהגדרנו – אם המחלקות ודרך הפעולה בכל הפקה של דו"ח היא שונה, אין סיבה שהכל ייכנס תחת אותו דיון. ולכן דיאגרמת ה Use Case תשתנה לצורה הבאה-



אבל זה לא מספיק, מאחר שהגדרנו פיצול לUC אנחנו עכשיו צריכים לחזור אחורה, ולעדכן גם את הטבלאות. מה שהיה טבלה בודדת של רכישת ציור, יעבור להיות עכשיו שלושה טבלאות שונות עם פירוטים עבור כל סוג רכישה.

באופן דומה, גם הפקת הדו"חות יעבור שינוי ופיצול, ונצטרך לשכתב את הטבלאות המתאימות להן.

עכשיו נגדיר את UC רכישת מאסטרפיס –



ונצרף גם את הטבלה המתאימה שלו:

Brief Description

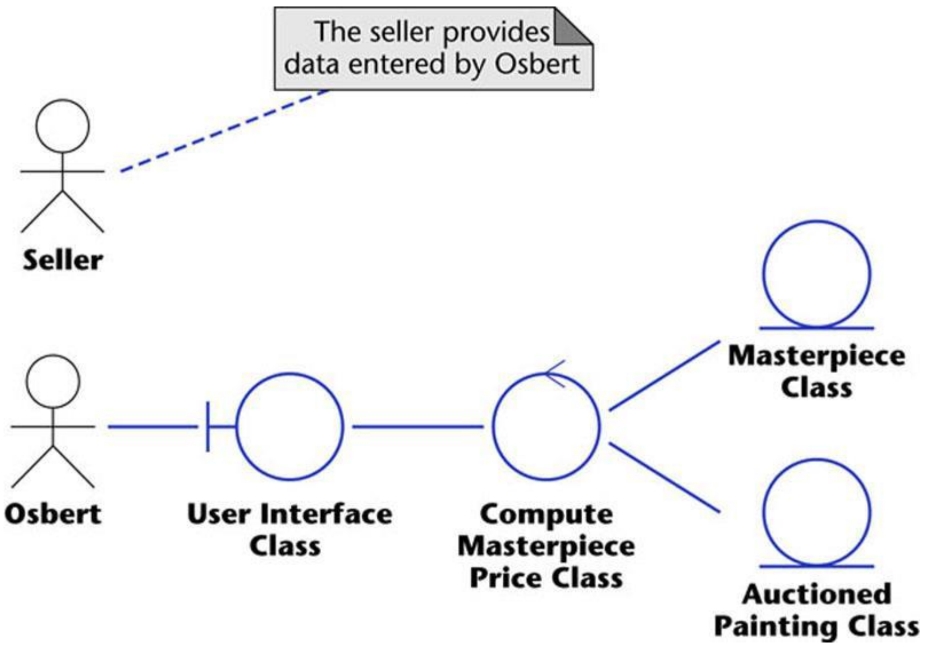
The `Buy a Masterpiece` use case enables Osbert Oglesby to buy a masterpiece.

Step-by-Step Description

1. Osbert inputs the details of the masterpiece he is considering buying. These are
   First name of artist
   Last name of artist
   Title of work
   Date of work
   Height
   Width
   Medium (oil, watercolor, other medium)
   Subject (portrait, still life, landscape, other subject)
2. The software product responds with the maximum purchase price Osbert should offer. In more detail, the software product computes the coefficient of similarity between each painting for which there is an auction record and the painting under consideration for purchase. Question marks in the first or last name of the artist or in the title or date of the work are to be ignored.
   The software product scores 1 for a match on medium, otherwise 0.
   The software product scores 1 for a match on subject, otherwise 0.
   It adds these two numbers, multiplies by the area of the smaller of the two paintings, and divides by the area of the larger of the two.
   The resulting number is the coefficient of similarity.
   The software product finds the auctioned painting with the largest nonzero coefficient of similarity. If there is no such painting, Osbert will not buy the painting under consideration.
   The software product computes the maximum purchase price by adding to the auction price of the most similar work 8.5%, compounded annually, for each year since that auction.
3. If the seller accepts Osbert's offer, Osbert enters further details. These are
   Date of purchase
   Name of seller
   Address of seller
   Maximum purchase price determined by the algorithm
   Actual purchase price
   Target selling price

למעשה, הטבלה הזאת היא בדיוק התיאור של רכישת ציור שהיתה לנו קודם, אך אחר ואנחנו הפרדנו את הרמות השונות, אנחנו נפריד גם את התיאורים, באשר כל רמת יצירה, תיקח איתה את הפסקאות הרלוונטיות אליה, אותם הגדרנו במעמד מחלקות השליטה.

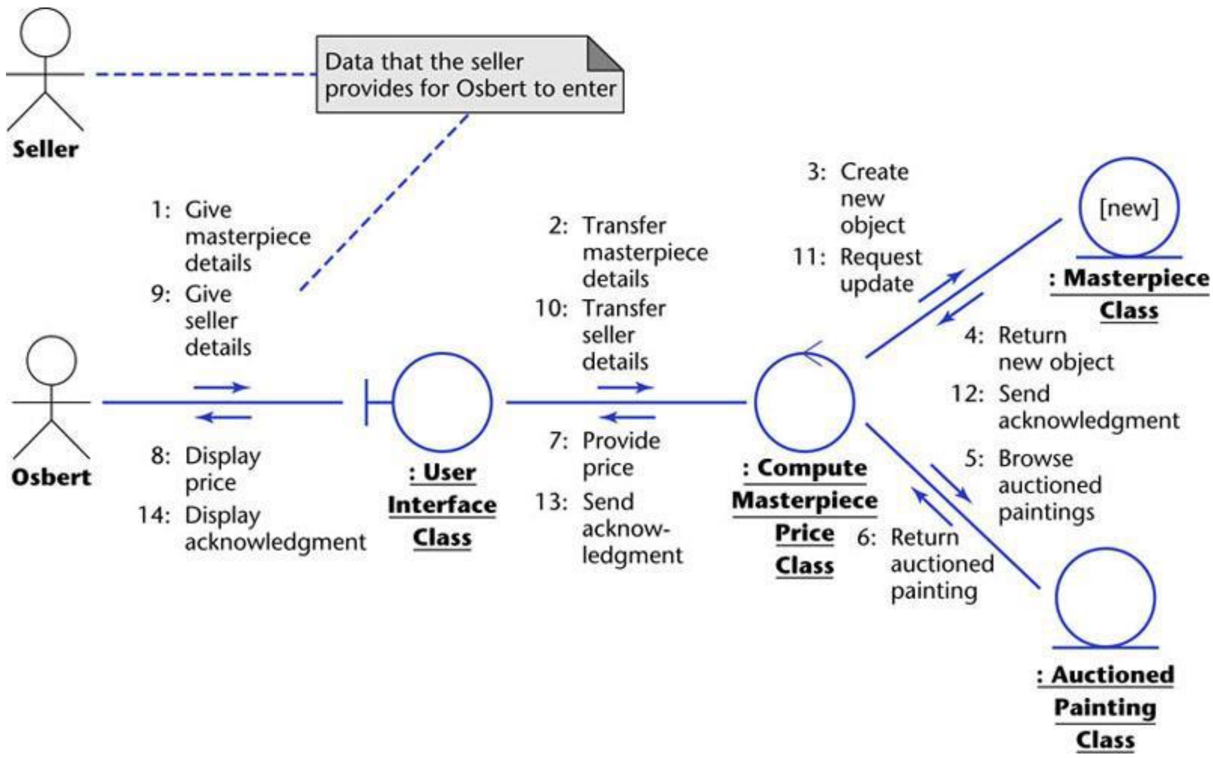
עכשיו מה שנותר לנו הוא לתאר את יחסי הגומלין בעבודה בין המחלקות השונות. נמשיך להתייחס בדיון לרכישת יצירת-מופת. התרשים הבא יראה לנו את שלבי העבודה בצורה שנוכל כבר להתחיל לתאר לנו איזה מחלקה מפעילה איזו מחלקה, בשביל לקבל את המידה הרצוי.



אנחנו מתחילים כמובן, עם השחקן שמציע יצירה למכירה, ומספר את הנתונים המתאימים עליה. אוסברט ניגש לתבנה, ודרך ה-UI מכניס את המידע הדרוש, המידה עובר הלאה למחלקה שאמורה לחשב את המחיר של המאסטרפיס, בהתחשבות בנתונים הכלליים של יצירה שכזו, ובמידע קיים של מכירות קודמות.

תרשים שיתוף-פעולה Collaboration Diagram

התרשים הבא שנבנה, מושתת על דיאגרמת המחלקות הבאה, אך מתארת ממש את הזרימה ודרך העבודה בין המחלקות, ואיך הכל עובר שלב אחר שלב:

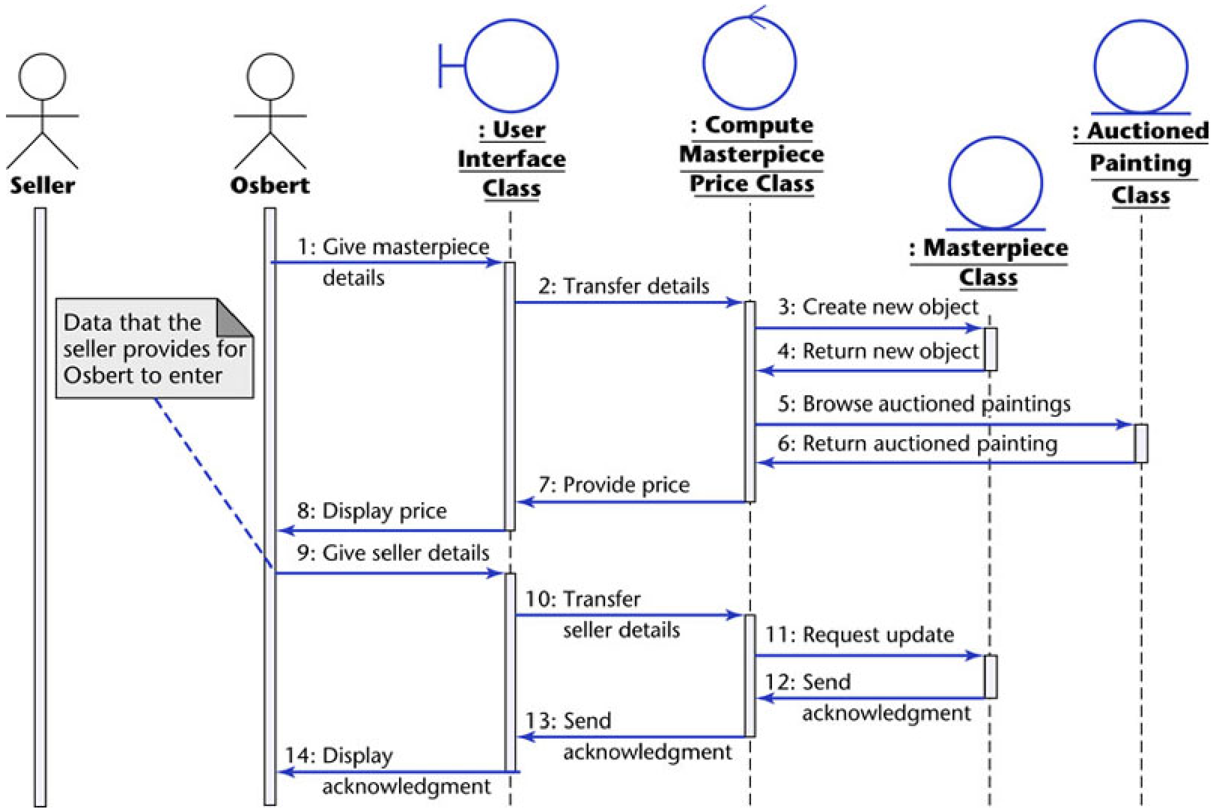


נשים לב מה קורה פה – על כל קו זרימה אנחנו מגדירים מה הקלט ומה הפלט, כאשר הקלט הוא בדרך כלל הצד העליון. אם אנחנו משתמשים פעמיים בזרם, אז הם יעמדו אחד מתחת השני. אם כן, רכישת

מאסטרפיס עובדת בשתי איטרציות, קודם כל מכניסים פרטים על מנת לקבל את המחיר אותו אנחנו מעבירים למוכר (1-8). אם המחיר מקובל עליו, אנחנו ממשיכים הלאה ומעדכנים את השדות הרצויים בשביל לבצע את הרכישה עצמה(9-14).

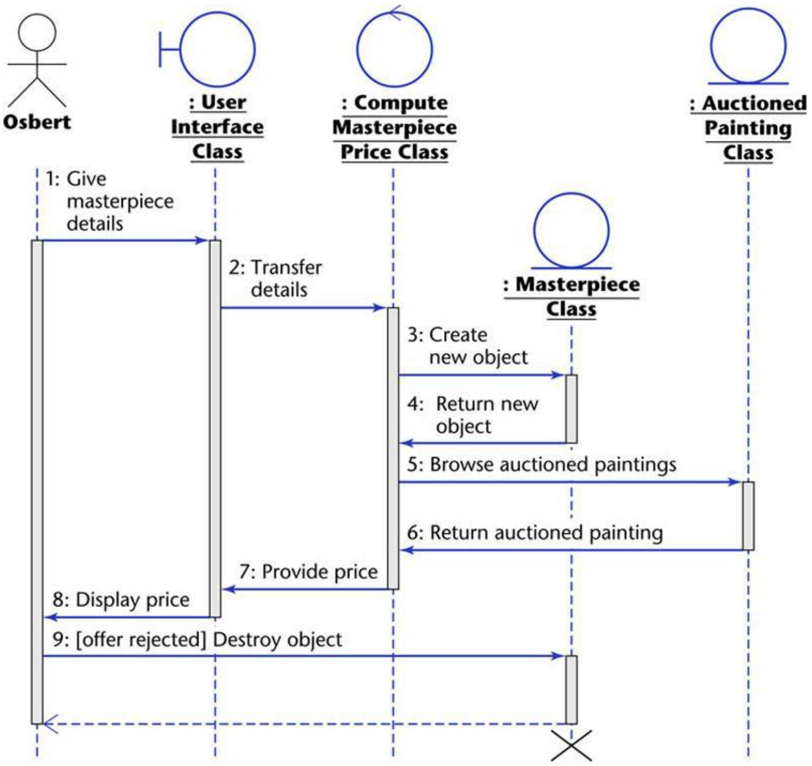
תרשים רצף Sequence Diagram

בשלב הבא, אנחנו מתארים את רצף האירועים במעין "ציר-זמן", כאשר הסדר לקוח מתרשים ה-Collaboration. אנחנו מחלקים את התרשים לשחקנים השונים, ולמחלקות אותם אנחנו מפעילים, כאשר כל ביצוע עובר בין המחלקות השונות על פי מה שקבענו-



כל איזור פעילות, גורם לקו להיות פעיל (עובי שונה בתרשים), כאשר אנחנו ניגשים למחלקה ויוצאים ממנה ישר (5-6) הרצף הוא מאוד קצר, לעומת זאת אוסברט והמוכר נשארים פעילים לאורך כל העבודה.

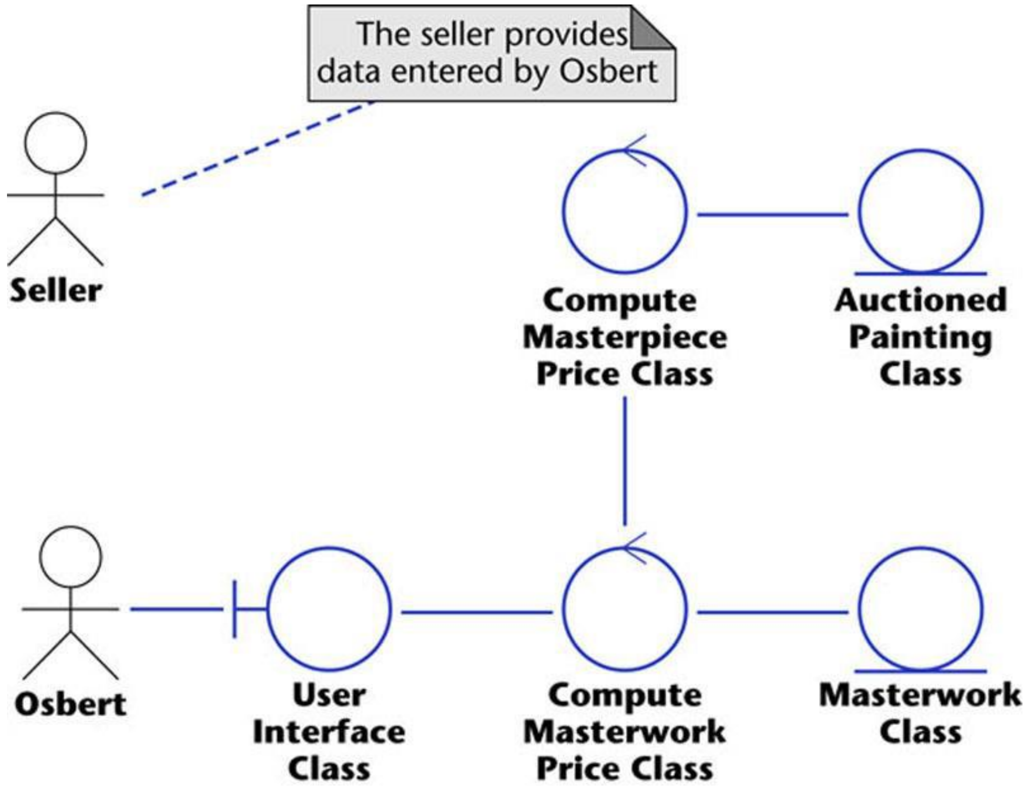
נשים לב, שמבחינת מוכר היצירה, אין הרבה "פעולות", הוא מביא את המידע לאוסברט, ומקבל ממנו הצעות, אבל כל עוד לא נפטרנו מהצורך שהוא יהיה קיים ורק יעמוד שם, מבחינתנו הוא עדיין פעיל.. ראינו לגבי ה Use Case, שאנחנו צריכים לשים לב לאירועים אלטרנטיביים. אמנם זה יכול להיות מאוד נחמד, אבל לא תמיד המוכר מרוצה מהמחיר המוצע של אוסברט, ועלול לבטל את כל העסקה. במקרה כזה אנחנו צריכים להביא גם תרשים רצף אלטרנטיבי המתאר מקרה כזה:



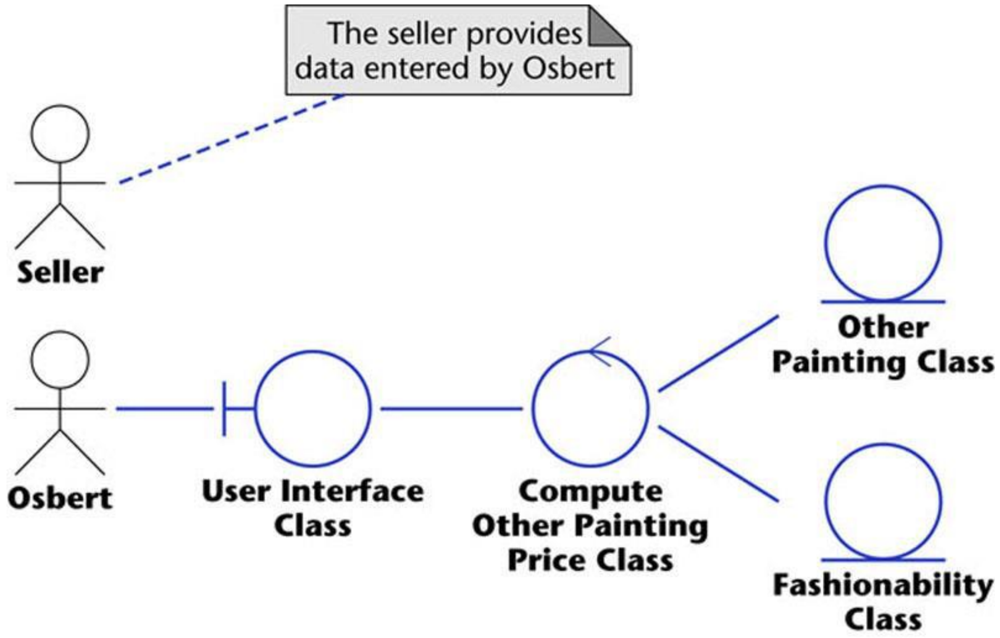
אם ההצעה נדחית, סוגרים את הבאסטה וממשיכים הלאה (או מציעים מחיר אחר, אבל זה בעצם חוזר לראשית התור).

עכשיו נוכל לראות את הדיאגרמות של רבישת מאסטרוורק וציור אחר ונראה את ההתייחסות למחלקות השונות-

קניית Masterwork:

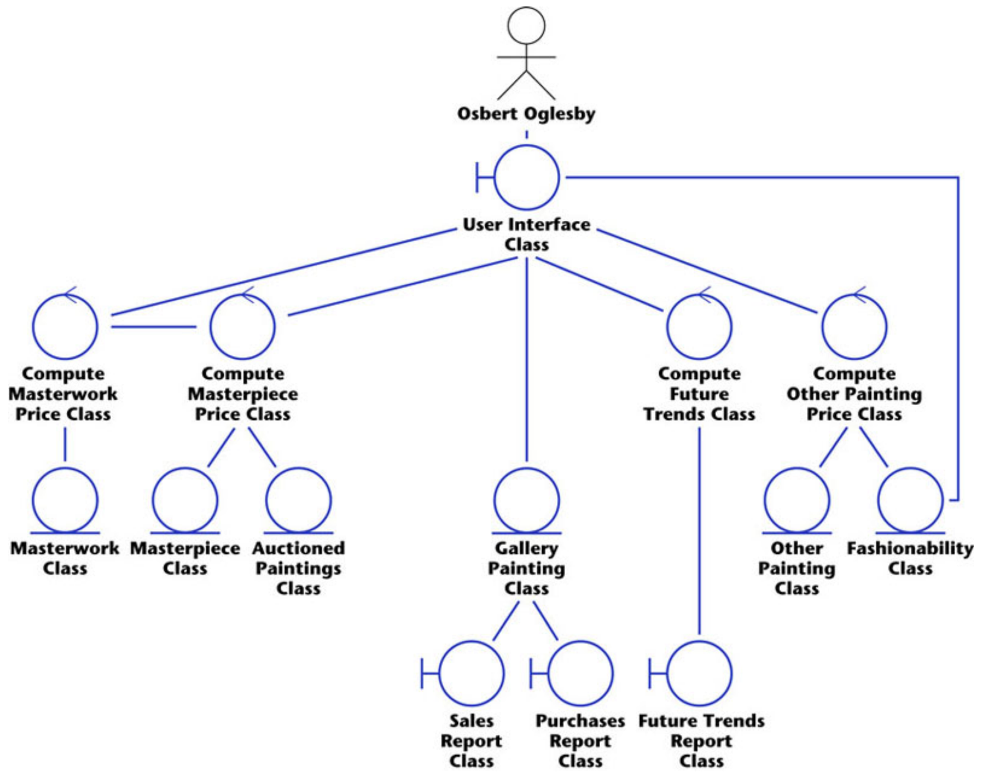


קניית יצירה אחרת:

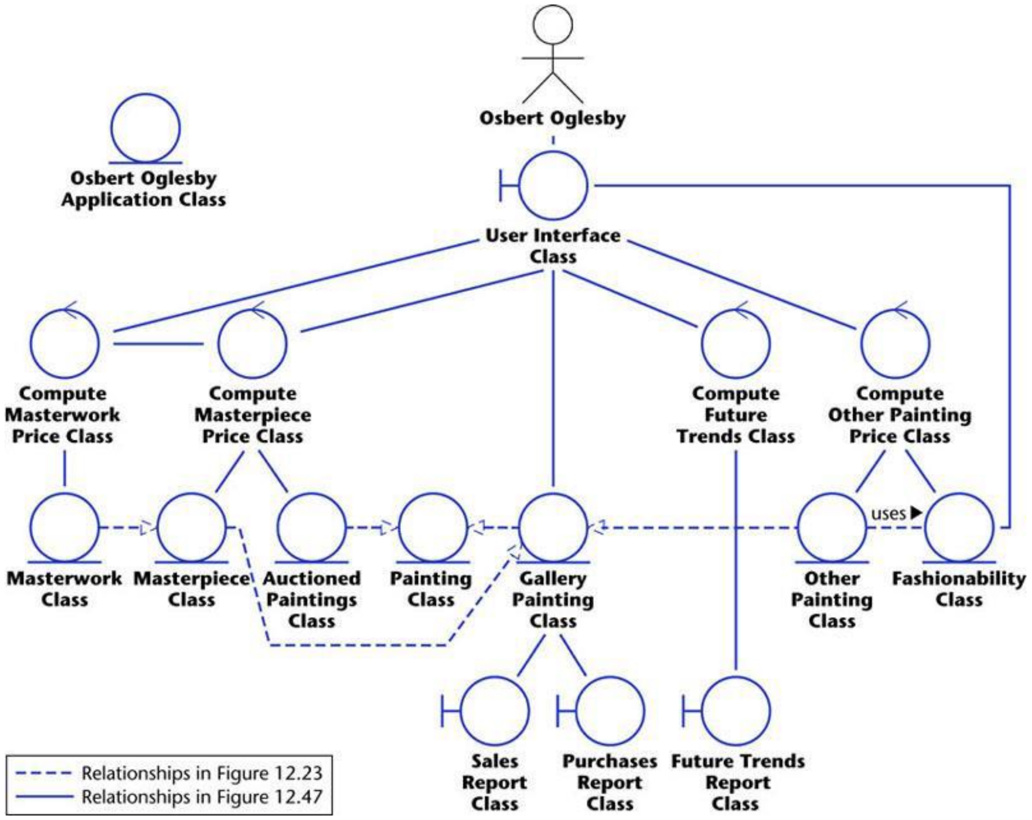


אחרי ששינינו את כל המחלקות האלה, אנחנו מבינים שעלינו לשנות גם את ההתייחסות למחלקת ה-UI.
אנחנו צריכים להוסיף את האופציות האלה בקלט נפרד. ולכן אנחנו נוסיף את האפשרויות הבאות –

עכשיו נוכל לחזור ממש להתחלה ולהגדיר את היחסים השונים בין המחלקות – נתייחס לכל המעברים בין המחלקות, מי משתמש במי וכו', ונוציא את הדיאגרמה הבאה:

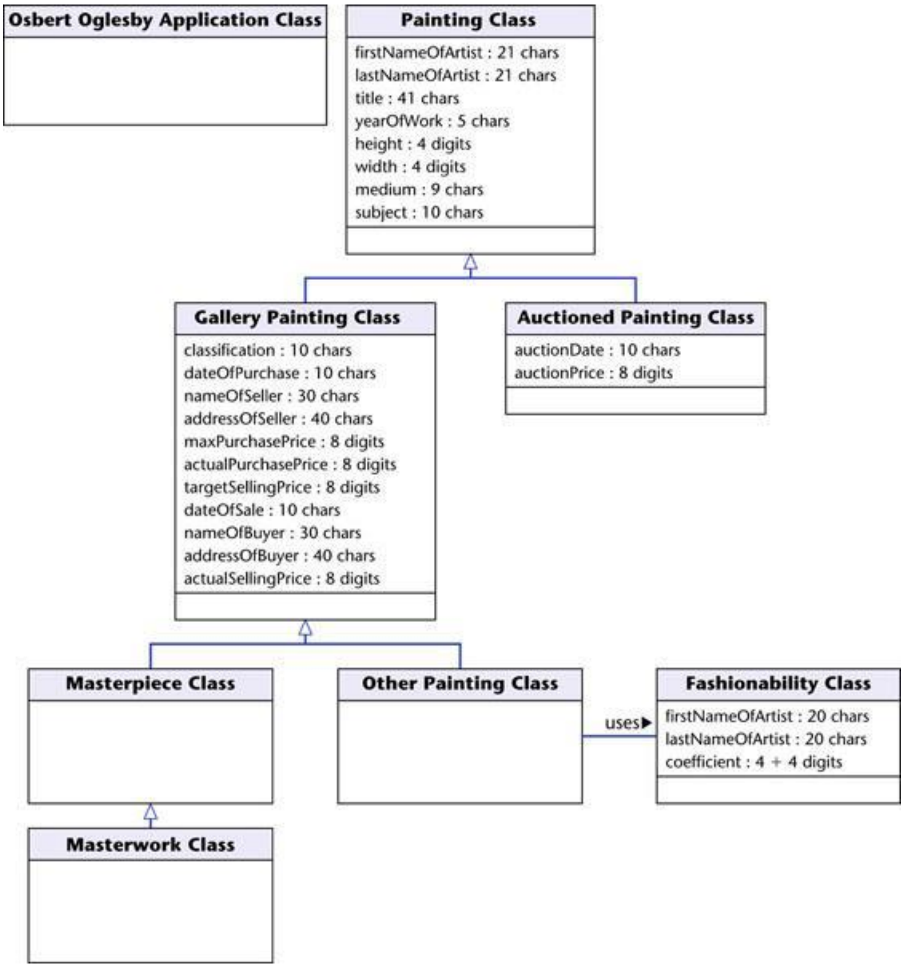


הפרדנו כל מחלקה לאופן בו אנחנו מגיעים אליה, אבל ישנם עוד יחסים אותם הגדרנו שלא הכנסנו לדיאגרמה הזאת. לכן נעשה איטרציה נוספת, ונוסיף לגרף את התיאורים הבאים:



היחסים שנוספו פה הם כל הירושות שהגדרנו אי אז בהתחלה, וחשובות בשביל להבין מה נוצר בכל פעם.

עכשיו נוסיף לתרשים המחלקות גם את כל המאפיינים שהגדרנו שצריכים להכנס – כל המידע הרלוונטי מבחינת מכירה/רכישה של היצירה, צריך להיות חלק אינטגרלי מהמחלקה.



ועכשיו אנחנו צריכים להוסיף את כל הפונקציות הרלוונטיות לכל מחלקה – הGet, Set שאותם אנחנו מכניסים במחלקות בהם אנחנו ניגשים לפרטים – כלומר – אנחנו לא נגדיר גטרים במחלקה של מאסטרפיס שתיגש עד מחלקת האב של יצירה, אלא נטמיע את הGetName (למשל) במחלקת האב העליונה ביותר.
וסיימנו את השלב הזה

שלב ההטמעה Transition Phase

לאחר שהכל מוכן, אנחנו מטמיעים את המערכת אצל הלקוח. כמובן, שלא מדובר בלזרוק אצלו את התכנה וללכת הלאה, אלא ממש להכניס, להדריך אותו על כל התכנה והפיצ'רים השונים שהיא מכילה, וגם אז, עלינו להמשיך ולהשאר איתו בקשר רציף, על מנת לבדוק אם יש איזה באגים שצריך לתקן או דברים שצריך להוסיף. ותמיד יש.

במערכות גדולות, שלב המעבר עלול אפילו לקחת שנים, כי לא יכולים לעשות את כל הדעכונים במכה אחת. למעשה, לפי מה שלמדנו, אנחנו ממשיכים עם שלב זה, עד שאנחנו מגיעים לרמה שאנחנו אומרים שאנחנו יכולים להפסיק לתת שירות לתכנה ולתת לה לגווע לאיטה.

פיתוח תכנה בעזרת שיטות Agile[10]

את שיטות האג'ייל, הזכרנו לא מעט תוך כדי הקורס, וגם בקורסים אחרים, בתור השיטה המרכזית איתה עובדים היום בפיתוח. עכשיו נכנס קצת לעומק השיטה ולאופנים המיוחדים בה. אחד הדברים המיוחדים בשיטה זאת, שבעוד כל השיטות שנעשו ונחקרו עד לאותו זמן, היו שיטות עסקיות. "עסקיות", במובן שאמנם ההטמעה היתה במערך התכנות, אבל כל הביסוס של השיטה הגיע מעולם העסקים ודרכי העבודה שלהם. כמו שראינו בפרק על השיטות השונות, אמנם הדברים האלה מאוד טובים בתאוריה ובעולם העסקים, אבל תכנת מחשב היא דבר דינאמי שלא תמיד יכול לקבל את הרדוקציה הזאת.

לעומת כל זה, קמה שיטת האג'ייל, שנוצרה על ידי אנשי מחשוב. אם נסתכל קצת על האנשים שכתבו את המניפסטים, נראה שהיה שילוב של אנשי אקדמיה מהטובים בתחומם, ואנשי תעשיה מוצלחים, והשילוב של שניהם, הוביל את השיטה להיות מדויקת ונכונה מאוד להנדסת תכנה. השיטה נהייתה כל כך טובה, שלקחו אותה הלאה, ועולם העסקים ניסה ללמוד משיטה זו וללמוד ממנה מה שאפשר.

הסיבה שהתחילו בכלל לחשוב על מסלול חדש בהנדסת תכנה, נבע מכך שעם כל השיטות הקיימות, עדיין אחוז גדול מהתכנות נפל, ורמות הטעויות והכשלונות היו כמעט בלתי נמנעות. יש לזכור, עד אז רוב השיטות היו מבוססות כל מודל מפל המים, והיתה הגבלה עצומה על היכולות לשינוי בתכנה וכו. עד שהגיעו מים עד נפש, והחליטו שצריך למצוא שיטה נכונה יותר.

המנשר הראשון

בחודש פברואר 2001, התכנסו 17 אנשי מחשבים באתר נופש (לא יודע למה זה חשוב, אבל כולם טורחים לציין שזה היה אתר נופש), כל האנשים הנוכחים היו האנשים המובילים בתחומם, וכולם הגיעו במטרה אחת מוצהרת – למצוא שיטה שתעזור לפתח את התכנה בצורה יעילה ונכונה, ולהגיב לשינויים בצורה מתאימה.

בראש כח המשימה עמד קנט בק - מיוזמי המנשר שלזכותו מאמרים וספרים רבים, ופיתח את שיטת XP שממשיכה את האג'ייל, ונדבר עליה בהמשך. ובנוסף, הוא אבי ה-Wiki, האנציקלופדיות המשותפות, וממפתחי הJunit.

איתו היו עוד 16 חוקרים ואנשי תעשיה, ולאחר מיצוי כל היכולות ואיחוד החשיבה, הם פרסמו את ה"מנשר" הבא, שהיווה את הבסיס הרעיוני לכל שיטות האג'ייל השונות[11] –

> אנו חושפים דרכים טובות יותר לפיתוח תוכנה תוך עבודה ועזרה לאחרים לעשות זאת.
> אלו הם ערכינו:
> **אנשים ויחסי גומלין על פני** תהליכים וכלים
> **תוכנה עובדת על פני** תיעוד מפורט
> **שיתוף פעולה עם הלקוחות על פני** משא ומתן חוזי
> **תגובה לשינויים על פני** מעקב אחרי תוכנית
> כלומר, בעוד שיש ערך לפריטים בצד שמאל, אנחנו מעריכים יותר את הפריטים **בצד ימין**

[10] מצגת 8

[11] טיפ למבחן – כמה שאפשר ללמוד את העקרונות והערכים בעל פה. זה בדרך כלל נותן כמה נקודות יפות.

מה המשמעות של המנשר הזה,ומה הערכים שהובלו בו? השיטות השונות עד לאותה תקופה, היו שיטות מאוד סדורות וברורות מההתחלה ועד הסוף – תעשה את שלב 1, תעבור לשלב 2 וכו'. עבור כל שלב היו את עשרות הפרוטוקולים והטפסים שצריך למלא – תחשבו רק על הפרוייקט המדומיין שאנחנו יוצרים בתרגול – כמה כתיבה וחירטוטים רק בשביל פרוייקט שהוא אפילו לא יגיע לכדי כתיבת שורת קוד בודדת! בנוסף, החוזים היו מאוד סגורים לשינויים מול הלקוח – אנחנו סוגרים איתו את מה שנעשה, ואז פעם הבאה שאנחנו נפגשים, זה כאשר הלקוח מקבל מאיתנו את התכנה בצורה המוגמרת. והדבר הנוראי ביותר – אם אנחנו עוקבים אחרי כל התבנית בצורה מושלמת, כל שינוי הכי קטן יכול להיות הרה-אסון, עשינו הכל לפי התבנית אבל טעינו באיזה דרישה, אז בסוף אנחנו עלולים להגיע למצב בו אנחנו בכלל במקום אחר.

כנגד כל זה, אומרים אנשי האג'ייל את הסעיפים הבאים:

אנשים ויחסי גומלין על פני תהליכים וכלים

השאיפה שלנו היא להגיע למצב בו אנחנו עובדים בצורה מסודרת ומתאימה, כלומר, כל התהליכים שלמדנו וכל הכלים היקרים ביותר, לא יועילו לנו אם אנחנו לא מסוגלים לעבוד כצוות – אנחנו צריכים להשקיע מאמץ ביצירת צוות שעובד נכון ובצורה נבונה, ודבר כזה יפצה אפילו על כלים שהם לא ברמה הכי גבוהה.

בגישות המסורתיות נתנו את עיקר החשיבות לתהליך המובנה, הדרך הברורה והמסודרת שצריך לעבור. בנוסף, עיקר ההתמקדות היתה בכלים השונים שיהיו ברמה הגבוהה ביותר.
בגישה האג'ילית לעומת זאת, התמקדו באנשים על פני התהליך – כלומר, ההצלחה של כל התהליך לא תלויה דווקא בכלים הגדולים ביותר. אפשר לקחת כלים שהם סבירים, ואנשים שהם מתכנתים שהם ברמה טובה, אבל הם מסוגלים לעבוד ביחד בצורה טובה, וזה יותר נוח, מאשר איש אחד גאון שלא מסוגל לעבוד עם אף אחד אחר.

תוכנה עובדת על פני תיעוד מפורט

בסופו של דבר, אנחנו אנשי תכנה, התיעוד המפורט של כל דבר שנעשה אולי נראה נכון, אבל הוא מבזבז לנו זמן יקר. אם נכתוב את התכנה כמו שצריך, והיא תעבוד, התיעוד יהיה משני להכל.

בגישות המסורתיות על מנת להבין את התכנה ואיך היא עובדת, היית צריך לקרוא את כל המסמכים, והתיעוד שעל גבי הקוד. כל מסמך שכזה יכול להיות ארוך מאוד – עבור כל פונקציה צריך להוציא בסופו של דבר מסמך. כאשר מגיע איש צוות חדש, והא צריך לעבור על כל הטופסיאדה ורק אז הוא יוכל להתחיל לעבוד
בגישות האג'יליות מתמקדים בהעברת מסר דרך הקוד. אם הקוד כתוב היטב (שמות משתנים ופונקציות משמעותיים למשל), אפשר לקרוא את הקוד ממש כמו סיפור, ולהבין ישר מה קורה ומה הולך לאן. במויות המסמכים הם מאוד נמוכות יחסית, מאחר ואם אנחנו צמודים למסמכים, כל שינוי קל בתכנה דורש גם לשנות במויות של ניירת, אם יש חוסר סנכרון קטן, כל הטפסים לא רלוונטים כי הם מתייחסים לדבר לא קיים. כאשר יגיע עובד חדש, פשוט ישב איתו מישהו ויסביר לו את המערכת, ברגע שגם היא תהיה בנויה נכון, אז לא יהיה צורך להתעכב יותר מידי על כל דבר.

שיתוף פעולה עם הלקוחות על פני משא ומתן חוזי

ברגע שהלקוח הוא חלק מתהליך העבודה, ולא רק מטרד של חוזים וכסף, העבודה שלנו תצליח יותר, ותקלע יותר לטעם הלקוח – לקוח מרוצה, אומר שאנחנו מרוצים.

בגישות המסורתיות הזמנת תוכנה היא הזמנת מוצר. אתה יושב בבית ומחפש מה שאתה רוצה, ומחכה שהמוצר יגיע אליך – לכן, הרבה יותר קל ליצור חוזה, כי הכל חקוק באבן. כמו כן, אין סיבה לייצר אינטראקציה בין שני הקצוות של הלקוח-מתכנת, בדיוק כמו שאנחנו לא מזמינים ארוחה במסעדה ואז נדחפים לטבח ומתחילים לשאול שאלות.

בגישה האג'ילית אנחנו מתקשים לנסח חוזים בשלבים הראשונים. התכנית עוברת הרבה איטרציות ושינויים, כך שהמיר חייב להיות גמיש. החוזה לא יכול לציין עלויות ולוחות זמנים מסודרים, פשוט כי אין דבר כזה. מסיבה זו, אנחנו שואפים גם ליצור קשר עם הלקוח לכל אורך העבודה, לוודא שהלקוח יקבל את המוצר שהזמין (ושהוא יבין אחר כך את החשבון שיגיע אליו).

תגובה לשינויים על פני מעקב אחרי תוכנית

כל תכנית היא בסיס לשינויים, יכול להיות שבמהלך העבודה, הלקוח מבין פתאום שצריך לעשות שינוי, הוא מבין את הדרישות שלו יותר טוב, אבל אם אנחנו עוקבים אחרי התכנית, יהיה לנו מאוד קשה להגיב לבקשה שלו בצורה נכונה. לעומת זאת, אם אנחנו חיים על שינויים, אז נאפס את עצמנו לטובת הדרישה החדשה ונמשיך הלאה.

בגישה המסורתית התכנה היא קבועה ואחידה. התכנון עבור שינויים נעשה מראש ומוגדר בצורה ברורה כמה שנים קדימה.

בגישה האג'ילית בונים את התוכנית בצורה גמישה וכבסיס עבור השינויים שיבואו. כל התכנונים ייעשו לזמן הקרוב בלבד, ותכניות מאוד כלליות וגולמיות יתוכננו עבור זמן רחוק יותר, אך לא בצורה ספציפית.

חשוב לזכור, שהאג'ייל לא שורף את כל מה שהיה, ואומר "יאללה בלאגן!" ומוביל אותנו לעולם בלי נהלים וסדר עבודה, אלא מתמקד בפרופורציות של החשיבות וההעדפה שלנו על צורה עבודה "אנושית" על פני קיבעון מחשבתי.

נפרט בהמשך בדיוק על כל ערך, איך הוא עומד בשונה לכל שיטות העבודה שהיו נהוגות באותו זמן.

12 עקרונות האג'ייל

1. בראש סדר העדיפויות שלנו, הוא שחרור תכנה בעלת ערך (עובדת) ללקוח. שחרור התכנה יהיה כמה שיותר מוקדם ורציף – הדבר הראשון שהגדרנו בשונה מהשיטות האחרות, הוא שאנחנו שואפים לתכנה שתוכל להימסר ללקוח כמה שיותר מהר. מאחר ותמיד יהיו סבבי תיקונים, אנחנו לוקחים זאת בחשבון כחלק אינטגרלי מהעבודה עצמה, על ידי בניה מתמשכת.

2. לקדם בברכה שינויים בדרישות, אפילו בשלב מאוחר של הפיתוח. תהליכי אג'ייל רותמים את השינוי להיות יתרון תחרותי עבור הלקוח – שינויים לא נובעים מזה שהלקוח מבין שהוא טעה (לא תמיד, לפחות), אלא לפעמים נובעים מהבנה של שינוי פיצ'ר או הוספה של משהו למערכת שתוציא מוצר יותר מוצלח.

3. **לספק תוכנה עובדת לעיתים קרובות, מכמה שבועות לכמה חודשים, עם העדפה ללוח זמנים קצר יותר** – אנחנו מתעסקים עם מונח שנקרא מסגרות זמן (Time Box), בו אנחנו מגדירים תקופה של עבודה מאומצת על מנת לתת תוצר רציף.

4. **אנשי העסקים והמפתחים חייבים לעבוד במשותף על בסיס יומי לכל אורך הפרוייקט** – לא לעשות פגישות פעם בכמה זמן, אלא גם במימד העסקי, הכל צריך לעבוד בצורה יומיומית ורציפה.

5. **בניית פרויקטים היא מסביב לאנשים בעלי מוטיבציה. צור את הסביבה ותן את העזרה שהם זקוקים, תוך מתן אמון בחברי הצוות שהם יבצעו את העבודה** – שוב, אנחנו עוסקים ומתמקדים מאוד ביחסי אנוש, ובין השאר ביצירת סביבת עבודה נאותה – אנחנו לא מחפשים ליצור "סדנת יזע", אלא מקום מוצלח שאפשר לעבוד בו.

6. **הדרך היעילה והטובה ביותר להעברת המידע אל צוות הפיתוח וממנו הוא שיח פנים אל פנים** – שיחה אישית אל מל המפתח, גם נותנת לו להבין יותר טוב, וגם תורמת ליצירת סביבת עבודה בריאה.

7. **תוכנה עובדת היא המדד העיקרי להתקדמות** – ערימות של טפסים ורעיונות זה לא מספיק, היכולת להצביע ולומר – היינו בנקודה הזאת והתכנה עבדה כך, ועכשיו התכנה עובדת כך, זה המדד המוצלח ביותר להראות התקדמות נכונה.

8. **תהליכי Agile מעודדים פיתוח בר קיימא. נותני החסות, המפתחים והמשתמשים צריכים להיות מסוגלים לשמור על קצב קבוע וללא הגבלת זמן** – ברגע שהמשתמשים והספונסרים רואים שיש התקדמות, קל להם יותר להשאר נאמנים לחברה. ובהתאם, היכולות שלנו להמשך גבוהות יותר.

9. **תשומת לב מתמשכת למצוינות טכנית (איכות גבוהה) ועיצוב נכון מגדיל את היכולת האג'ילית** – היכולת להיות "זמיש" (הלחם של זריז וגמיש) נובע ממחשבה מתמשכת ותשומת לב עקבית על הפרוייקט.

10. **פשטות הינה הכרחית** – שאיפה לדברים שלא מסתבכים מעל הראש.

11. **הארביטקטורות, הדרישות והעיצוב הטובים ביותר מתגלים מתוך צוותים המתארגנים מעצמם** – ברגע שהצוות נבון ומביר את העבודה שהוא אמור לעשות בצורה טובה, הם יוכלו לבנות את המסגרת המתאימה עבור המערכת הספציפית שלפנינו בצורה מתאימה יותר.

12. **במרווחי זמן קבועים, הצוות ינסה לייעל את עצמו (שיקוף עצמי: איך להיות יותר אפקטיבי!) הצוות ישנה את התנהגותו בהתאם** – ההתמקדות במתבנת מגיעה לכך שאדם נדרש להיות מסוגל למצוא דרכים בהם יוכל לתקן את עצמו ובצורה משותפת לייעל את כל העבודה והמערכת.

הבדלים בין הגישה האג'ילית לגישה המסורתית

מעבר לכל מה שציינו בכל אחד מהערכים, וההבדלים השונים. יש לעמוד על מספר גורמים שמשפיעים על הפרוייקט בכללותו. **היקף** - מה הדרישות של המערכת, עד לאיפה היא תגיע. **עלות** – נגזר מזה ישירות גם כמות המפתחים **זמן** – זמן העבודה של המפתחים, במסגרת השוטפת של העבודה על הפרוייקט, ובכלל בדרישה לתקופת זמן מסוימת. **איכות** – עד כמה המערכת מבצעת את הדרוש בצורה מלאה.

נמקד את שינוי התפיסה בגישה הבסיסית ובתהליך הפורמלי.

שינוי תפיסה בגישה הבסיסית

<u>חוזה</u> – בעוד שהחוזה המסורתי הגדיר את ארבעת הגורמים שהזכרנו, באג'ייל אנחנו לא מסוגלים להתחייב על ההיקף של העבודה, וההתחייבות על הזמן מגיעה בהתייעצות עם הלקוח – מהם מרווחי הזמן הדרושים לעבודה, ומהם המסגרות הכלליות שנעבוד לפיהם.

<u>פיתוח</u> – במסורתי, אנחנו מפתחים לפי דרישות הלקוח. אם הוא מוסיף דרישה, אנחנו נוסיף פיתוח ונמשיך לעבוד עליו עד כמה שיידרש. באג'ייל, יש איטרציות זמן קבועות, שעל פיהם אנחנו עובדים. אנחנו מפרקים את הדרישות למרווחי זמן ולא לרישה בכל פעם.

<u>תקציב ועמידה בזמנים</u> – בשיטות המסורתיות, עובדים ומתמקדים מאוד בענייני הדרישות של הלקוח, אך בסוף לא נשאר זמן עבודה עצמה, ועושים אותה מהר ובצורה פחות איכותית. אג'ייל יעדיף לוותר על חלק מהדרישות, ובלבד שנעמוד בזמן הדרוש וההגיוני עבור כל דבר. אם נצטרך עוד זמן, ניקח עוד זמן.

<u>תפקידים</u> – בגישה האג'ילית, הלקוח הוא חלק בלתי נפקד מצוות העבודה. בעוד שצוות הפיתוח יחליט איך לבנות כל דבר, הלקוח יגיד מה בכלל לבנות, ובמה להתמקד.

<u>לוח זמנים</u> – בגישה המסורתית, אנחנו מנסים לחזות את לוח הזמנים וכמה זמן ייקח לנו כל תהליך – אנחנו בונים תוכנית ארוכת טווח, ומנסים לעמוד בה. באג'ייל, הלקוח הוא זה שיחליט את לוחות הזמנים עבור כל חלק, ואנחנו מתעסקים בסבבי השחרור השונים של התכנה, ומבחינתנו זה דבר קדוש.

<u>דדליין</u> – אם אנחנו מקובעים מאוד ללוח הזמנים, ואנחנו רואים שאנחנו מתקרבים לסוף ועוד לפנינו עבודה רבה, אנחנו נכנסים למצב טירוף ומשקיעים עבודה עד השעות הקטנות של הלילה רק בשביל שנוכל להגיד שעמדנו ביעד הזמן. באג'ייל, אנחנו מגדירים שבוע עבודה של X שעות. אם אנחנו עובדים יותר מזה, כנראה שאנחנו נותנים מאמץ במקומות לא נכונים, ואם בפחות מזה, אולי אנחנו מפספסים דברים חשובים.

שינוי תפיסה בפורמליזם התהליך

<u>טקסיות</u> – בגישה המסורתית, כל ישיבה היא ישיבה. בחדר מסודר ובמרווחי זמן מוגדרים. הכל מאוד רציני ומוגדר. לעומת זאת, ישיבות האג'ייל נעשות על בסיס יומי, ונעשות בעמידה. אנשים לא מתעפצים תוך כדי הישיבה וידברו בדיוק על הדרוש בצורה מהירה. חברי הקבוצה עומדים ביחד, וכל אחד מתאר את העבודה שהוא צריך לעשות במשך היום, ככה גם מונעים כפילויות, וגם כל אחד יודע להגדיר לעצמו מה הוא הולך לעשות.

שינוי תפיסה בביצוע שינויים בפרוייקט

<u>ביצוע שינויים</u> – בגישה המסורתית, השינוי הוא השטן ואבי כל חטא. ברגע שיש שינויים בדרישות אנחנו על המסלול הבטוח להתנגשות. באג'ייל אנחנו מאמצים את השינוי, ואם יש בעיה עם ביצוע של שינוי, כנראה הבעיה היא בכל תהליך העבודה.

שינוי תפיסה במעורבות המשתמש

<u>מעורבות לקוח</u> – כולם מודים שעל מנת שפרוייקט יעבוד צריך את הלקוח. נקודת המחלוקת, היא עד איפה צריך אותו. בגישות המסורתיות, ביקשו שהלקוח יהיה נוכח בשלב הדרישות, ומעבר לזה הוא לא רלוונטי ומפריע. הגישה האג'ילית סוברת שהוא צריך להיות חלק אינטגרלי מהתכנה, ולספק משוב לכל אורך הדרך.

המנשר השני

בעקבות ההצלחה של האג'ייל, והקבלה שלו בקרב חלקים רחבים, נוצר רצון לשפר את השיטה בעוד קצת. בשנת 2006, יצא המניפסט השני שנקרא "Manifesto for Software Craftsmanship" – המניפסט להפיכת התכנות לאומנות. שימו לב, לא מדובר פה על תכנות שיהיה יצירת אמנות, אלא ביסוס של

מקצועיות וידע טכני שיעלה את כל הנדסת התכנה לקומה הבאה. לערכים השונים שהיו כבר כתובים לנו, אנחנו מוסיפים מספר ערכים נוספים:

לא רק תוכנה עובדת, **אלא גם תוכנה איכותית**

לא רק מגיבים לשינויים, **אבל גם מוסיפים ערך לתוכנה בהתמדה**

לא רק אנשים ואינטראקציות, **אבל גם קהילה של אנשי מקצוע**

לא שיתוף פעולה הלקוח בלבד, **אבל גם שותפויות פוריות**

כלומר, במרדף אחר הפריטים בצבע כחול, מצאנו את הפריטים בצבע אדום **כהכרחיים**.

לא רק תוכנה עובדת, **אלא גם תוכנה איכותית**

על מנת שהתכנה תצליח, היא לא צריכה רק לעבוד, אלא להיות גם איכותית. לזה יש לנו הגדרות ברורות מה נחשב איכותי ומה לא. מהירות, נוחות וכו', וזה הכרחי לדאוג לכך שהתכנה תהיה גם איכותית, ולא רק המינימום הנדרש.

לא רק מגיבים לשינויים, **אבל גם מוסיפים ערך לתוכנה בהתמדה**

הערך הזה, לוקח את קבלת השינויים לשלב הבא – לא רק להגיב לשינויים, אלא ליזום שינויים! ברגע שאנחנו רואים שיש לנו את הדרוש, לחפש את הדרכים בהם אנחנו יכולים להוסיף לתוכנה ולמערכת בשביל שתהיה אפילו איכותית יותר מהדרוש.

לא רק אנשים ואינטראקציות, **אבל גם קהילה של אנשי מקצוע**

יצירת קהילה של אנשי מקצוע, עוזרת גם למתכנת הפחות-מוצלח להגיע לרמות גבוהות. מעבר לרמת הצוות, יצירת קהילה ופורומים שונים עוזרות לכל המפתחים להגיע למקצועיות. פתירה של שאלות ספציפיות וממוקדות גם ממקצעת את העונה וגם את השואל.

לא שיתוף פעולה הלקוח בלבד, **אבל גם שותפויות פוריות**

השאיפה לשותפות עם הלקוח, צריכה להתרחב לשיתוף פעולה עם אנשים נוספים ותחומים רחבים בכל האפשר, על מנת שנוכל בעזרת זה להגיע למקומות שלא היינו יכולים כאשר היינו מוגבלים תחת תחום מסוים. דבר זה הוא הכרחי, על מנת להתמקצע ולהבין תחומים רבים בכל האפשר.

מתודולוגיות זמישות

בעקבות האג'ייל, התפתחו עוד מספר שיטות, המבוססות על האג'ייל. נעבור בקטנה על חלק מהם:

XP (Extreme Programming) – שיטה זאת פותחה על ידי קנט בק (ממייסדי האג'ייל), ועיקרה היה להכניס את הבלאגן של האג'ייל תחת שיטה מאוד מוגדרת. עיקרי השיטה היו לעבוד בצוותים של זוגות, כאשר אחד כותב את הקוד והשני בודק אותו סימולטנית. בנוסף היו מספר כללים שהקשו על העבודה, ברמה שאם אפילו אחד מהכללים הופר, כל תהליך העבודה הלך לטמיון. בנוסף, מנהלים לא אהבו את השיטה של העבודה בזוגות, שגרמה להם לשלם כפול עבור שני מתכנתים על כל שורת קוד.

SCRUM – שיטה המתמקדת ביכולות הצוותים לנווט באופן עצמאי. זו השיטה הנפוצה ביותר היום, ומאוד חשובה, ויש עליה גם מצגת שלא למדנו בכיתה.

יש עוד מספר שיטות במצגת, אך אין חשיבות מיוחדת באף אחת מהן.

חסרונות השיטה

- לא מתאים לכל המקרים והארגונים – מתאים בעיקר לצוותים קטנים ובינוניים. חברה של 200 עובדים לא יוכלו לעשות ישיבות עמידה, גם יהיה קשה לעשות את חלוקת העבודה בצורה מסודר. בנוסף, מקומות בעלי חשיבות לתהליך מסודר כמו מקומות צבאיים וכדומה לא יכולים להתמודד עם כל המערכות בלי הנהלים המתאימים.
- מה קורה אם מאמצים רק חלק מההרגלים? בגדול, אנחנו לא מצליחים לעבוד. כל נקודת החוזק של השיטה, מתבססת על כך שמבצעים הכל. אי אפשר להיות רק חצי מסודר, וחצי מבולגן. אם יוחלט להשאיר את כל עניין הטפסים והנהלים, אז השיטה כבר לא אפקטיבית, כי לא מתעסקים ב"זריזות".
- שינויים מאוחרים עדיין יכולים לעלות הרבה – העלות של השינויים עלולים להיות גבוהים בלי קשר לשאלה האם אתה מקבל את השינוי בידיים פתוחות או לא.

מתי משתמשים בשיטה

- פרוייקט דינאמי עם קבוצה קטנה ובעלת מוטיבציה – על מנת שהאג'ייל יעבוד, צריך הירתמות של כל הצוות לטובת העניין. אם אחד מהצוות לא משתף פעולה ועובד "מסודר מידי", הוא לא יעיל ופוגם בכל יכולות הצוות.
- דרישות משתנות – אם אנחנו יודעים מראש שהדרישות עלולות להשתנות, אנחנו צריכים להיות מוכנים לזה, ולהמליץ על השיטה.
- יש לקוח מעוניין וזמין – האג'ייל דורש מהלקוח, לא פחות מאשר הוא דורש מהמתכנת, ואולי זה ייראה לו מגניב בהתחלה להיות ממש חלק מהפיתוח, אם הוא לא זמין ומגיב לשינויים, אנחנו מאבדים חלק ניכר מיתרון השיטה.

בדיקות[12]

בכל תכנה שנראה או ניצור בעצמנו נמצא שיש פגמים. חלק מהפגמים יהיו קטנים יותר וחלק גדולים, ככל שנעבוד מסודר יותר אנחנו נוכל למזער את התקלות השונות, אבל אין שום דרך לבטל את התקלות לגמרי (חשוב למבחן – אין תכנה בלי תקלות!). אם התוכנה לא עובדת בצורה שהיא אמורה לעבוד, זה אומר אחד משני דברים – או שיש לנו באגים בקוד המקור של התכנה – כלומר משהו כתוב לא נכון, או משהו גרוע יותר יש לנו מה שנקרא דפקט (Defect) שבכלל בנינו משהו לא נכון ברמת הדרישות.

בדיקות התכנה, הוא חלק שמשולב בתוך מחזור חיי התכנה, עוד משלב איסוף הדרישות (תלוי במודלים השונים), כאשר המטרה הכללית של הבדיקות הוא לזהות שגיאות שונות באחד מהתחומים הבאים:

בדיקות נכונות (Correction) – בדיקות אלו מוגדרות כולידציה[13] – בדיקות שהתכנה כמערכת עובדת בצורה הנכונה.

בדיקות שלמות (Completeness) – בדיקות וריפיקציה[14] - יוצאים מנקודות הנחה שהמתכנת הבין את הדרישות, מה שאנחנו צריכים לאשר פה, הוא שהביצוע עצמו נעשה כמו שצריך, ושאין טעויות בכתיבת הקוד בעצמו.

במילים פשוטות – אנחנו צריכים להיות ערוכים לתפוס את כל הבאגים (סכמה שאפשר) לפני שהלקוח יראה אותם. כי באלה אנחנו - מושלמים.

בדרך כלל, אנחנו כמפתחים נוטים לזלזל בבדיקות, כי "אנחנו ידעים שהקוד כתוב נכון!", אבל אנחנו צריכים לזכור שכל באג כזה שלא נתפס עלול לגרום במקרה הטוב, עיכוב קל או הפסד כספי, ובמקרה הגרוע יותר ממש לאובדן חיים. במצגת הובאו מספר דוגמאות מההיסטוריה הלא-יותר-מידי-רחוקה על כל מיני באגים שגרמו למוות של אנשים. רק תחשבו מה עלול לקרות אם יש באגים בתוכנה של רכב אוטונומי וכדו', זה יהיה לא נעים בכלל. אנחנו מבחינתנו צריכים להיות ערוכים להקטין כמה שאפשר את הסיכוי לבאגים מכל הכיוונים.

סוגים שונים של בדיקות תכנה

נעבור על הסוגים השונים של הבדיקות. חלק מהבדיקות מוגדרות של סוג בדיקה כנגד ההבדיקה ההופכית לה, וחלק כסט של בדיקות. עלינו להבין את ההבדלים השונים בבדיקות ולהבין את הדקויות.

- **בדיקה סטאטית מול דינאמית –** <u>הבדיקה הסטאטית</u> נעשית עוד בשלבים של כתיבת הפסודו-קוד. אנחנו כותבים לעצמנו מה אנחנו הולכים לעשות, ואז אנחנו מריצים על יבש את האפשרויות השונות בשביל לוודא שאנחנו עושים נכון. הצורה האופטימלית של הבדיקה הזאת היא לשבת מול מישהו שלא קשור לכתיבת הקוד, ולעשות את זה בצורה של מעין ריאיון, בו אנחנו בודקים את האפשרויות השונות.
<u>הבדיקה הדינאמית</u> לעומת זאת מתרכזת בבדיקות הרגילות שאנו כמתכנתים עשים – מריצים את הקוד עצמו, ובודקים האם יש נפילות וקריסות במקום כלשהו.
- **בדיקות פיתוח מול בדיקות עצמאיות –** <u>בדיקות הפיתוח</u> – אלו הבדיקות שהמתכנתים כותבים לעצמם, לאור הקוד שהם כתבו. במקרה המוצלח, אם עובדים בצורה של TDD, אז הבדיקות מוצלחות. במקרה הפחות טוב – המפתח בודק בדיוק את המקרים שהוא יודע שיעבדו, ואז לא

12 מצגת 11

13 Validation

14 Verification

עשינו כלום.

<u>בדיקות עצמאיות</u> – בדיקות שנעשות על ידי בודק חיצוני, או מפתח אחר שלא מתעסק במה שנכתב כבר בקוד, אלא כותב בצורה עצמאית ומנותקת. הוא יודע מה הפונקציה אמורה לעשות, וכותב את הבדיקות המתאימות אבל בצורה לא משוחדת.

- **בדיקות קופסא שחורה מול קופסא לבנה** - <u>קופסא שחורה</u> - לוקחים את המערכת ומתייחסים אליה לצורת שימוש של תכנה שאנחנו לא יודעים ולא אכפת לנו מה קורה בפנים, ומה שאנחנו הולכים לבדוק זה ההתנהגות של המערכת - כאשר מכניסים קלט, בודקים את הפלט היוצא האם הוא נכון או לא. אנחנו לא בודקים את כל התהליך של העבודה בפנים, אלא רק את שתי הקצוות של המערכת.

  <u>קופסא לבנה</u> - בדיקות שנכנסות ממש לתכנות, וחיפוש של בעיה בתוך הקוד. אנחנו עוברים על כל השלבים של התוכנית ובודקים האם משהו שם עלול להיות לא נכון. כאשר בדקנו רק את הפלט, יכול להיות שקיבלנו פלט נכון, אבל בטעות – למשל עשינו 7/7 וקיבלנו 1, 8-9 ושוב קיבלנו אחד וכו' אבל פשוט התכנה מוציאה 1 על כל קלט שנכניס, אם נמשיך לבדוק ככה, אנחנו עלולים בכלל לא לשים לב, ועבור פלטים אחרים אנחנו נקבל משהו לא נכון בכלל. כמובן שאנחנו גם לא יכולים לבדוק את **כל** הקלטים האפשריים כי הכמויות הקומבינטוריות של הדבר הזה הם עצומות, לכן אנחנו מוודאים שבאמת כל תהליך החישוב כתוב בצורה נכונה.

- **בדיקה אוטומטית מול בדיקה ידנית** – <u>בדיקה אוטומטית</u> – הרצת סקריפטים שונים על המערכת ובדיקה אוטומטית אם יש נפילות.

  <u>בדיקה ידנית</u> – בודק שעובר על כל האפשרויות ורואה היכן יש נפילות. חשוב לשים לב, שעל אף שהאוטומציה נראית לנו מאוד קורצת, דווקא הבדיקה הידנית (כל עוד היא נעשית נכון) הרבה יותר מוצלחת. בהשאלה מקורס האלגוריתמיקה, בדיקות אוטומציה הן בעיות הכרעה, הוא אומר לך איפה אתה נופל ואיפה לא. לעומת הבדיקה הידנית שהיא בעיית אופטימיזציה – ברגע שרואים שיש איזה באג, בודקים את כל הסביבה שלו, מה משפיע עליו ומה מושפע ממנו, וכך הבדיקה היא יותר יסודית.

- **בדיקות שפיות מול בדיקות קבלה ובדיקות עשן** – כאן יש לנו כבר תת-קבוצה – <u>בדיקות עשן</u> – נתחיל בכוונה מהסוף, השם של בדיקות העשן, נילקח בהשאלה מעולם האלקטרוניקה. נגיד מישהו מרכיב מעגל חשמלי, החלק המלחיץ ביותר, הוא השלב בו הוא מתחיל להעביר בו זרם חשמלי ואז הוא מחכה לראות האם יצא עשן מהמעגל או לא. באופן דומה, בדיקות העשן של התכנה הוא ברגע שקימפלנו ואנחנו מפעילים את התכנה האמיתית ומחכים לראות האם הכל יעלה נכון.

  <u>בדיקות קבלה</u> – דומות מאוד לבדיקות עשן, רק שבמקום לבדוק את ההפעלה הראשונה, אנחנו שמים את התכנה אצל הלקוח, ומחכים לבאגים שמגיעים בסביבה פחות אופטימלית מאשר המחשב שלנו שאנחנו מכירים.

  <u>בדיקות שפיות</u> – בדיקה שטחית במטרה לבדוק האם שינויים שנעשו בקוד בעקבות תיקונים לא פגעו במקומות אחרים בקוד.

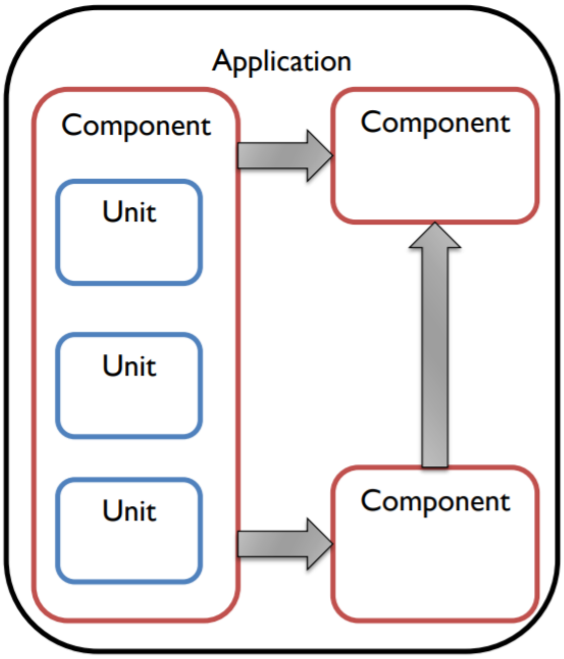
- **בדיקות רגרסיה** – דומה מאוד לבדיקות השפיות שנעשות לאחר שינויים בקוד, אך נעשות בצורה הרבה יותר מרוכזת. מטרת בדיקות אלו היא: 1. לוודא שהבאגים <u>תוקנו</u>. 2. פונקציה שפעלה בעבר <u>לא נפגעה</u> בעקבות השינויים. 3. תכונות חדשות <u>לא יצרו בעיות</u> בגרסאות קודמות.

- **בדיקות קופים** – בדיקות בהם נותנים לקבוצת נסיינים "לשחק" ולחקור את המערכת. המטרה שלהן היא לעבור כמה שיותר על כל המערכת בשביל לבדוק אם לחיצה אקראית על רצף מסוים תוביל לאיזה באג לצוף לפני השטח. נקרא בדיקות קופים, כי היינו יכולים להביא את הבדיקות האלה גם לחבורת קופים שסתם ירביצו למקלדת עד שיצא משהו.

עקרונות ביצוע בדיקות תכנה

1. בדיקות מראות נוכחות באגים – אמנם זה נראה דבר טריוויאלי לומר, אבל המשמעות של זה, היא שהדבר היחיד שאנחנו יכולים לומר בוודאות לאחר שמצאנו את הבאגים, זה שמצאנו באגים במקום מסוים. אנחנו לא יכולים להעיד בעזרת זה לא על באגים במקומות אחרים בתכנה ואפילו לא על אותו איזור. אנחנו נצא מנקודת הנחה שתמיד יש באגים שעדיין לא גילינו.
2. לא ניתן לבצע את כל הבדיקות והמקרים האפשריים – ניקח למשל תרחיש בו יש לנו 15 שדות קלט, וכל אחד מהם יכול לקבל אחד מ5 שדות אפשריים. מספר הקומבינציות האפשרי הוא 5^{15}. כמובן, שאף אחד לא שואף אפילו להתקרב לרמה כזאת של בדיקות, גם לא בטוח שמעבר על כל זה יוביל אותנו לאיזה מסקנה בנוגע לקוד. לכן, עלינו להתמקד במקרים העיקריים ומקרי הקצה הרלוונטיים.
3. יש לבצע את הבדיקות מוקדם ככל האפשר – זה דבר הגיוני מאוד שדנו בו כבר בעבר. ככל שבאג יוכרע במקום מוקדם יותר בשלפי פיתוח התכנה השונים, כך הוא ישפיע על פחות חלקים. אם אנחנו נשנה איזה מודול בשלב מאוחר, יכול להיות שיש כבר הרבה דברים שבנויים עליו שעלולים להיהרס. בסופו של דבר, תיקון מוקדם מסייע לנו גם בפן הכלכלי וגם מהבחינה שככל שיש לנו יותר זמן לטפל בבעיות, כך התוכנה תהיה איכותית יותר.
4. איחוד תקלות – אנחנו יוצאים מנקודת הנחה, שאם יש באג בקוד, כנראה שהוא לא היחיד, והוא קרא גם לכל החברים שלו. תמיד יכול להיות שסתם יש איזה טעות קטנה, אבל בהרבה מקרים, אותו איזור של הבאג יהיה מוצף בתקלות. כך שאנחנו לא ישר צריכים לחפש את המחלקות הכי רחוקות, אלא "לחפש מתחת הפנס".
5. עיקרון ההדברה – כאשר מדבירים חרקים בקביעות, יש לשנות פעם בכמה זמן את ההרכב הכימי של ההדברה, וכן לרסס במקומות שונים. גם בבדיקות התכנה, אנחנו נשאף לשנות את הקוד בהתאם לפיתוח המוצר והדרישות השונות שמתחדשות בכל רגע.
6. בדיקות נעשות על פי הצורך – לא כל בדיקה שווה באופיה. אנחנו לא צרכים להגיד שאנחנו מכירים איך לעשות בדיקות, אלא להבין שאם עומדת בפנינו מערכת רפואית או עסקית, אנחנו מגיבים אחרת ומתייחסים אחרת לכל פן שונה בבדיקות.
7. לא קיים מצב שאין תקלות.

סוגי בדיקות תכנה עיקריים



בשביל להבין את הסוגים השונים של בדיקות התכנה, יש להבין קודם כל, את המרכיבים השונים של התכנה.

כשאנחנו מסתכלים על המסגרת הגדולה שנקראת "אפליקציה", אנחנו מבינים שבעצם אין לנו פה גוף אחד שעובד, אלא מערכת גדולה שמורכבת ממספר חלקים. כל חלק כזה נקרא "מרכיב" או Component בלעז. המרכיבים השונים מתקשרים ביניהם ויוצרים אינטראקציות שונות (מרכיב של רישום מנוי פונה לבדיקת מנוי וכו'), כאשר התוצאה הסופית היא ההתנהגות של האפליקציה. כל מרכיב בעצמו הוא מורכב. כלומר, בתוך כל קומפוננט יש לנו מספר

יחידות Unit שונות, שכל אחת מהן מביאה לנו חלק קטן – בדרך כלל ברמת המחלקה או משהו בדומה. היחידות הם החלקים הכי קטנים של התכנה.

סוגי הבדיקות יתחלקו באופן הבא, בהתאם לחלק המערכת עליו בודקים:

בדיקות קבלה (Validation) – בדיקה על האפליקציה ברמת המשתמש. לראות שהכל עובד נכון ובסדר הרצוי. כמו כן, בודקים שעמדנו בכל דרישות הלקוח.

מטרת הבדיקות היא לוודא שאין באגים ברמת המשתמש – כלומר, הכל עובד חלק. אנחנו מתייחסים לבדיקות אלו בבדיקות Black Box ולא מתייחסים בכלל לאלגוריתמיקה שמאחורי כל דבר, אלא רק לשאלה אם זה בכלל עבד.

בדיקות אינטגרציה –בבדיקות אלו בודקים את כל הרכיבים בפני עצמם. יורדים לרמת הרכיבים ובודקים שכל אחד עובד בפני עצמו, ובנוסף מסתכלים על כל האינטראקציה שהוא יוצר עם הרכיבים השונים, שאין בה שום בעיה ובאג שיוצר לנו אנומליה.

בדיקות יחידה – נכנסים ממש לכל פונקציה ופונקציה בפני עצמה ובודקים אותה שהיא עובדת בצורה נכונה. אם אי לנו באגים בכלל ברמות האלה, אנחנו יכולים לדעת שהתכנה תעבוד בצורה טובה וללא באגים.

[15]

קידוד בדיקות יחידה

חלק זה, הוא החלק הרלוונטי ביותר למפתחים עצמם. מאחר וכאן אנחנו יורדים ממש לרמת הקוד, וכל פונקציה בתוך מחלקה מכילה משמעות חשובה, אנחנו חייבים לתת את הדעת על בדיקות אלו.

כאשר אנחנו מגדירים לנו בדיקות על היחידות השונות אנחנו צריכים לשים לב לשלושה איזורים עיקריים אליהם אנחנו צריכים להתייחס –

[15] בודק תוכנה נכנס לבר. מזמין בירה. מזמין 99999999999 בירות. מזמין לטאה. מזמין 1- בירות. מזמין דגנדהנדךלהו.
לקוח אמיתי ראשון נכנס ושואל איפה השירותים. הבר עולה באש, והורג את כולם.

1. מקרי קצה – הכנסה של מספרים גבוהים מידי / נמוכים מידי. רצפים ארוכים / קצרים. אותיות במקום מספרים וכו'. כל דבר שאנחנו אומרים שהוא בכלל לא רלוונטי, כי למה שמישהו ירשום שהוא נולד בשנת 156 לספירה – אנחנו צריכים לוודא שאנחנו מוגנים בפני ליצנים.
2. המצבים הפשוטים – בדקנו שאם מישהו יכניס את המספר הכי גבוה כלום לא יקרה, אבל לא בדקנו ש1+1=2. כמובן שככה לא נגיע רחוק. וע"ע הציוץ מלמעלה.
3. הימנעו מלבצע יותר מידי בדיקות – אנחנו לא באת צריכים לבדוק את **כל** הקלט האפשרי.

יספיקו לנו מקרי הקצה ומספפר מקרים רגילים שמראים לנו מה קורה. אם נעשה את ה5[15] מהדוגמא הקודמת, זה לא אומר שאנחנו עושים בדיקות רציניות יותר.

דוגמאות לבדיקות יחידה

```
public static double div(double numerator, double denominator)
```

הפונקציה הזאת היא יחסית פשוטה- לקחת שני מספרים ולחלק ביניהם. קודם כל נבדוק את מקרי הקצה. הקלאסי שבהם, הוא חילוק ב-0. אמנם זה דבר לא חוקי, אבל בעצם אנחנו מצפים לקבל אינסוף. נעשה את הבדיקה הזאת פעמיים, פעם אחת עם חילוק של מספר חיובי ב-0, ופעם אחרת עם חילוק של מספר שלילי. התשובה בכל פעם תהיה שונה במעט – פעם זה יהיה פלוס אינסוף ופעם מינוס.

לכן אנחנו נכתוב פונקציה של טסטים שמחלקת ב-0, ונבדוק את שני המקרים האלו –

```
@Test
public void testDivInfinity()
{
    double posInf = MathEx.div(10, 0);
    assertEquals(Double.POSITIVE_INFINITY, posInf, 0.1);

    double negInf = MathEx.div(-10, 0);
    assertEquals(Double.NEGATIVE_INFINITY, negInf, 0.1);
}
```

החלוקה בסוגריים של הפונקציה Assert, מציינת את התוצאה שאנחנו מצפים שתתקבל, ובצד השני את טווח הטעות שאנחנו מקבלים. כמובן, שפה הוא חסר כל משמעות כי אנחנו רוצים לקבל אינסוף.

מה שמעניין הוא, שבמקרה רגיל, אנחנו לא נוכל לחלק באפס, ואם ננסה לעשות השמה כזאת למשתנה אנחנו לא נקבל אינסוף אלא בעיטה החוצה. אבל בגלל שהגדרנו את זה תחת פונקציית Assert אז אנחנו מסוגלים לקבל את מקרי הקצה האלה.

אחרי זה נבדוק חילוק של מספרים רגילים – נתחיל בחילוק רגיל של מספרים ללא שאריות וחלקים עשרוניים, ולאט לאט נסבך את זה קצת יותר, עם חלוקה בשלילי, וחלוקה של 10 ב-3, כך שנוכל גם להתחשב במרווח הטעות. כי התוצאה תהיה 3.33 שהיא תשובה לא מדויקת.

```java
@Test
public void testSimpleCases()
{
    double pos = MathEx.div(10, 2);
    assertEquals(5, pos, 0.1);

    double neg = MathEx.div(-10, 2);
    assertEquals(-5, neg, 0.1);

    double dbl = MathEx.div(10, 3);
    assertEquals(3.33, dbl, 0.1);
}
```

בעת כיסינו את המקרים הסבירים. עכשיו כל מה שנוסיף מעבר למקרי הקצה והמקרים הפשוטים האלה לא באמת רלוונטי. אנחנו לא צריכים לבדוק גם חלוקה של 10 ב-0 וגם חלוקה של 5. ואנחנו גם לא צריכים לבדוק את כל האפשרויות השונות של חלוקה של מספרים תקניים. אלא עוצרים פה ומבינים שעשינו נכון.

```java
public static int[] sort(int[] array)
```

מקרה שני אותו נפרט לגבי הבדיקות הוא פונקציית המיון. פונקציה שמקבלת מערך של מספרים שלמים, ואמורה להחזיר אותה באופן ממוין. כמובן שפה זה כבר שונה לגרי מפונקציה אריתמטית – מקרי הקצה לא ממש ברורים כמו בחילוק 0. לכן, ברור שנצטרך למצוא שיטה אחרת לבדוק את הפונקציה.

הבדיקות שנעשה פה מתייחסות קודם כל לבדיקה שהפלט אכן ממויין (בדיקת אוראקל), הרצה של מספר מערכים. אבל כל זה לא בחומר הנלמד! אז רק נדגיש דבר נוסף שהועלה בחלק מהשקופיות בהמשך.

אנחנו לא בודקים את כל הקוד! בדיקה של כל הקוד מההתחלה עד הסוף תהיה עסק יקר מאוד ואיטי. ולכן אנחנו מתרכזים בפונקציות עיקריות שמהוות את בסיס הפונקציונליות, ואת החלקים המורכבים יותר. כמובן שאפילו לא נחשוב לבדוק פונקציות גטרים וסטרים כי חבל על המאמץ.