CASE – Computer Aided Software Engineering

1. מבוא

הגדרה 1:
1. הנדסת תוכנה היא הביסוס והשימוש בשיטות ועקרונות הנדסאיים תקינים כדי לקבל תוכנה כלכלית שהיא אמינה ופועלת על מחשבים אמיתיים.
2. הנדסת תוכנה היא בניית מערכות תוכנה בעלות גרסאות מרובות ע"י צוות (ולא ע"י איש אחד).

הגדרה 3: CASE = Computer Aided Software Engineering הינה:
1. "נעליים לילדי הסנדלר".
2. עזרת המחשב בגישה הנדסית לפיתוח ותחזוקה של תוכנה.
3. תיעוש של טכניקות של הנדסת תוכנה כדי לשפר ולהביא לאוטומציה של תהליך הפיתוח והתחזוקה של מע' תוכנה.

1.1 כדורי כסף:
לא קיים "כדור כסף" שבאמצעותו ניתן להרוג את משבר התוכנה.
דוד הראל טוען שיש דרך להתגבר על משבר התוכנה.
ברוקס מנתח במאמרו חלק מהדברים שקיוו שיהוו כדורי כסף ומסביר למה הם לא כדורי כסף ומדוע לא יהיו כדורי כסף. הוא טוען שאין כדורי כסף, שכן כל הבעיות שדרושות לפיתוח ותחזוקה של מע' תוכנה גדולות ניתן לחלק ל-2 סוגים:
1. ***Essential Problems*** – בעיות עקרוניות.
2. ***Accidental Problems*** – בעיות מקריות.

כל אותם דברים שנחשבו ככדורי כסף פותרים את הבעיות המקריות, אולם אין הם פותרים את הבעיות העקרוניות.

Essential Problems:

I. ***Complexity:*** סיבוכיות מבחינת האלגוריתם, מבחינת ההבנה:
   א. כל דבר הוא שונה (אין שני מרכיבים זהים בתוכנה).
   ב. מס' המצבים הוא עצום (מצב = ערכים רגעיים של המשתנים במע'). מס' המצבים של מע' תוכנה גדול לאין שיעור ממס' המצבים של מע' החומרה המתאימה.
   ג. אין scale up פשוט: זה שניתן לפתור מע' תוכנה קטנה לא גורר שניתן לפתור מע' תוכנה גדולה יותר.
   ד. אין קירובים מסדר ראשון.

II. ***Conformity:*** התוכנה חייבת להתאים לעולם החיצוני:
   א. בתוכנה אין תיאוריות בסיסיות פשוטות (לעומת מדעי הטבע שבהם יש).
   ב. בענפי ההנדסה השונים יש רק גרסאות חדשות, אין שדרוג. לעומת זאת, בתוכנה צריך לשפר ולהתאים להווה – יש תחזוקה כל הזמן.

III. ***Invisibility:*** את התוכנה לא רואים ולכן יותר קשה לבנות אותה ולפקח עליה באופן אוטומטי.

Accidental Problems:

I. ***High Level Languages:*** השפות העיליות פותרות בעיה מסוימת, הן פותרות complexity מסוים, אולם אין הן פותרות את ה-complexity הבסיסי.

II. ***Interactive Work:*** עדיין לא פותר את הבעיות העיקריות.

III. ***סביבות פיתוח:*** יש התקדמות אך זה עוזר רק לתקשורת בין אדם למחשב ולא למשבר התוכנה.

תקוות לכדורי כסף שהכזיבו:
1. שפות עיליות חדישות (למשל, Ada).
2. Object Oriented.
3. שיטות פורמליות (אימות תוכניות, מפרטים פורמליים).
4. AI ומע' מומחה – תכנות אוטומטי, כלומר שהמערכת תפותח לפי הדרישות שינתנו.
5. CASE.

דוד הראל מסביר למה, לדעתו, יש בכל זאת תקווה. הגישה של בניית מודלים בצורה טובה ובאמצעות כלים שיש להם סמנטיקה חד-משמעית הם התקווה לעתיד.

1.2 מודלים למחזור חיים ותהליכי תוכנה:

1.3 CMM וסטנדרטים אחרים: CMM – באיזו מידה חברה יכולה לפתח תוכנה.

2. מיון של כלי CASE:

הגדרה 2:
1. תהליך (process) הוא קבוצה של צעדים בסדר חלקי המיועדים להשיג מטרה מסוימת.
2. מודל של התהליך הוא הצגה כלשהי של התהליך (למשל, רשתות פטרי).
3. תהליך תוכנה הוא סדרה של צעדים (משימות) הנדרשים כדי לפתח או לתחזק תוכנה: זוהי סדרה של פעולות בקשר למוצר תוכנה מהרגע שהצורך במוצר מתעורר עד לרגע פרישת המוצר (ייתכן שזה קורה עוד לפני ההתקנה).
4. הגדרת תהליך תוכנה היא הגדרה פורמלית של תהליך התוכנה. דיגום התהליך (Process Modeling) הוא יצירת מודלים של תהליכים והשימוש בתהליכים כאלה.
5. תהליך של הנדסת מוצר הוא אחד מהתהליכים הבאים: הנדסת דרישות, עיצוב, תכנות, אינטגרציה, אימות ונכונות, תחזוקה, ניהול מוצר, ניהול פרוייקט, אבטחת איכות, ניהול נתוני פרוייקט.
6. תהליך של הנדסת תהליכים (מטה-תהליכים) הוא אחד מהתהליכים הבאים: שיפור התהליך של הנדסת מוצר, דיגום ותכנון של תהליך, מדידת תהליך, שימוש חוזר של תהליך.

תהליך התוכנה מתחלק ל:
1. תהליך מוצר – product process.
2. תהליך להנדסת תהליכים – meta process.

(א:9) כל אחד מ-2 התהליכים הנ"ל נתמך ע"י תמיכה של תהליך המוצר ותמיכה לתהליך הנדסת תהליכים. מימוש התמיכה נעשה ע"י טכנולוגיית CASE: כלי CASE שתומכים בתהליך המוצר וכלי CASE שתומכים בתהליך של הנדסת תהליכים.

הגדרה 4:
1. כלי תוכנה (CASE Tool) הוא עזר תוכנה כדי לבצע <u>פעילות מסוימת</u> בתהליך התוכנה (למשל: מהדר, UML).
2. מטה-כלי (Meta-CASE tool) הוא כלי ליצירת כלי תוכנה. הוא חלק מהטכנולוגיה של תהליך הנדסת התהליך (מטה-תהליך) (למשל: yacc; הכלי שמאפשר לצייר UML)

<u>מיון כלי CASE:</u>

<u>1. Editing:</u>
- Graphical Editor - עוזר לצייר UML ו-DFD לפי הכללים.
- Textual Editor: משמש לכתיבת תוכניות ולתיעוד. יש editor רב תכליתי, רב שפות (emacs), רב-לשוני. בודק synatx.

<u>2. Programming:</u>
- Coding and Debugging:
  - Assemblers.
  - Compilers.
  - Cross Compilers: קימפול תוכנית על מחשב א' שתרוץ על מחשב ב'. זה שימושי, למשל, עבור micro processors.
  - Debuggers.
  - Interpreters
  - Linkage Editors: לוקח רכיבים/מחלקות שונים שיושבים בשפת מכונה ועושה מהם תוכנית אחת (יכול להיות רב-לשוני או חד-לשוני).
  - Precompilers/Preprocessors: מחליף string של #include, למשל, ל-string שהיא מייצגת - הקובץ.
- Code Generators: נותנים עיצוב ל-code generator ומהעיצוב הוא יוצר קוד.
  - Compiler Generators: למשל, yacc (מקבל הגדרת EBNF).
  - Code Restructurers: נותן מבנה חדש של קוד נתון. בד"כ הכלי הוא semi-automated.

<u>3. Verification and Validation:</u>
- Static Analyzers: מופעל על הקוד כמו שהוא (בלי הרצה), למשל: ה-static anayzer יגלה משתנה שלא מקבל ערך.
  - Cross-reference Generators: יודע את כל המקומות בהם מופיע x ואיפה יש קריאות לפונקציה מסוימת.
  - Flowcharters.
  - Standards Enforcers: בודק האם הקוד הוא בהתאם לתנאים
  - Syntax Checkers.
- Dynamic Analyzers:
  - Program Instrumentors: תוכנית ששוכנת מעל התוכנית שלי ומשלבת קוד בתוך התוכנית שלי, עוזר ל-debugging.
  - Tracers/Profilers: בודק מי קורא למי וכמה פעמים.
- Comparators: טוב ל-testing. מריצה את התוכנית עם נתוני קלט ובונה

קובץ פלט ומשווה בינו לבין התוצאות הרצויות.
- Symbolic Executors: כאילו מריץ תוכנית עם שמות של משתנים ולא עם ערכים קונקרטיים. עוזר למציאת פקודות שלא מגיעים אליהן.
- Emulators/Simulators.
- Correctness Proof Assistants: assert(). בדיקה דינמית של pre- ו-post-conditions. אימות תוכניות.
- Test-case Generators: בונה ערכים לבדיקת התוכנית.
- Test-Management Tools: מעקב אחרי בדיקות רגרסיביות.

4. Configuration Management:
- Configuration and version-management tools.
- Configuration builders.
- Change-Control monitors.
- Librarians.

5. Metrics and Measurement:
- Code Analyzers.
- Execution Monitors / Timing Analyzers.

6. Project Management:
- Cost-estimation Tools: מודלים להערכה, כמו COCOMO.
- Project-planning Tools.
- Conference Desks.
- Email.
- Bulletin Boards.
- Project Agendas.
- Project Notebooks.

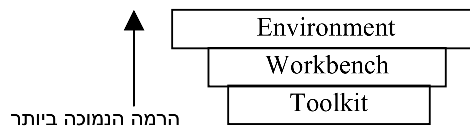
7. Miscellaneous Tools:
- Hypertext systems.
- Spreadsheets.

כל הכלים לעיל לא כוללים את מה שקשור למנשק.

הגדרה 5: [חלוקה של מחזור החיים הקלאסי לשניים]
1. כלים אנכיים הם כלים המשמשים <u>בשלב מסוים</u> של מחזור החיים. כלים אופקיים הם כלים המשמשים <u>לאורך מספר שלבים</u> של מחזור החיים של התוכנה.
2. כלים מסוג: “upstream”, “front end”, “Upper CASE” הם כלים המשמשים לפעילויות הניתוח והעיצוב של תוכנה. כלים מסוג: "down stream", "back end", “Lower CASE" הם כלים המשמשים לקידוד, בדיקת תוכנה, הנדסה מחדש וכדו'.

הגדרה 6: Workbench הינו אוסף של כלי תוכנה התומכים בפעילות אחת או מספר קטן של פעילויות (לא חייב להיות קשר בין הכלים). [workbench מכיל כלים של אותו שלב במחזור החיים]

הגדרה 7:
1. סביבה (Environment) היא מסגרת המארגנת אוסף של כלי תוכנה ו-workbenches. היא מבקרת הפעלת הכלים ותומכת באינטראקציה של הכלים לאורך חלק ניכר של תהליך התוכנה.
2. Toolkit הוא סביבה בעלת אינטגרציה מזערית בלבד. [toolkit יכול להכיל כמה workbenches]



3. ארכיטקטורות של מערכות תוכנה:

הגדרה 8:
1. ארכיטקטורת תוכנה היא אוסף של רכיבים חישוביים יחד עם תיאור האינטראקציות (הקשרים) בין הרכיבים.
2. ארכיטקטורת pipe-and-filter: הרכיבים (filters) מפעילים טרנספורמציות מקומיות על הקלט שלהם. החישוב מתבצע סדרית, כך שתהליך הפלט יכול להתחיל לפני סיום תהליך הקלט. הקשרים (pipes) מהווים צינורות להעברת המידע: הפלט של רכיב אחד מהווה קלט של הרכיב שאחריו.
3. מערכת שכבתית (layered system) היא מערכת שמאורגנת בשכבות כך שכל שכבה מספקת שירותים לשכבה שמעליה ומהווה לקוח של השכבה שמתחתיה. לעתים השכבות אטומות, לעתים הן שקופות.
4. repository הוא מערכת תוכנה שבנויה מ-2 חלקים:
   א. מאגר נתונים מרכזי (המייצג את המצב הנוכחי של המערכת).
   ב. אוסף של רכיבים בלתי תלויים הפועלים על מאגר הנתונים המרכזי.

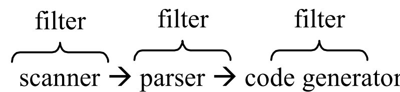
במסדי נתונים קלאסיים הקלט קובע את בחירת התהליכים אשר על המערכת לבצע.
בארכיטקטורת blackboard המצב הנוכחי של מאגר הנתונים המרכזי הוא שבוחר את התהליכים אשר יש לבצע.

5. מערכת לשיתוף מידע היא מערכת תוכנה שאוספת, מעבדת ושומרת על פריטי מידע בעלי מבנה מגוון, לעתים מסובך, ובד"כ בכמויות גדולות.

הגדרה 10: המסד של סביבת CASE מאחסן:
1. כל המידע על המערכת, כגון: תיאורי תהליכים, מפרטי המודולים, תיכון המאגרים, לוגיקת הבקרה.
2. מידע-על, כגון: תקנים של תיעוד, מפרטים, הגדרת תהליכים, ניהול תצורה.

כל filter הוא יחידה בלתי תלויה ב-filters האחרים.
filter מסוים לא חייב לדעת מי הם ה-filters האחרים.
הסדר של ה-filters יכול להשתנות והתוצאה תהיה זהה.
אימות: זה שהיחידות בלתי-תלויות מפשט את אימות התוכנית.
הארכיטקטורה הזו מעודדת reuse – מסתמך על זה שה-filters הם בלתי-תלויים.
reuse: 1. לקיחת רכיב ולהשתמש בו במקום אחר.
2. את ה-filter המסוים הזה ניתן לכתוב בכמה גרסאות שכולן מקבלות אותו קלט ומוציאות אותו פלט
← ניתן לבחור איזו גרסה שרוצים.

דוגמה:

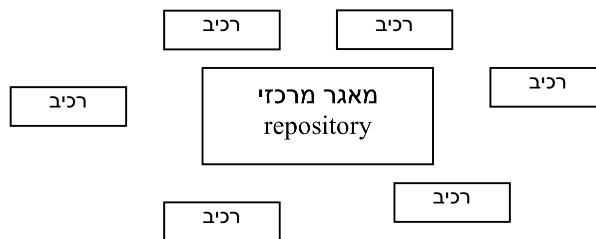


ה-pipes הם הפלט בכל שלב.

חסרונות:
1. הארכיטקטורה הזו לא טובה עבור יישומים אינטראקטיביים.
2. זה שפלט אחד הוא קלט של השני גורם לכך שהמעברים הם ברמה פשטנית מאוד - אחידה. אם רוצים לעשות ב-filter דברים יותר משוכללים, צריך להיות ב-filter משהו שיודע להוציא מידע ולעשות מניפולציות על הקלט הפשוט שלו ובסוף לקודד מחדש ולרדת לרמה הפשטנית יותר.

דוגמה למע' שכבתית: השכבות של הפרוטוקולים.
חסרון: - בניגוד ל-CASE, אין סטנדרדיזציה של השכבות.
- ייתכן שהמע' היא שכבתית מבחינה לוגית, אולם קשה לממש אותה כשכבתית מבחינה טכנית.

repository:



דוגמאות לשיתוף מידע:
1. Information System: מכילה את כל הלקוחות עם פרטיהם וההזמנות, הנהלת חשבונות וכו'.
2. מע' כלי CASE: השיתוף הוא בין הכלים השונים. ניתן לראות את ההתפתחות של מע' כאלו דרך הארכיטקטורות השונות.

אינטראקציה ניתנת לביצוע ב-repository.

4. ניהול תצורה וגרסאות:

4.1 מבוא:
מהן הבעיות?
1. ***ניהול תצורה:*** מע' תוכנה מורכבת מרכיבים שונים. צריך לדעת ממה בונים את ה-exe.
2. ***ניהול גרסאות:*** איפה יושב ומה יושב, baseline.
3. ***ניהול שינויים:*** צריכה להיות פרוצדורה מוגדרת שבה בוחנים הצעת שינוי ועליה מחליטים כן/לא עם נימוק למה ולשמור את ההחלטות והנימוקים.

הדברים הנ"ל חייבים להיעשות בצורה אוטומטית ע"י מחשב.

כבר ברמה 2 של ה-CMM (repeated) דורשים ניהול תצורה.

הגדרה 9:

1. ניהול תצורה (SCM) הוא התהליך של:
   א. זיהוי והגדרת פריטים במערכת.
   ב. בקרת שינויים של פריטים אלה.
   ג. רישום ודיווח של הסטטוס של הפריטים ובקשות שינוי של הפריטים.
   ד. אימות השלמות והנכונות של הפריטים.

   פריטים אלה נקראים פריטי תצורה (configuration item) והם נחשבים ליחידות בסיסיות שאינן ניתנות לחלוקה.
2. Revision של מודול היא גרסה חדשה שמטרתה להחליף גרסה ישנה יותר.
3. Variation של מודול היא גרסה אחרת שקיימת בו-זמנית עם הגרסה המקורית כאפשרות שוות ערך.
4. גרסה (version) היא או revision או variation.
5. Baseline של מערכת תוכנה הוא אוסף כל פריטי התצורה של המערכת שנבדקו באופן פורמלי וקיבלו אישור ע"י הגורמים המוסמכים. ניתן לשנות את ה-baseline אך ורק ע"י בקשת שינויים פורמלית ואישור הבקשה ע"י הגורמים המוסמכים.
6. תצורה של מערכת תוכנה (system configuration) היא רשימת המודולים מהם מורכבת המערכת.
7. Derivation של מערכת תוכנה הוא רישום מדויק של איזה גרסאות של המודולים האינדיבידואליים שנבחרו, איזה מהדר ו-linker היה בשימוש בבניית ה-exec, ואיזה פרמטרים וארגומנטים נבחרו עבור כלים אלה.
8. Release של מערכת הוא גרסה המופצת למשתמשים. הוא בד"כ מכיל תיעוד אלקטרוני ומודפס, תוכנית התקנה, קבצי נתונים, ותיעוד התקנה המתייחס לתצורה.
9. גרף תלות המודולים של מערכת תוכנה הוא גרף מכוון ללא מעגלים שמוגדר כדלהלן: קיימת צלע מכוונת מקודקוד א' (המייצג מודול מסויים) לקודקוד ב' (המייצג מודול אחר) אם"ם קומפילציה (רה-קומפילציה) של א' מחייבת קומפילציה (רה-קומפילציה) של ב'.

4.2 תכנון והקמה של מערכת SCM (Software Configuration Management):

על מה צריך לדון ולקבל החלטות?

1. מהם הפרטים שיש לנהל עבורם תצורה? למשל, מחלקות (class), GUI, תיעוד, נתוני בדיקה, שאילתות SQL.
2. מי אחראי על ניהול התצורה?
3. קביעת נהלים בנדון.
4. מה המדיניות לגבי SCM של ספקים חיצוניים?
5. החלטה על הגדרת כלים ל-SCM.

4.3 ניהול שינויים:

יש מע' שקיימת ורצה. רוצים לעשות בה שינויים. מס' דברים שחייבים להביא בחשבון:

1. הגשת בקשת שינוי (בד"כ טופס מוכן).
2. ניתוח של הבקשה; מדוע? מה ההשפעה על המע' ועל הארגון? וכו'.
3. הערכה של אופן ביצוע השינוי.
4. הערכה של מחיר השינוי והזמן הנדרש.
5. רישום במאגר ה-SCM של הבקשה עם כל המידע שנאסף בסעיפים 2, 3, 4.
6. החלטה של ועדת השינויים: קבלת ההצעה/דחייה/השהייה.
7. רישום של ההחלטה עם ההנמקה במאגר ה-SCM.
8. אם ההחלטה על השינוי הייתה חיובית – ביצוע השינוי עם קישור של התיעוד עם מאגר ה-SCM.
9. לאחר ביצוע השינוי – טיפול בהתאם לפרוצדורות רגילות ע"י צוות QA וצוות SCM.

clear quest – תוכנה לניהול שינויים. clear case – תוכנה להגדרת תהליך לניהול תצורה.

4.4 ניהול אחסון ורסיות:

1. לשמור ורסיות שונות בקבצים שונים (a1.0, a1.1, …, a2.0, …) (10:א)

   חסרונות: 1. כל ורסיה חדשה תופסת מקום.
   2. השמות שונים ← לא ניתן לעשות סריקה אוטומטית בשביל לעבור על כולם.
   3. התגלה באג שנמצא בכל הורסיות. כל ורסיה היא קובץ נפרד ← צריך מישהו שיעבור על כל הורסיות ויעשה את התיקון.
2. לשמור על ורסיה אחת בלבד ולהציג כל ורסיה אחרת ע"י דלתות (δ), ז"א שומרים בנפרד את השינויים הנדרשים מהורסיה המלאה (10:ב).

   יתרון: 1. תופס פחות מקום.
   2. לפעמים זה טוב יותר עבור תיקון בגים (בהנחה שהדלתות לא נוגעות בקטע הזה).
   3. טוב בשביל revisions, לא עבור variations.

   חסרון: 1. כשיש הרבה התפצלויות בין גרסאות זה מסתבך.
   2. קשה למצוא מה בדיוק השינוי (יש כלים אוטומטים שעוזרים).

3. קומפילציה מותנית: state-oriented storage.
4. change-oriented storage: בזה משתמשים כיום. שומרים גרסה אחת שמכילה הכל: בכל מקום שיש שינוי עושים פיצול ע"י קריאה ל-preprocessor שמשלב קטע מסוים או שילוב דברים לוגיים.
   חסרון: עלול ליצור גרסאות שאין להן הצדקת קיום.

דוגמה: יש 12 ורסיות שונות: 3 סוגי יחידות קלט, 4 סוגי יחידות פלט. לשמור גירסה אחת ולשלב עם preprocessor.

ב-state-oriented storage השינויים מבוססים על קבצים פיסיים (שמירת כל ורסיה – קבצים או דלתות).
ב-change-oriented storage השינויים מבוססים על לוגיקה – רמה בתוך הקוד.

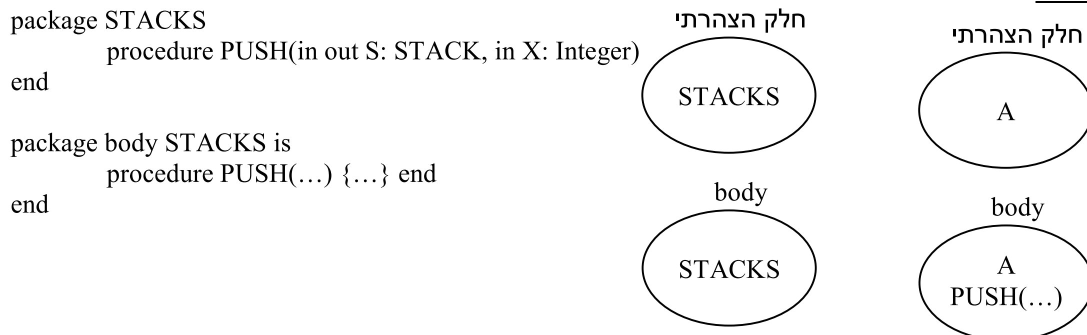
11 נושאים שהנדסת תוכנה השפיעה בהם:
1. peer review - review and inspections.
2. info hiding.
3. daily build – incremental development.
4. user involvement.
5. automated revision control.
6. open source - development using internet.
7. cobol and fortran – programming language hall of fame (לפניהם תכנתו באסמבלר).
8. CMM.
9. OO programming.
10. visual basic - component based development.
11. מטריקות ומידות.

4.5 מסד התצורות:
במסד התצורות יושב כל המידע בנוגע לתצורות. צריך לענות על:
1. איזה לקוחות קיבלו גרסה x (קונפיגורציה) של המערכת?
2. כמה תצורות קיימות? מתי נוצרו ומה ההבדל ביניהן?
3. אילו תצורות מושפעות משינוי גרסה של מודול מסוים?
4. כמה בקשות שינוי קיימות לתצורה מסוימת?
5. אילו באגים התגלו בתצורה מסוימת?
6. איזה מערכת הפעלה ואילו חומרים נדרשים כדי להריץ תצורה מסוימת?
7. איך יש לבנות תצורה כדי למלא דרישות של לקוח מסוים?

(ד:10) ***make:*** עושה קומפילציה ו-linkage להרבה קבצים. ה-make יודע בעצמו אם יש קבצים שהשתנו ולכן צריך לקמפל אותם מחדש (לפי התאריך).

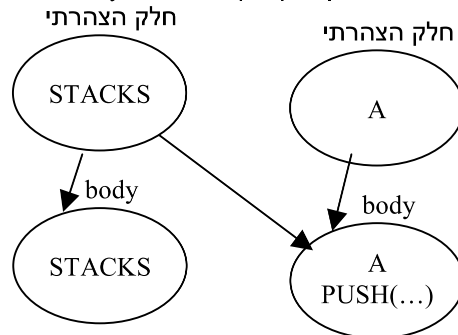
חסרונות של make:
1. קשה ליצור ולהבין make file למערכות גדולות ולכן זה נעשה בהן ידנית.
2. make לא מבחין בין קבצים שנוצרו עם אופציות שונות של מהדר.
3. make קשור לקבצים פיסיים ולא לוגיים.
4. make אינו קשור לכלי לניהול גרסאות.

ב-ADA:



איך מקמפלים? יש **גרף תלות**: ה-body של A תלוי ב-STACKS.
העיקרון: בקומפילציה של הקריאה ב-A מסתכלים על המס' והקריאה ומשווים עם מה שכתוב בחלק ההצהרתי.
מניחים שמה שכתוב בחלק ההצהרתי הוא אמת. ולכן צריך לקמפל את ה-body אחרי ה-header.

סדר הקומפילציה:
אסור שיהיה מסלול סגור בגרף.



הפתרון הזה קשור לקבצים לוגיים ← פותר חסרון של make.

מה עם re-compilation?

- אם עושים שינוי ל-body של A ← ה-body של A צריך לעשות רה-קומפילציה לפי הגרף. כיוון שאף אחד אחר לא תלוי ב-body של A, רק ה-body של A צריך לעבור רה-קומפילציה.
- אם עשיתי שינוי בחלק ההצהרתי של A ← צריך לעשות רה-קומפילציה לחלק ההצהרתי של A, ולפי גרף התלות עושים אוטומטית גם רה-קומפילציה ל-body של A.
- אם עושים שינוי ל-push ב-body של STACKS ← רק ה-body של STACKS עובר רה-קומפילציה.
- אם שיניתי משהו ב-header של STACKS ← צריך לעשות רה-קומפילציה גם ל-body של STACKS וגם ל-body של A.

4.6 כלי CASE עבור SCM:

תוכנות: Clear Case (1992) → DSEE (1987-1991) → CVS → RCS (1985) → SCSS (1975). כל התוכנות מכילות את make.
SCSS: שמירה על ורסיות בהתבסס על המערכת הראשונה.
RCS: מיזוג ורסיות שונות בקטעים משותפים (semi-automated).

DSEE (ב:10) מע' מבוזרת:

- צרכנים שונים יכולים לעבוד בו-זמנית ולהשתמש בה.
- צרכנים שונים יכולים בו זמנית ליצור תצורות שונות של אותה מערכת.
- מאפשרת קומפילציה מקבילית ← חוסך זמן.
- שומרת גם על ה-object pool וגם על source code versions, כלומר: כשצריך לעשות קמפול היא בודקת אם צריך לקמפל מחדש או שניתן לקחת משהו שנמצא כבר ב-object pool.

מיזוגים: נניח שיש מע' תוכנה:

```
main    f1.c    f2.c    …    f10.c    version 1
                         .
                         .
                         .
main    f1.c    f2.c    …    f10.c    version m
```

כשהגענו לורסיה m יוצרים project A שמשתמשים בו ב-f1.c, f2.c ומפתחים עוד ורסיות עבור הפרוייקט הזה:

```
project A:    f1.c    f2.c    m
              f1.c    f2.c    m + 1
              f1.c    f2.c    m + 2
```

מגלים כעת bug וצריך לעשות bug fix:

```
f2.c    f3.c    m
f2.c    f3.c    m + 1
```

בינתיים המע' ממשיכה ומגיעים לורסיה n. כלומר: במע' המקורית אנחנו בורסיה n, בפרוייקט A אנחנו בורסיה m + 2 וב-bug fix אנחנו בורסיה m + 1.
כעת, הפרוייקט וה-bug fix נגמרו וצריך למזג את הכל חזרה:
לגבי main, f4.c, …, f10.c אין בעיה. הבעיה היא ב-f1, f2, f3. clear case עוזר לפתור את הבעיה ולמזג את הקבצים.
מה עושה clear case? דבר ראשון ממזגים את ה-bug fix למע' עצמה (גרסה n) [מיזוג ← צריך לעשות testing]. כעת עושים מיזוג של המע' (גרסה n) עם גרסה m + 2 בפרוייקט (לפי ההחלטה שרוצים לשמור על הגרסאות של הפרוייקט). מהפרוייקט ממזגים חזרה למע'.

5. מחזור החיים של CASE

5.1 מחזור חיים של כלי CASE:

(12:א) *איזה כלי CASE קונים?*

- צריך לקבוע את תהליך התוכנה ולפי זה לבחור את כלי ה-CASE.
- צריך להתחשב בסטנדרט של הארגון.
- צריך להביא בחשבון איזו חומרה יש וחומרה עתידית.
- צריך להביא בחשבון את ההוצאות.
- צריך לבדוק כמה הכלי אמין ותחום היישום שלו.

התאמת המע':
- התקנה.
- לקחת את הגדרת התהליך ולשלב בו את הכלי.
- אינטגרציה עם כלים אחרים.
- תיעוד.
- introduction: הכנסת הכלי לפעולה בצוות.
- הטמעת הכלי.
- operation: שימוש בכלי.
- אבולוציה: יוצאת ורסיה חדשה ורוצים לשכלל את הכלי – פיתוח או קניית כלי חדש?
- obsolescence: הכלי מיושן כבר ← זורקים אותו. כדאי לעשות חפיפה בינו לבין הכלי החדש.

5.2 CMM:

Initial: תוהו ובוהו.

Repeatable:
- Software Configuration Management: לא ניתן להיעשות בלי כלי CASE
- Software Quality Assurance
- Software Subcontract Management
- Software Project Tracking and Oversight
- Software Project Planning
- Requirements Management: יש ניהול ורסיות גם על הדרישות.

Defined:
- Peer Reviews: בלתי פורמלי.
- Intergroup Coordination
- Software Product Engineering
- Integrated Software Management
- Training Program
- Organization Process Definition
- Organization Process Focus

Managed:
- Software Quality Management
- Quantitative Process Management

Optimizing:
- Process Change Management
- Technology Change Management
- Defect Prevention

איך משלבים את הכלים? צריך להבחין בין שילוב כלי CASE בודדים לבין שילוב מע' כלי CASE. קודם כל צריך להגדיר את תהליך הנדסת התוכנה ורק אז ניתן לשלב כלי CASE (כלומר צריך להיות ברמה 2 ב-CMM).

(12:ג) התהליך לא פועל טוב כי:
- אין שום ***אחידות***: ועדת בדיקה אחת תבצע הערכה שונה מועדה אחרת.
- לא כל האנשים שיושבים בועדות מוכשרים מספיק לבצע את ההערכות.
- התהליך נמשך כשבוע – לא מספיק זמן.
- זה נהפך למעין תהליך אוטומטי - לא מופעל שיקול דעת.
- החברה מספקת חומר לא רלוונטי מפרוייקט אחר.
- החברה מכינה את העובדים לראיונות.

10 הנקודות החשובות ביותר להקטנת פגמים בתוכנה:

1. גילוי ותיקון שגיאה עולים פי 5-100 אחרי מסירת התוכנה (לעומת הגילוי בשלבים הראשונים – בשלבי הדרישות והמפרט).
2. בפרוייקטים מקדישים 40%-50% מהמשאבים בתיקון שגיאות שאפשר היה להימנע מהן.
3. בערך 80% מהשגיאות שאפשר למנוע אותן באות מ-20% מהדפקטים (מודול שאינו בסדר).
4. בערך 80% מכל השגיאות באות מ-20% מהמודולים, ובערך 50% מהמודולים בכלל ללא שגיאות.

5. בערך 90% מה-downtime (הזמן שהמע' לא פועלת – הזמן שהיא נפלה) נובע מ-10% מהדפקטים.
6. peer-reviews מגלים 60% מהדפקטים.
7. review מכוון נושאים מגלה 35% יותר דפקטים מ-review לא מכוון.
8. תהליך ממושמע (כלומר, יש תהליך תוכנה מוגדר) מקטין הכנסת דפקטים בעד 75%.
9. זה עולה 50% יותר לפתח תוכנה בעלת אמינות גבוהה מאשר לפתח תוכנה בעלת אמינות נמוכה (אך יש להתחשב במחיר התחזוקה).
10. בערך ב-40%-50% מכל האפליקציות יש דפקטים לא טריוויאליים.

6. אינגרציה של כלים

הגדרה 11:
1. אינטגרציה של פלטפורמה פירושה שהכלים מתפקדים על אותן פלטפורמות של חומרה ו/או מערכות הפעלה.
2. אינטגרציה של נתונים פירושה שכל המידע של הסביבה מנוהל כיחידה שלמה.
3. אינטגרציה של בקרה מאפשרת צירוף גמיש ושילוב אוטומטי של הכלים בסביבה.
4. אינטגרציה של הצגה פירושה קיומן של פעולות של הכלים המופעלות בצורה זהה ומביאה להורדת העומס הקוגניטיבי.
5. אינטגרציה של התהליך פירושה כי הכלים פועלים ביחד בצורה יעילה כדי לתמוך בתהליך התוכנה המוגדר.

6.1 אינטגרציה של כלים (פלטפורמה):

פלטפורמה: המחשב, הארכיטקטורה ומע' ההפעלה שבהם יהיה הכלי.
אינטגרציה של פלטפורמה לא קיימת. אין כלי שניתן שיהיה בכל מחשב, בכל מע' הפעלה.

6.2 אינטגרציה של נתונים:

יש אוסף של כלים, כאשר הפלט של כלי אחד יכול לשמש ישירות כקלט לכלי השני. יש כמה דרגות:
1. ***גרעון גס:*** קבצים משותפים (למשל, Unix). (13:א)
2. מבני נתונים משותפים (13:ב).
3. הכי מפורט: ***repository משותף*** (13:ג): כל כלי יכול לקבל ולהחזיר נתונים ללא המרות.

OMS = התוכנה של ה-repository. מסוגלת לנהל את האובייקטים שיושבים במאגר של ה-repository.

הגדרה 12: מערכת ניהול אובייקטים (OMS – Object Management System) היא מסד נתונים המאפשר הגדרת ישויות מסוגים שונים, לצרף תכונות לאובייקטים, ולהגדיר יחסים ביניהם.

6.3 אינטגרציה של הצגה:

1. ***אינטגרציה של חלונאות*** – קיים.
2. ***אינטגרציה של פקודות:*** כמו cut & paste (אותו דבר בכל הפלטפורמות). ב-Unix לכל פקודה יש syntax שונה, יש פקודות שמהשם שלהן לא ידוע מה הן עושות ← אין אינטגרציה.
3. ***אינטגרציה של אינטראקציה:*** לכל הכלים השונים יש אותם תפריטים שבתוכם זה גם אותו דבר.

6.4 אינטגרציה של בקרה: (13:ד) הרעיון: שכלי אחד יפעיל כלי אחר. ייתכן message server שמעביר ביניהם את הבקשות.

6.5 אינטגרציה של תהליך: (13:ה)
ב-CMM משלב מסוים יש צוות שמנהל התהליך. מגדירים מודל של תהליך ולפי זה הכלי מודיע, מחלק הוראות ומשתף פעולה עם ה-case tools, שומר על ה-baseline.

מרחב אינטגרציה: איזה כלי לקנות: צריך לשקול את רמת האינטגרציה בין הכלים שרוצים לקנות לכלים שקיימים. בונים 5 צירים: פלטפורמה, נתונים, הצגה, בקרה ותהליך.

7. Workbenches

הגדרה 3: Workbench פתוח הוא:
1. אוסף של כלים המספק אינטגרציה של בקרה [כלי אחד מפעיל כלי אחר], או מאפשר תכנות ע"י המשתמש של אינטגרציה כזו.
2. הפרוטוקול של אינטגרציית נתונים הינו גלוי למשתמש.

ה-Workbench הפתוח המפורסם ביותר הוא Unix.

צימוד חזק: יש אפשרות לקבל מידע מפורט על ההצגה של הכלים של יצרן אחד.
Open Representation: פרסום ההצגה כך שאפשר לפתח כלים מכמה מקורות. לפעמים כלים כאלה יכולים לטפל בנתונים, אבל הם לא יכולים לשנות את ההצגה של הנתונים לצרכיהם שלהם.
Conversion Boxes: מספק filter שמייבא ומייצא נתונים בהצגות זרות (מכלי אחד לכלי אחר). כלים אלו בד"כ מאבדים את האפשרות להשתמש ב-repository באופן incremental.
No Contact: כלי שלא מסוגל בכלל להתייחס ל-repository

7.1 תכנון ומודלים של עסקים: (14) בדיקה האם המערכת מתנהגת לפי המפרט ולפי העולם האמיתי.

7.2 ניתוח ועיצוב:
ניתוח: איסוף הדרישות וניתוח המפרטים ← עיצוב.
ה-workbench דואג שכולם ידברו דרך המאגר ← אינטגרציה בין הכלים.
רשימת workbenches לניתוח ולעיצוב:
1. עורך טקסט ודיאגרמות – יצירת DFD וכיו"ב. המידע יושב כישות במאגר.
2. מנהל data dictionary – ריכוז הגדרת המושגים.
3. ניהול שאילתות ל-repository.
4. יכולת ליצירת תיעוד.
5. כלי להגדרת reports, forms.
6. כלים לייצוא ולייבוא מידע.
7. code generator – יכול לבנות את ה-header של הפונקציות, לא את הקוד עצמו, כי העיצוב לא יורד לגרעון מספיק עדין.

דרישות לגבי ה-workbenches של analysis and design:
1. א. יכולת לצייר דיאגרמות בשפה הנידונה (UML).
   ב. יכולת לגלות ולהודיע על שגיאות תחביר בדיאגרמות (ואולי גם שגיאות סמנטיות).
   ג. יכולת לגלות סתירות וחוסר קונסיסטנטיות בין דיאגרמות שונות (ואולי גם חוסר שלמות).
   ד. תמיכה בהופעה אסתטית ופשוטה של הדיאגרמות.
   ה. יכולת לייצר תיעוד באופן אוטומטי.
2. תמיכה ב-repository שיכיל את כל המידע הקשור לכל סוגי הדיאגרמות.
3. יכולת לניווט (יכולת לעבור בקלות מדיאגרמה אחת לאחרת).
4. תמיכה במס' משתמשים בו-זמנית.
5. code generation עם אינדיקציות לחלקים שנוצרו אוטומטית – צריך אינדיקציה מה נוצר אוטומטית ומה נוצר ידנית.
6. יכולת ל-reverse engineering.
7. אינטגרציה עם כלים אחרים – המטרה: שיהיה סטנדרט בין הייצוג הפנימי ← יעשו כלים טובים וזולים, אולם היצרנים מתנגדים לכך, שכן הם רוצים בלעדיות, ולכן רוב ה-workbenches הם סגורים.

7.3 GUI:
1. כלים שמאפשרים ליצור את המנשק – כלים אלו הם **<u>סגורים</u>**.
2. Visual C / Visual Basic – משתמשים באבני בנייה. השיקול הוא לפי כמה זמן צריך את המנשק וכמה זמן הכלי יתקיים (למשל: Visual Basic ימשיך להיות עוד 20 שנה לפחות).

7.4 תכנות: (14:ב), (14:ג), (15:א).

7.5 Verification and Validation: (15:ג), (15:ד)
oracle: - גרסאות קודמות.
- prototype.

7.6 תחזוקה והנדסה הפוכה:

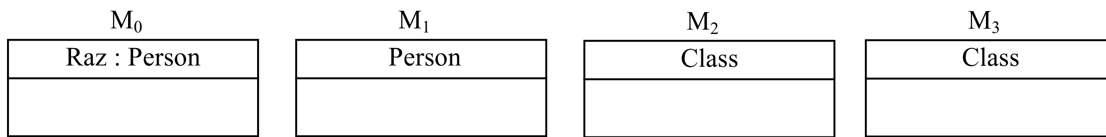
7.7 SCM: ראה פרק 4.

7.8 ניהול פרוייקטים: יש תוכנות לניהול פרוייקטים, למשל: PERT.

7.9 כלי Meta:
meta-case זה כלי שבונה כלים.

יש 4 רמות:
M_0 – data: instance, מופעים קונקרטיים.
M_1 – model: זה ה-UML בעצמו, מחלקות קונקרטיות.

M_2 – meta-model (UML meta-model): הסתכלות מלמעלה על ה-UML. מגדירים מה זה class.
M_3 – meta-meta-model: הרמה העליונה.



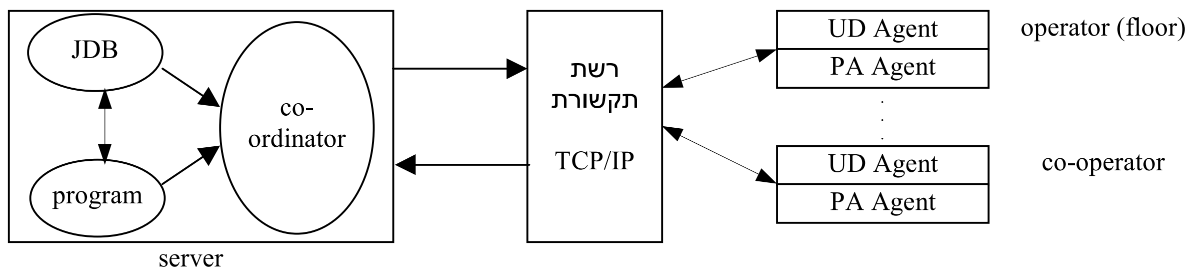
בכל הרמות התחביר מוגדר היטב, אולם הסמנטיקה אינה מוגדרת – היא בשפה טבעית.
אם יש מטה-כלי אזי ניתן "להטעין" עליו את ההגדרה של ה-UML (מטה-UML) והפלט שלו יהיה כלי ל-UML.

(16:א) Method Compiler הוא מטה-כלי. הקלט שלו הוא תיאור ER או תיאור DFD או תיאור FSM. הפלט עובר ל-design editor: עורך לעיצוב או לציור של דיאגרמות ER/DFD/FSM.

(15:ד)

(21) כלי שמיועד ל-testing: אנשים שבנו מפענח ל-JVM ורצו לבדוק אותו. המפענח של JVM מסוגל לעשות בדיקות בטיחות שהמהדר אינו יכול לעשות. הם רצו לבדוק את המפענח. בשביל זה הם פיתחו על-כלי:
(21:א) code generator generator: מטה-code generator כי הוא מקבל הגדרה של מטה-שפה. לכלי הזה הם מכניסים דקדוק של שפה (במקרה שלהם: דקדוק של שפת JVM).
ה-test זה תוכניות ב-JVM שאותן הם רצו לבדוק.
ה-seed הוא הקלט ל-code generator ובאמצעותו נוצר הקוד עבור תוכניות ה-test.

Co-Debugger: WYSIWIS: What You See Is What I See. תרשים:



UD: User Display.
PA: Personal Assistant.

UD Agent: עוסק בפיתוח הקונקרטי.
PA Agent: כל סוגי התקשורת האחרים. לכל אחד יש "מזכירה פרטית" כזו שמנהלת שיחות, טיפול בבקשות והעברת בקשות. פונקציות: stop session, start session, leave, enter, exit.

האנשים עובדים בו-זמנית. מבין כל האנשים רק לאחד (ל-operator) יש אפשרות לגעת בתוכנה בזמן מסוים, אבל כל האנשים יכולים לראות את זה. רק כשהוא מוותר על ה-"floor", מישהו אחר יכול לגעת בתוכנה.
← קצב פיתוח מהיר יותר ומס' באגים קטן יותר (שכן יש peer-review ו-public domain).

ה-UD Agent רואה:
1. חלון של תוכנית מקור.
2. חלון של מחסנית הפעלה.
3. חלון של משתנים.
4. חלון של ה-floor.
5. חלון של היסטוריה של floors.

} כל אחד רואה את זה

לפי מה קובעים את ה-floor?
1. מי שבא ראשון (תור).
2. מי שכתב את המודול הנדון.
3. המלצה של בעל ה-floor הנוכחי.
4. מי שהתחיל את ה-session.

סדר עדיפות: נמוך ↓ גבוה

שאלות ותשובות:
1. האם מע' זו מתאימה לצורת העבודה של בנ"א (ללא מערכת)?
ת: מהי צורת העבודה של בנ"א בו-זמנית על אותו קטע???

- מבחינת peer-review: peer-review זה כאשר אחד מפתח ואח"כ מגלים באגים, והתיקונים נעשים במקום אחר.
- דומה ל-open source.

2. מה היעילות של תהליך הפיתוח עם ובלי המע'?
ת: אם יש הרבה אנשים בצוות, אזי זה פועל בצורה לא יעילה.
במע' המתוארת התקשורת היוותה גורם מעכב.

7.10 תיעוד: אוסף של כלים המיועדים לתיעוד (כלים לציור, מעבד תמלילים, יצירת HTML, desktop publishing וכו').

7.11 Cross-Development:

פיתוח תוכנה בארכיטקטורה אחת עבור ארכיטקטורה אחרת.

איזה כלים צריך ב-workbench הזה?

- Cross Compiler.
- Simulator (אחרי קבלת ה-object code צריך לעשות מבדקים).
- תקשורת עם המטרה.
- כלי מדידה והשגחה מרחוק.

8. Integrated Environments:

8.1 Ada Programming Support Environment – APSE:

(א:22) צריך להיות גרעין – Kernel APSE - KAPSE – שצריך להיות משותף.
מעל לזה – Minimal APSE – MAPSE – בו יושבים כלים בסיסיים.
זו הפעם הראשונה שנתנו ביטוי איך צריכה להיראות <u>סביבת</u> פיתוח שמיועדת לשפה מסוימת. אולם הדבר לא עבד:

1. היה יותר קשה לפתח אינטגרציה מאשר חשבו תחילה.
2. מפתחים היססו בגלל פיתוח ה-PC והגרפיקה.
3. לא כ"כ הבינו בכלל מה זה data integration.
4. אי-אפשר היה לפתח סטנדרטים בגלל טכנולוגיות שהשתנו מהר מדי.
5. הניסיון הראה שהתפוקה אינה עולה משמעותית הודות לכלי CASE.
6. סוף המלחמה הקרה. הדבר גרם לכך שתקציבים של צבא ארה"ב ירדו באופן דרסטי ולא רכשו כלי CASE ולכן מפתחים העדיפו לא לגעת בזה.

8.2 Unix:

Unix אינו environment. האם הוא toolbox? אולי.
Unix גם אינו workbench שכן הוא מכיל הרבה כלים.
מס' כלים ש-Unix מכיל: wc (counts characters, words and lines), sort, grep, cat, ed, vi, emacs, comm (compares sorted files), uniq (compares 2 consecutive rows), spell, nroff/troff, make, lex/yacc.

8.3 Integrated Software Engineering Environment – ISEE:

כדי שתוכנה תהיה ISEE צריך אינטגרציה של 5 סוגים:

1. פלטפורמה (ב:16).
2. נתונים.
3. הצגה: הכלים השונים מתנהגים באותה צורה.
4. בקרה: כלי אחד קורא לכלי השני.
5. process: תהליך התוכנה מוגדר ויש סביבה שמנהלת את התהליך ואומרת מה לעשות.

<u>*חלוקת ה-environment ל-3 שכבות:*</u>

1. Workbench Applications: מפתח כלים. כמפתח כלים צריך לדעת את השפה בין ה-framework services ל-workbench applications.
2. Framework Services: מפתח environments: צריך לדאוג לאינטגרציה.
(ד:16) מודל הטוסטר אמור לספק את השירותים הבאים:
א. שירותי repository:

שירות	תיאור
Data Storage	מספק תמיכה ליצירה, קריאה, עדכון ומחיקה של ישויות , כאשר לישות יש שם, קבוצת ערכים ועשויים להיות לה קשרי גומלין.
Relationship	מספק תמיכה להגדרה וניהול של יחסי גומלין בין ישויות סביבה.
Name	מספק תמיכה ליצירת שמות לישויות. לישויות יש גם מאפיין ייחודי שניתן להם בידי שירותי ה-repository.

Location	מספק תמיכה לביזור הישויות ברשת של תחנות עבודה, ולכן כולל פעולות כמו: הזזה, העתקה וכו'.
Data Transaction	מספק תמיכה לטרנזקציות אטומיות שמאפשרות למסד הנתונים להתאושש במקרה של כישלון.
Concurrency	מספק תמיכה לסימולציה של טרנזקציות מרובות.
Process Support	מספק פעולות תהליך כמו: התחלה, סיום, השהייה וכו'.
Archive	מספק תמיכה לאחסון off-line ושחזור ישויות.
Backup	מספק תמיכה לשחזור נתונים במקרה של כישלון.

ב. שירותי אינטגרציה של נתונים.

שירות	***תיאור***
Version	מספק תמיכה לניהול ורסיות מרובות של ישויות.
Configuration	מספק תמיכה לקיבוץ ישויות לקונפיגורציות וניהולן כישות מורכבת.
Query	מספק גישה ועדכון שירותים לורסיות.
Meta-Data	מספק אמצעים להגדרת סכמות וניהול.
State Monitoring	מספק אמצעי triggering שמאפשרים לבצע לפעולות מסוימות כשמגיעים למצב מסוים של מסד הנתונים.
Sub-Environment	מספק תמיכה להגדרה וניהול של תת-קבוצות של הנתונים ושל פעולות בסביבה, ומאפשר להתייחס אליהן כסביבה נפרדת.
Data Interchange	מספק מכניזם לייבוא וייצוא של נתונים מהסביבה.

ג. שירותי ניהול תהליך התוכנה.

שירות	***תיאור***
Task Definition	מספק אמצעים להגדרת המשימה שכוללים pre- and post-conditions, קלטים ופלטים, משאבים נחוצים והתפקידים המעורבים במשימה.
Task Execution	מספק אמצעים לתמיכה בביצוע המשימות. זה עשוי להיות כרוך בהגדרת אינטראקציות המשימה בתהליך שפת תכנות.
Task Transaction	מספק תמיכה לטרנזקציות הכוללות משימה אחת או יותר שמתבצעות במשך הרבה זמן. צריכה להיות אפשרות ל-recovery מכישלון בלי צורך ב-roll back של המערכת למצב שלפני שהמשימה החלה.
Task History	מספק אמצעים לשמור את ביצוע המשימה ולבדוק ביצועים קודמים.
Event Monitoring	תומך בהגדרת המאורעות או ה-triggers שגורמים למשימה מסוימת להתבצע.
Audit and Accounting	מספק מידע על מה שבוצע ובאילו משאבים השתמשו בסביבה.
Role Management	מספק אמצעים להגדרה ולניהול תפקידים בסביבה.

ד. שירותי הודעות.

שירות	***תיאור***
Data Repository	כל שירותי ה-data repository מסופקים ע"י ECMA PCTA, למעט שירות ה-backup.
Data Integration	מספק את כל שירותי אינטגרציית הנתונים, למעט שירות query.
Task Management	לא מספקים שירותי ניהול משימות, למעט שירותי auditing and accounting.
Message	מספק שירות של שליחת הודעות.
User Interface	מניחים שסביבות מבוססות PCTE משתמשות ב-X-Windows למימוש מנשק המשתמש.

ה. שירותי מנשק MMI (Man-Machine Interface).

3. Platform Services.

8.4 Portable Common Tool Environment – PCTE:

(22:ב)
(20) ל-PCTE יש repository (מאגר + שירותים מסביב).
ב-PCTE יש סכימות מוגדרות מראש (source program pipe) ויש סכמות שכל אחד יכול להגדיר. הגדרת סכימה חדשה נעשית ע"י meta-schema (כשעושים instantiation ל-meta-schema מקבלים schema).

דרישות של מאגר PCTE:

1. מבנה מורכב המתאים לסוגי פריטים מגוונים. הגישה לפריטים לא תהיה בעלת גירעון גס/עדין מדי.
2. ניהול ורסיות.
3. אורך משתנה של נתונים (פריטים).

4. פעולות מקוננות על המאגר.
5. יכולת של עבודה בו-זמנית.
6. data integration.
7. control integration (pipes).

9. Process Centered Software Engineering Environment – PCSEE

המטרה: מע' אוטומטית שתכוון אותנו.

9.1 הגדרות

הגדרה 14:

1. סביבת הנדסת תוכנה מכוונת לתהליך התוכנה (PCSEE) היא סביבה בה התהליכים בהם משתמשים בפיתוח מוגדרים בצורה מפורשת ע"י המשתמש, וטבועים בתוך הסביבה.
2. צעד בתהליך (process step, task) הוא פעולה אטומית של התהליך, ז"א: פעולה שאינה ניתנת לחלוקה, שאין לה מבנה משלה.
3. סוכן (agent) הוא ישות (אדם או מחשב) המבצע צעד של תהליך.
4. משאב הוא ישות הנדרשת כדי לבצע צעד של תהליך (למשל: source code).
5. מוצר הוא פלט של צעד של תהליך.
6. מודל של תהליך הוא תיאור של הצעדים המבוצעים ע"י סוכנים, תוך שימוש במשאבים, כדי לייצר מוצר.

(22:ג), (22:ד).
(23:א) – ארכיטקטורה של PCSEE.

דרישות מ-PCSEE:

1. אנטי-דרישה: יש פעילויות שהן קריאטיביות.
2. תיאור התהליך לא יכול להיות סטטי. הוא חייב להתאים לסביבה העבודה, לצוות, למשאבים ולפרוייקט.
3. ייתכן שיש מגבלות חיצוניות (למשל: ISO9000).
4. PCSEE מתאם פעילויות של הסוכנים.
5. יכולת הגדרת התהליך באופן איטרטיבי.

- יש מערכות שהן מבוססות דיאגרמות או מבוססות שפות תכנות.
- PCSEE עוסק ב:
  - What to do
  - When to do it
  - How to do it
  - Who should do it
  - With what to do it

PCSEE מכוון 3 תהליכים:

1. Software Engineering:
   - א. Automation: תהליך הפיתוח צריך להיות לפחות באופן חלקי אוטומטי – מכוון ע"י הסביבה.
   - ב. Guidance: הסביבה צריכה להדריך אותך.
   - ג. Enforcement: לא נותן לעשות דברים המנוגדים לסביבה.
   - ד. Status: בודק איפה הפרוייקט נמצא מאספקטים שונים.
2. Project Management:
   - א. Status Monitoring.
   - ב. Modification.
   - ג. Controlling.

   (לאו-דווקא קשורים לפרוייקטים בהנדסת תוכנה, אלא לפרוייקטים בכלל)
3. Process Engineering – הגדרת תהליך התוכנה, אופטימיזציה:
   - א. Analysis.
   - ב. Definition.
   - ג. Simulation – לגלות אם הגדרת התהליך טובה או לא.
   - ד. History.
   - ה. Measurement: שייך לרמה 4-5 ב-CMM.
   - ו. Improvement: שייך לרמה 4-5 ב-CMM.

למה צריך להתייחס?

1. הצגת התהליך.
2. ניתוח התהליך.

3. Process Instantiation: נותנים הגדרה כללית של התהליך ואח"כ למודל של התהליך עושים instantiation (קובעים מי יעשה מה).
4. Process Enforcement – הפעלת התהליך.
5. Process Simulation.
6. Process Monitoring.
7. תמיכה בשינוי התהליך.

- תהליך מוצר זה פיתוח מע' תוכנה משלב הדרישות ועד ההרצה. יש כאן <u>יצירתיות</u>. משתדלים שהאלמנט היצירתי יהיה קטן כמה שאפשר!

<u>*איך הדברים צריכים להתממש?*</u>
(22:ד)
יש PCSEE שהוא resource oriented ,product oriented ,activity oriented (ה-x-oriented מתאר את היישות המרכזית שמופיעה ב-PCSEE). למשל, בחב' תעופה הפעילויות הן בעיקר resource oriented. במקרה של תוכנה שבה מוגבלים מאוד המשאבים זה גם resource oriented.

הגדרה 15:
1. תחזוקה מתקנת עוסקת בתיקון שגיאות שנכנסו תוך כדי תהליך התוכנה (הן בשלב הפיתוח והן בשלב התחזוקה).
2. תחזוקה מתאמת עוסקת בהתאמת מערכת תוכנה לשינויים בדרישות ומפרטים של המערכת.
3. תחזוקה משכללת עוסקת בשיפור הביצועים של המערכת.
4. תחזוקה יצירתית עוסקת בשילוב רעיונות חדשים במערכת.

<u>*9.2 מודלים למחזור חיים*</u>

<u>*9.3 APPL/A, Spade, Process Weaver*</u>
1. ***APPL/ADA:*** שפת תכנות המבוססת על Ada להגדרת תהליך.

(23:ג) מלמעלה באה הגדרת תהליך. עם ההגדרה המתכנת יכול לפתח תוכנה ולקבל תוכניות. למעלה יש מהנדס תהליך שגם מתכנת. התוכנית שהוא בונה זה ה-instantiation של תהליך מסוים ולכן:
"Software Processes are Process Too" – ניתן להגדיר את התהליך ע"י תוכנה.

(24:ב) יש מאורעות שמגדירים ויש trigger שכל הזמן פעיל.
אחד הדברים הבעייתיים הוא לשנות את הגדרת התהליך.

<u>*דוגמה:*</u> יש תהליך מוגדר. קנינו עוד כלי, למשל wc, וצריך להפעיל אותו עם sources חדשים. יש trigger שפעיל כל הזמן. כאשר מישהו מביא source שיש לתרגם ל-object הוא מפעיל את wc ומוסיף את הנתונים. או כאשר מישהו מוחק source וכו'.
כל ה-insert ו-delete מוגדרים ב-relation.
← ניתן להגדיר שינויים בתהליך על-פי תוכנית. הפורמליזם להצגת התהליך זה קוד.

Lehman ביטל גישה זו, שכן הוא טוען שאיננו יודעים לכתוב תוכניות טוב ולכן השילוב בין כתיבת תוכנית להנדסת תוכנה הוא לא שילוב טוב.

(26) ***Process Weaver:*** <u>toolkit</u> מסחרי. הפורמליזם להצגת התהליך זה רשתות פטרי.

(27) המע' הקיימות פועלות בצורה הטובה ביותר בתחום של ניהול תצורות וניהול ורסיות.

(28) ***Spade – Software Process Analysis, Design and Enactment:*** גם מבוסס על רשתות פטרי.
(28:א) תיאור תהליך ה-testing.
עיגול = תנאי. אם יש אסימון בעיגול הכוונה היא שהתנאי קוים.
המלבן עצמו הוא רשת פטרי בפני עצמה.
בסופו של דבר זה אמור לתת תוכנית.
בשפה יש type שהם built-in וזה סוג של ירושה.

<u>*10. הנדסה אחורנית*</u>

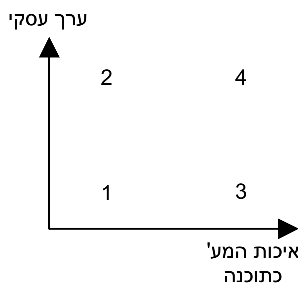
<u>*10.1 מבוא:*</u>
הגדרה 16:
1. הפשטה של מערכת היא מודל המסכמת את הפרטים של מערכת תוכנה שהיא מייצגת. להפשטה ברמה גבוהה יש פחות פרטים מאשר הפשטה ברמה נמוכה.

2. הנדסה קדמונית (forward engineering) היא התהליך (המסורתי) של מעבר מהפשטה ברמה גבוהה למימוש פיזי של מערכת.
3. הנדסה אחורנית (reverse engineering) היא התהליך המנתח מערכת כדי:
   א. לזהות את מרכיביו והיחסים ביניהם.
   ב. לייצר הצגות של המערכת בצורה אחרת או ברמת הפשטה גבוהה יותר.
   זהו תהליך של בחינה, לא תהליך של שינוי או הכפלה.

הגדרה 17: להבנת תוכנה יש מספר רמות.
1. מבט על רמת המימוש הוא ההבנה של המרכיבים הבסיסיים של המערכת, בד"כ פריטים של בקרה או של הצהרה [מסתכלים על הקוד עצמו].
2. מבט על רמת המבנה הוא ההבנה של איך המרכיבים הבסיסיים קשורים אחד לשני, למשל: לפי טווח של הצהרה או הכלה.
3. מבט על רמת הפונקציונליות הוא הבנה של תפקיד כל מרכיב כיחידה בודדת.
4. מבט על רמת התחום הוא ההבנה של ההקשר בו פועל כל מרכיב של המערכת.

הגדרה 18:
1. תיעוד מחדש (redocumentation) הוא היצירה או השכתוב של תיעוד של מערכת שקולה באותה רמת הפשטה.
2. שחזור התיכון (design recovery) הוא חלק מההנדסה האחורנית בו מוסיפים ידע תחומי, ידע חיצוני, היסק לוגי ועוד לתצפיות על מערכת תוכנה קיימת, כדי לזהות הפשטות גבוהות יותר.
3. הבנייה מחדש (restructuring) היא טרנספורמציה של הצגה אחת לשנייה באותה רמת הפשטה, תוך שמירת התנהגות המערכת (סמנטיקה ופונקציונאליות).
4. הנדסה מחדש (reclamation = rejuvenation = renovation = reengineering) היא בחינה ושינוי של מערכת ובנייה מחדש.
5. שימוש חוזר (reuse) הוא השימוש של חלק של מערכת תוכנה (תיעוד, דרישות, מפרט, תיכון, קוד, מקרים לבדיקה וכדו') לבניית מערכת אחרת.

<u>למה רוצים לבצע הנדסה אחורית?</u>
1. כדי להגביר את ההבנה של המערכת הקיימת:
   א. לתחזוקה.
   ב. לפיתוח חדש.
2. לצמצם את סיבוכיות המערכת (לתחזוקה; בשביל restructuring).
3. לייצר מבטים חדשים על המערכת.
4. לשחזר מידע שאבד.
5. לגלות תופעות לוואי.
6. להקל על שימוש חוזר.
7. שלב ראשון של reengineering.



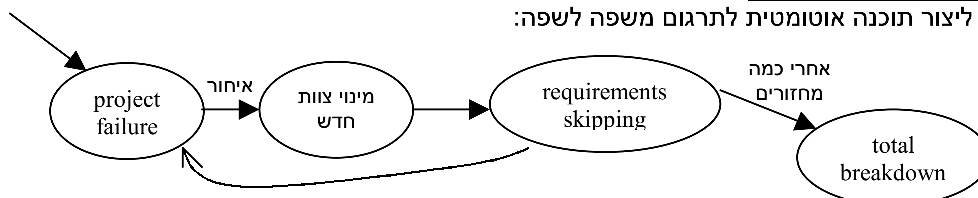
<u>השיקולים: מתי עושים מה:</u>
1. אם איכות התוכנה ירודה ביותר והערך העסקי נמוך: זרוק את המערכת.
2. יש למע' ערך עסקי גבוה, אבל איכות התוכנה נמוכה ← עושים reengineering או פיתוח מחדש.
3. ערך עסקי נמוך, איכות גבוהה: להמשיך בתחזוקה (או לזרוק).
4. ערך עסקי גבוה, איכות גבוהה: להמשיך בתחזוקה (אולי reengineering).

ציר ה-y מצביע על המימד הסוציו-אקונומי וציר ה-x מצביע על המימד המחשובי.

בעלי העניין שקובעים את רמת איכות המערכת כתוכנה הם: צרכני המע', אגף המחשבים.
בעלי העניין שקובעים את רמת הערך העסקי בחברה: מנכ"לים וכדו'.

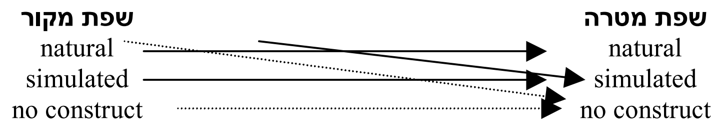
<u>הקשר לחוקי Lehman:</u> לפי Lehman מע' תוכנה היא מע' ששוכנת בעולם האמיתי.

<u>10.2 תרגום משפה לשפה:</u>
ניסיון ליצור תוכנה אוטומטית לתרגום משפה לשפה:



מה שכן יצליח זה הסבה משפה לשפה בארגון שזו לא מטרתו. כלומר, הארגון יפתח וישקיע משאבים בהמרת תוכנה מסוימת (הדבר לאו דווקא יגרור רווחים).

מדוע זה לא עובד?



מעבר כזה מהווה בעיה.
הדבר היחיד שניתן לשאוף אליו זה semi-automated.

למה לבצע הסבה משפה לשפה?

- עוברים ל-hardware אחר.
- עוברים למע' הפעלה אחרת.
- חסרים אנשים שמבינים בשפה הנתונה.
- רצון לעבור ל"שפה אופנתית".

בעיה נוספות שקיימת: homonym (אותו דבר, סמנטיקה שונה) ו-synonym (דברים שונים, אותה משמעות).

10.3 הנדסה מחדש של נתונים:

דורש reengineering ואח"כ הנדסה מקדימה.
איך עושים reengineering? עוברים על כל הקוד וכל פעם שמוצאים record שמים אותו ב-repository (ניתן לעשות בצורה semi-automated) כ-ERD (ב-ERD ניתן בקלות לייצג כל מה שרוצים). אח"כ צריך לעבור עליהם ולהשוות ביניהם.

- אם יש כמה repositories ופונים לסירוגין לחלקים מהם ← הנדסה מחדש תיצור database אחד.
- קבועים שלא הוצהרו אלא נכתבו בקוד כמספרים, וצריך לעבור על כל הקוד על-מנת לשנותם.

10.4 הנדסה אחורית (Reverse Engineering):

נרצה לעשות הנדסה אחורית כשלב ראשון לפני reengineering או לפני בניית מע' חדשה.
הנדסה אחורנית דורשת עלייה ברמת האבסטרקציה. אין כלי אוטומטי שיכול לבצע את זה.
בשביל לבצע הנדסה אחורנית צריך repository:
- ניתוח התוכנית: מציאת כל המשתנים שבתוכנית (cross-reference). איזה פונקציה קוראת לאיזו פונקציה.

(30), (31:ב).

10.5 Restructuring של קוד ומודוליזציה.

11. למה לא משתמשים ב-CASE?

ברוב המקומות לא הכניסו מערכות environment מסודרות של כלי CASE ושילבו אותם בהגדרת התהליך.
הסיבות לכך:

1. בזמנו עשו מ-CASE silver bullet. ברגע שמנפחים את הציפיות וזה לא מתממש ← אנשים מתאכזבים. זה לא פותר את משבר התוכנה.

[הציפיות מכלי CASE: התפוקה תעלה; הוצאות פיתוח ותחזוקה תרדנה; זמני פיתוח יהיו קצרים יותר; התיעוד יהיה מוצלח ורחב יותר]

2. זה לא העלה את התפוקה. בשביל מה להשקיע מאמץ בזה אם לא ברור שזה יעזור?

הגדרה 19: נתונה מערכת תוכנה P ו"קריטריון פריסה" $C = <s, v>$ כאשר S פקודה ו-v משתנה. פרוסה (slice) של P ביחס לקריטריון C היא אוסף של פקודות S המקיים:

א. S מתקבל מ-P ע"י מחיקת אפס או יותר פקודות.
ב. S נכונה תחבירית.
ג. הערך של v בדיוק לפני הפקודה s זהה עבור execute(P) ו-execute(S).

ה-slicer הוא כלי להנדסה אחורנית ולהנדסה.