

**פתרון שאלה 3:**

סעיף א:

כפי שלמדנו במהלך הסמסטר, על ידי צוות קורס, אחד ההיבטים החשובים והמשמעותיים ביותר בעבודה בתור מהנדסי תכנה לעתיד הינו ניהול גרסאות. כאשר על פרויקט ממונה צוות המכיל אפילו יותר מבן אדם אחד, יש צורך בכלי לניהול גרסאות וכמובן לניהול תצורות.

טרם פיתוח הקוד במהלך השלבים הראשוניים של הפרויקט, הרגשנו בצורך של ניהול גרסאות בצורה אופטימלית. כמו כן עם ניסיון עבר של חלקנו בתוכנות של ניהול גרסאות ידענו כי ישנו צורך בהתנסות מעשית "Hands On" עם הכלים טרם עבודה ממשית בכדי למנוע טרגדיות עתידיות.

החלטנו לחלק לעצמנו Action Item של ביצוע מחקר על מספר סוגים של כלים לניהול גרסאות, כפי שציינו במסמך הToDoList שהוגש כבר בהגשה הראשונה. במהלך המחקר גילינו כי ישנן שתי גישות מרכזיות לניהול גרסאות:

* **גישה ריכוזית** – כל המסמכים וקבצי הקוד נמצאים בתיקייה אחת Repository אשר כל אחד מחברי הצוות יכול לגשת אליה ולשנות אותה על ידי התחייבות של השינויים ((Commit.
* **גישה מבוזרת** – הקבצים נמצאים בענן, כאשר כל אחד מחברי הצוות יכול להעתיק את הקבצים ולעבוד עליהם לוקאלית ורק כאשר הוא מרוצה מהעבודה או הרגיש שהוא סיים תכולה גדולה הוא יכול לבצע דחיפה (Push) לענן.

התייעצנו והתדיינו בתוך הצוות והגענו להחלטה לפעול בגישה מבוזרת, כעת השלב היה למצוא כלי שיהיה ידידותי למשתמש שנוכל להתרגל אליו די במהירות ולהקים את התשתית של הפרויקט, מצאנו מספר כלים בשוק כמו GitHub, GitLab, BitBucket, BeansTalk, AWSCodeCommit.

לאחר סקר שוק קצר החלטנו להתקדם באמצעות GitHub כיוון שזהו כלי חינמי בעל יתרונות רבים כגון: אין צורך בהתקנה מוקדמת או יצירת סביבת עבודה, מאפשר התחברות נוחה מכל מקום פיזי שונה, וכמובן היתרון המרכזי היה שזהו אחד הכלים הכי פופולאריים כיום ולכן ישנם מסמכים ומדריכים רבים כיצד ניתן לפעול ולהשתמש בכלי זה.

כמו כן בכדי להקל על השימוש בכלי בסביבת העבודה של GitHub, החלטנו להשתמש באחד מהכלים והתוספים הקיימים בשוק כיום, לאחר מחקר קצר הגענו לשני כלים עיקריים שהם גם חינמיים וגם מאפשרים פיצ'רים נוחים לשימוש: TortiseGit, GitExtensions. לבסוף בחרנו להשתמש בTortiseGit כיוון שזהו כלי שמאפשר אינטגרציה מאוד נוחה ופשוטה עם מכשירים בעלי מערכת הפעלה של Windows אשר יש לכלל חברי הצוות (זה בעצם shell extension). כמו כן הוא מאפשר icons overlay כך שניתן לראות בקלות איזה קבצים שוני באופן מקומי מחשבים, על אילו קבצים בוצעו commit ומהם הקבצים החדשים שנוספו.

כמו כן ידענו שעל אף הזהירות ומקדמי הבטיחות שניקח יהיו מקומות בהן לא יהיה מנוס מפעילות על קטעי קוד משותפים וכמובן כצורך יוצא מכך נצטרך לראות את ההפרשים והשינויים בין קבצים בגרסאות שונות (diff) ולמזג ולאחד את הקבצים לגרסה אחת עדכנית (merge), כלומר בנוסף לכל הכלים נצטרך לכלי השוואה של קבצים. לשמחתנו TortiseGit מכיל בתוכו גם כלי השוואת קבצים פנימי והוא גם מאפשר להשתמש בכלים אחרים אשר מותקנים במחשב, ולכן בחרנו להשתמש בכלי חינמי של Beyond Compare 4.

כעת ברגע שהגענו להסכמות לגבי הכלים שבהם נשתמש התחלנו לתכנן את שיטת העבודה, ואת דרכי הפיתוח בדגש על ניהול הגרסאות והימנעות מדריסה ופגיעה בקוד של חברי הקבוצה בצורה לא רצונית.

כפי שציינו הפרויקט שלנו מנוהל בספרייה בGitHub וכל אחד מחברי הצוות יוריד עותק לוקאלי של הספרייה אליו למחשב ויעבוד על מטלותיו באופן מקומי. בעת סיום המטלות וקבלת ביטחון בעבודה (confidence) שהיא מצליחה להתקמפל ולרוץ ללא שגיאות יבוצע commit אל הספרייה. מעת לעת יש צורך לבצע בקשת משיכה (pull request) מהספרייה בGitHub בכדי לוודא שאנו עובדים על הגרסה העדכנית.

החלטנו להתנהל בשיטה של ביצוע תכן מקדים לכל קטעי קוד שנרצה לכתוב כאשר החלטנו שנקפיד על מספר כללים, זיהוי והוצאת קוד שייתכן ויהיה בו צורך במספר מוקדים שונים אל מחלקות כלליות שכל אחד יוכל לגשת אליהן. טרם מימוש בפועל של המחלקות הכלליות נגיע להסכמה לגבי פעולות שהן אמורות לספק והכנת API כך שגם אם הן אינן ממומשות נוכל להשתמש בהן. וכמובן חלוקה מודולרית של הפרויקט לתתי רכיבים וקומפוננטים כך שנקטין את הסיכון לביצוע שינויים באותם קטעי קוד.

כלומר תהליך הפיתוח שלנו כלל את השלבים הבאים:

**יתרונות השיטה:**

* צמצום התלויות של חברי צוות אחד בשני ויכולת לעבוד עצמאית והתקדמות בזמן האישי במטלות
* צמצום היקף כל קובץ, ובכך יותר קל לקרוא אותו ולהבין בצורה לוגית מה הוא אמור לשמש
* שליחת מתודות ואובייקטים למחלקות שניתן להשתמש בהן ביותר ממקום אחד ובכך לחסוך זמני פיתוח
* כמות מיזוגים וקונפליקטים מינימליים, רק כאשר יש צורך בכך

**חסרונות השיטה:**

* כל אחד עובד לוקאלית אצלו ובמחשב וישנן מתודות (בעיקר של הקבצים) אשר תלויות במיקומים של קבצים, אשר ישתנו בין מחשב למחשב.
* מסד הנתונים מתעדכן על בסיס קבוע ויש צורך בפעולה אקטיבית בכדי לטעון את הקבצים לWorkBench של כל אחד מחברי הצוות.
* חוסר היכרות עם קטעי קוד שלא כתבת, או ביצעת עליהם Code Review
* כאשר היה צורך במיזוג, כנראה שנדרש לנו יותר זמן כיוון שהיה מספר יותר מצומצם שלהם.

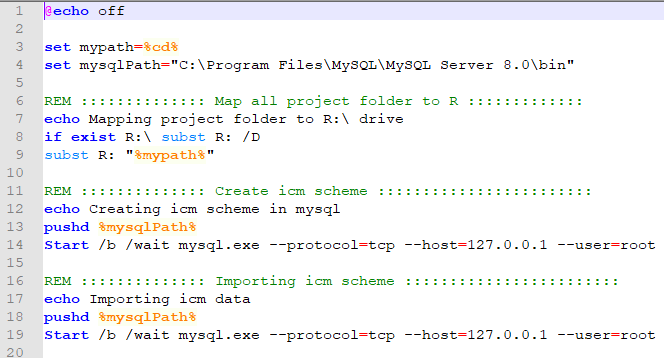
**פתרונות שלנו לחסרונות:**

* כיוון שיש צורך במיקומים זהים של הקבצים בפרויקט בהתחלה ניסינו להתחייב על סטנדרט מסוים, כל אחד מחברי הצוות יקרא לתיקיית הפרויקט באותו השם וישים אותה בתיקייה קבועה מראש בכונן C. כמובן שהיה לנו קושי לעקוף את הסטנדרט הזה ונוצרו לנו מספר בעיות של חוסר התאמה.

לכן הבנו שעלינו למצוא פתרון יותר יעיל שידרוש פחות מאמצים והתעסקות, לאחר מחקר הגענו למסקנה שהכי נוח יהיה לייצר סקריפט אשר ימפה את כל תיקיית הפרויקט לכונן לוגי R ובכך גם אם כל חבר צוות ייתן לה שם שונה או ישמור אותה במקום אחר, הרי שבכונן R הכל יראה אותו הדבר.

* החלטנו להתלבש על הפתרון הקודם ולהכניס לסקריפט גם את ייבוא הטבלאות החדשות אל הWorkbench

צילום קטעי הסקריפט הקצר:



* בכדי שנכיר את כלל הפעילות שמתרחשת בצוות החלטנו שניפגש אחת לשבוע במינימום ובמידת הצורך נגדיל את תדירות המפגשים וננצל את המפגש בעיקר לתת במה לכל אחד להציג את הפעילות שהוא הספיק לבצע, להסביר על מה הוא עובד כיום, ובמה הוא נתקע וכיצד ניתן לסייע לו במידת הצורך.
* בכדי להקל על פעולת המיזוג החלטנו למפות את הקבצים בהם אנו חושדים שיהיו מספר של אנשים שיצטרכו לשנות ולעדכן שם דברים בו זמנית. ואם הייתה אפשרות ניסינו לחלק את העבודה שלנו על קבצים אלו בכדי להימנע ממיזוג (כלומר חבר צוות אחד מודיע בקבוצה שהוא הולך לעבוד על קובץ שיתופי ובמידה ואפשר חברי צוות אחרים ימנעו מלבצע שם שינויים). ובמידה ולא הייתה ברירה אזי אנחנו מהנדסי תכנה ונוכל לעבור בעדינות ובזהירות על הגרסאות השונות ולבצע איחוד מבלי לפגוע בקוד. כמו כן במידה וביצענו טעות הרי שTortiseGit שומר היסטוריה של כל הגרסאות השונות ולכן יכלנו בכל עת לבצע חזרה לאחור (Revert) לגרסה שאנו יודעים שהיא עובדת.