

קורס הנדסת תוכנה

סיכום:

הרצאה 1

- **Scale-Up** – היכולת להגדיל את התמיכה לכמות משתמשים גדולה יותר בזמן.
- **Maintenance** – תחזוקה של התוכנה (עדכוני תוכנה, טיפול בバאגים וכו').
- **Log's** – תיעוד של המערכת שנכתבה.
- **מוצרים גנריים** – מוצרים שנמצאים על מדף של החברה ומוכנים לקנייה כמו שהם.
- **מוצרים שהליך מבקש מחברת תוכנה מסוימת ליצור עבורו.**
- **תוכנה מקצועית** – מפותחת על ידי צוותים, תוכנה שאנשים מוכנים לשלם עליה כסף. עבור תוכנה מקצועית יש צורך לתרום בלקוח, לתת לו דוקומנטציה, לדאוג שה מוצר יפעל בצורה תקינה וכו'.
- **הטרוגניות** – בעיות כלויות שחזרות על עצם בפיתוח תוכנה, Security, Stability, Scaling.
- **אפליקציות Stand Alone** - אפליקציה שאינה דורשת שימוש ברשות כדי שתעבד, קשה לדאוג לתחזוקה שלה כי היא אינה רצה על הרשות, אך עדין יש פניה לשרת כדי לדעת האם יש צורך לעדכן תוכנה.
- **בדרכ נטול המחיר של פיתוח תוכנה גדול יותר מהחומרה הנדרשת כדי לפתח את התוכנה.**
- **מודל עסק ידוע היום עבור אפליקציות שרצות על הרשות, למשל יוטיוב, הן קבלת הכספי ממפרטים ופחות משתמשים אשר רובם משתמשים בשירות בחינם. רוב השירותים היום נדרכים מאפליקציות אשר מותקנות ורצות על שרת מסוים שניית לגשת אליו דרך ה Web ולא להוריד את התוכנה למחשב האיש שלנו.**
- **Native Application** - אפליקציית Web שמיועדת להתקנה על מכשיר מסוים למשל הטלפון שהמשתמש מתקין בלבד על מנת לקבל שירות שנראה ומרגש יותר נוח וחלק לשימוש.

• **תכונות של תוכנה טובה:**

- **Maintainability** – יכולה להתאים את עצמה לכמות משתמשים עולה.
- **Dependability and Security** – טווח רחב של תכונות ובינן אמינות, אבטחה, בטיחות (במקרה של קריית מערכת שלא יהיה סכנה של אבוד מידע או כספים).
- **Efficiency** – שהתוכנה תבצע ניצול יעיל של ה- System Resources, למשל ניצול יעיל של הזיכרון, זמן עיבוד מהיר וכו'
- **Acceptability** – מותאם למערכות שונות, למשל שייעבור גם עבור משתמשי אפל וגם עבור משתמשי וינדוז.
- מערכות שונות דורשות תהליכי פיתוח שונים, למשל מערכות Real-Time دورשת תהליך פיתוח שונה מממשק של e-commerce, כמובן שזה משתנה לפי אופי המערכת והדרישות הרצויות.

• **סוגים של תוכנות:**

- **Stand-Alone** – הוסבר בעמוד הקודם
- **Interactive transaction-based application** – תוכנה שהפעולה העיקרית שלה היא לתקשר עם המשתמש מאגר מידע ומבצע Transaction Molutores Data base.
- **Embedded control systems** – תוכנה שmagua ביחד עם מוצר מסוים, למשל תוכנה שמודעת איך להפעיל את מכונת הכביסה בכל מיני מצבים.
- **Batch processing systems** – מערכת שמיועדת לעיבוד גדול של מידע, רצה מאחוריו הקלעים מנתחת את המתוונים ומיצרת דוח בסוף כל תקופה מסוימת.
- **Entertainment system** – אתרי הימורים וכו'
- **Systems for modelling and simulation** – מערכות שמפעילות סימולטורים מסוימים.
- **Data collection systems** – מערכות שתפקידם לאסוף מידע, למשל סוסורים במפעלים או ספדים את המידע על הפעולה שהם עוקבים אחריה.
- **Systems of systems** – מערכת שתפקידה לסנכרן בין מערכות.

הרצאה 2

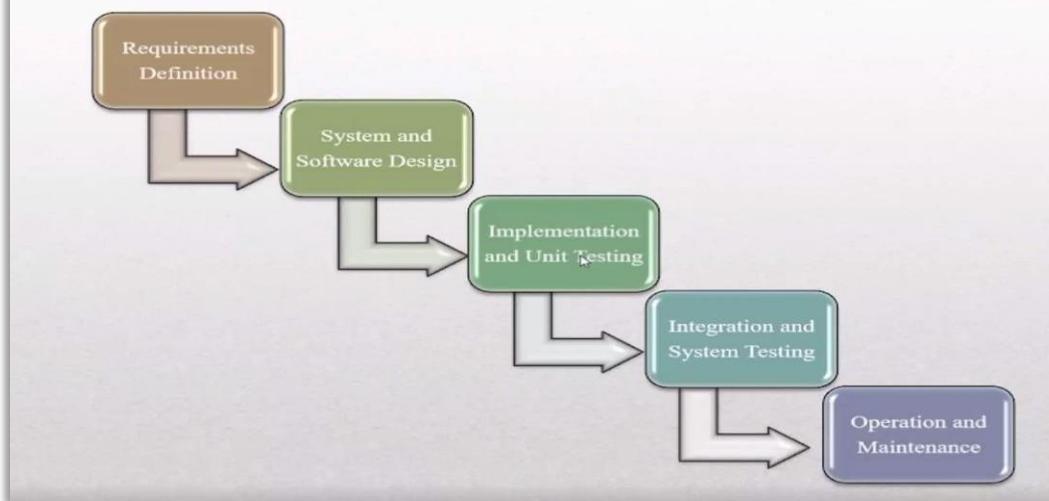
- **Web Application** – אפליקציה שעבדת אל מול שרת, למשל פניה לבנק להעברת יתרה, צריך להעדכן גם בשרתים של הבנק ולא רק בטלפון.
- **מחשב ענן** – חדרי ענק של שירותי שנמצאים בבעלות של חברות גדולות אשר מוכרים שירותים של שירותי עבור חברות קטנות, חוסף לחברות הקטנות

עלויות של תחזוקת חדרי שירותים וטיפולים נוספים. בנוסף מאפשרים לחברות להשתמש באלגוריתמים מסוימים מסיבות שונות של אוטם חברות קטנות.

• 4 שלבים בפיתוח תוכנה:

- **Software Specification** – שלב ההגדה של הלקוח וההנדסיה תוכנה על איזו תוכנה הם מתכוונים לפתח ומה האילוצים עבור התוכנה.
- **Software Development** – שלב עיצוב התוכנה וכתיבה הקוד.
- **Software Validation** - שלב הבדיקה שהתוכנה עובדת שזו אכן התוכנה שביקשנו.
- **Software Evolution** – שלב שינוי התוכנה הן מבקשת הלקוח או שינוי בשוק.
- **מודלים לפיתוח תוכנה** – זה בעצם מתודולוגיות שונות לביצוע השלבים שלמעלה.
- **Plan driven Method** – מודל שמוסחת על תיכנון היבט כל שלב בפיתוח התוכנה ולספק מסמכים לשלב התוכנות.
- **Waterfall Model** – השלבים של פיתוח התוכנה מתבצעים בצורה של מפל, רק בסיום השלב העליון ממשיכים לשלב הבא בפיתוח.
- **חרונות** – הלקוח נotent פידבק רק בסיום הפיתוח של המוצר ויתכן כי הלקוח ירצה לשנות משהו ואז יהיה צורך לחזור אחריה ולתקן, בנוסף יש אפשרות שהיא שנעשה כבר לא יהיה רלוונטי לשוק.

Waterfall model



- **Incremental Development Method** – פיתוח של התוכנה בצורה אינקרמנטלית, כל פעם לכתוב חלק קטן מהמערכת ולהציג ללקוח כדי לקבל פידבק האם זה מה שהוא בקש והאם זה לשביעות רצונו, אם כן אז ממשיכים לפיתוח של החלק הקטן הבא וכן הלאה, אם זה לא לשביעות רצונו של הלקוח אז מבצעים מקופה שיפורים.

- **יתרנות** – הליקות תמיד נמצא בשלבי הפיתוח ונותן פידבק בזמן אמיתי, יותר קל לטפל בעבויות בזמן אמיתי.

Incremental development



שאלות נפוצות:

Frequently asked questions about software engineering

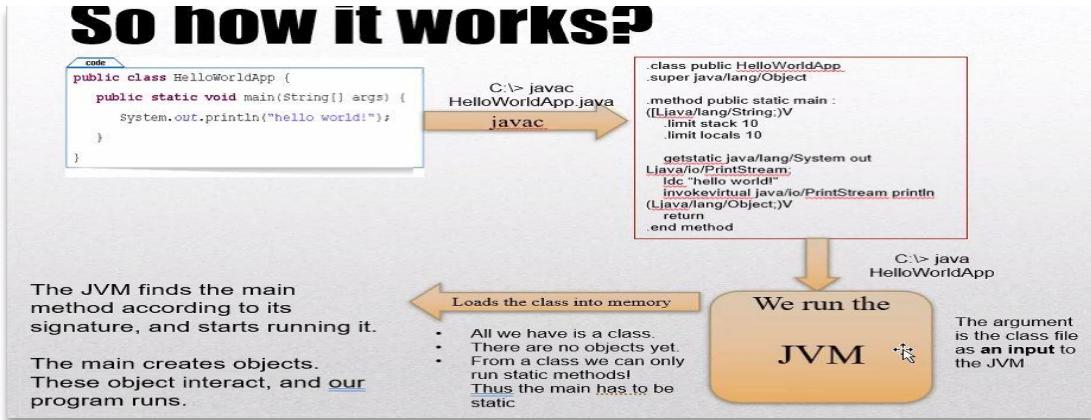
Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

הרצאה 3

- **Re use** – שימוש חוזר של קוד מסויים (הסבה של קוד קיים לתוכנה אחרת).
- ב – Java הטיפוסים הם טיפוסים פרימיטיביים ולא אובייקט כמו בפייתון (int,double..).
- מחרוזת ב- Java זו מחלוקת בפני עצמה.
- **יתרון למנוחה עצמים טהור** זה שיש הרבה יכולות לשלוט בשפה, כמו למשל אפשרות להציג לרבים תשתיות כדי לעיל את תהליכי כתיבת הקוד של התוכנית.
- **JVM Architecture** – הקומpileציה זה יוצר קובץ הפעלה שיכל לרוץ על גבי מחשבים אחרים ובעל-

ממשקים שונים (Mac, Windows..), זה מאפשר לחברות תוכנה לדלג על השלב של התאמת לממשק מחשב של לקוחות. חשוב מאד, אף פעם אתה לא שולח ללקוח קוד מקור.

So how it works?



- **GC(Garbage Collector)** – תוכנית בפני עצמה שרצה במקביל כחלק מההרצה של התוכנית שלנו, כדי לעזור לנו בתור תוכניות לפנות זיכרון שהוא לא בשימוש.
- אם ב- `hi` לכתובת מסוימת אין מצביע לאובייקט מסוים אז ה-GC מנקה את הזיכרון הזה.
- **יתרונות** – לתוכנת אין צורך לשחרר זיכרון.

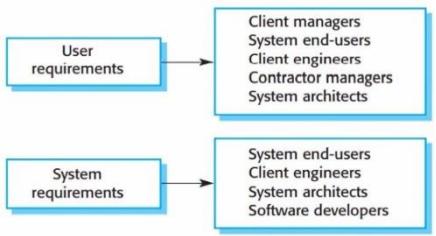
שלב ראשון בהנדסת תוכנה – System Specification – (הגדרת דרישות).

- התאמת ציפיות בעזרת מסמך דרישות, הדרישות צרכות להיות אבסטרקטיות (תקציר) למה נדרש ואיך יראה המוצר, זה יוצר בעיה של הרבה דרכים לראות את המוצר וגורר חוסר התאמה בין איך שהליך רואה את המוצר לבין שהמתכנת רואה אותו. ולפעמים מסמך מפורט יותר, זה תלוי לkokoth. הלוקוח לא תמיד יוצר את המסמך הזה ומיל שועזר לkokoh ליצור את המסמך זה ה- Marketing ומסמך שיוצא נקרא **MRD**(Marketing Requirements Document) או **URD**(User Requirements Document)

– המסמך היותר מפורט של PRD (Product Requirements Document)

דרישות הלקוח.

Readers of different types of requirements specification



:URD/MRD,PRD

Example

URD/MRD	PRD
High Level	Detailed Level
<ul style="list-style-type: none">The system will support shoe sales in the stores.	<ul style="list-style-type: none">Daily inventory ReportHistorical report about historical levels of inventory. The report should be graphic and up to a year back.The system will send daily alarms about low levels of supplyEach store will have a different threshold.The system will provide geographic report (uniting stores in the same geographic region).

Stakeholders Examples for a Medical Clinic

- Clinic customers and their family relatives.
- Doctors/nurses
- Treatment coordinators
- IT system team
- Ethics manager
- Operation Managers
- Legal and standards consultants.

Stakeholders – בעלי עניין ישירים או עקיפים במערכת לטובה או לרעה, כלומר אלו שה מוצר טוב להם ואלו שהוא לא טוב להם, למשל תוכנה לקניית מוצרים באופן עצמאי ללא קופאיות.

נרצה שיהיה סגמנט נוסף במסמך דרישות שנקבע כך, נמצא ב- MRD/URD.

צריך שצוטרי הבדיקות של המערכת יכירו את המסמך (PRD,MRD/URD) ושהמסמך יהיה נגיש להם.

Feasibility Study – שלב של חקר ישימות, כלומר לחזור האם יש לנו את המידע המסופיק כדי לעמוד בדרישות למערכת, האם ניתן לבנות את המערכת בזמן המוקצב והאם הפיתוח יעמוד בתקציב המוגדר.

URD/MRD – User/Marketing Requirements Document •

מסמך שפרט את דרישות הלקוח באופן רחב, במסמך מצוינים בעלי העניין של המערכת.

כיצד נבנה את המסמך בצורה טובה? **نمינה שיטות לאיסוף המידע הנדרש:**

- **Interviews:** לראיין את ה- Stakeholders שאלות הקשורות למערכת ואנו מתעדים את אותם ראיונות. הראיונות הם סגורים וגם ראיונות פתוחים.
בראיון סגור - נשאל את המרואיין שאלות שהכנו מראש.
בראיון פתוח – זה בעצם דיון שאינו נדרש באופן קפדי בין בעלי העניין והאחראים על מסמך הדרישות.
- **Observations:** לצפות ביום העבודה של מקום שאתה רוצה למחשב ולתעד את זה, לפחות יוטר טוב מהראיונות כי הרבה פעמים בעלי העניין (Stakeholders) לא מודעים לקבלת החלטות שלהם במהלך התהליך המסורם.
- **Scenarios** – לתאר תסיטיטים שהלקוח יתkal בהם עם המערכת, (לספר סיפור).

למשל מה הדבר הראשון שהלקוח רואה כשהוא ניגש לשימוש במערכת, מי מטפל במערכת במצב של תקלת ואייזה צעדים יש לבצע כדי לטפל בבעיה ולהתאושש ממנה, באיזה מצב המערכת נמצאת לפני הבעיה, תיאור של המצב של המערכת בסוף אותו תרחיש וכו'.

Good Customer Requirements – מהי דרישת טובה? •

- **Unique** – שכלי דרישת קשורה לנושא אחד בלבד וייחודית.
- **Complete** – שהדרישה מכילה את כל האינפורמציה שמנדרת את הצורך במוצר והיכולת לבדוק שהדרישה זו מתקינה, חשוב להבין מי הם כל בעלי העניין שלנו.
- **Internal Consistency** – דרישות שאין סותרות דרישות אחרות של המוצר.
- **Atomic** – שהדרישה תהיה סגורה ושלא תהיה דרישת שנייה לחלק לשני דרישות (למשל שבתריט בדיקה יהיה ניתן לבדוק את הדרישה בפועל אחת או סדרה של פעוליות).

- – שהדרישה במסמך דרישות תהיה נcona וROLONGNTIAT לעכשו.
- – שניתן יהיה לסוג את הדרישה לצורך ספציפי של עסק מסויים.
- – שייהי ניתן לבדוק שהמערכת עומדת בדרישה.
- – שלדרישה תהיה פרשנות יחידה, כלומר שלא תהיה דרישת מעורפלת.

:Requirement Preparation Process Summary •

- – מהו היעד של המסמך דרישות שנקרא MRD/URD , כתת אותו גופים שבאים לבנות ללקוח את המערכת זה עוזרים ללקוח להבין מה הוא בעצם רוצה, והם בונים את המסמך דרישות המפורט שנקרא PRD.
- – דוקומנטציה של דברים שכבר נכתבו, ראיונות שצינו לעלה, מסמכים ייזום מוצר אשר מהווים את השלבים הראשונים של תכנון ופיתוח המוצר וכו'.
- – דיברו על זה בעמוד הקודם Interviews, Observations, Scenarios .Process
- – זה בעצם מסמך הדרישות. Main outcome
- – שלב ה- MRD/URD עולה בין 10-20 אחוז מההוצאות הכלולות של המערכת. Cost

:PRD (Product Requirements Document) •

המסמך PRD צריך לגשר בין הפער של איך המתכנת רואה את הדרישות של המערכת לדרישות כיצד שהליך רואה אותם, כלומר צמצום הפער בין הדרישות לביצוע בפועל.

PRD – המשמשים במסמך ה- PRD: Users of the document include ♦♦♦

- – לקוחות פוטנציאליים. Potential Customers
- – מנהלים (אחראים על תכנון של פעילות המשר וכו'). Managers
- – מעצבים המערכת. System Architects
- – צוות הבדיקה של המערכת. Testing
- – אנשי התחזוקה של המערכת. Maintenance

❖ Requirements Validation

- אימות שהדרישות במסמך אכן מגדירות את הצורך המלא והאמיתי של המשתמשים, במקרה שלא, חשוב לבצע תיקון לפני המעבר לשלב הבא, שכן נדרש את בניית המערכת ונדרש עם הלקוח שזה אכן מה שהוא מצפה, אם זו דרישת קטנה שקשורה ביצוע כמו למשל שינוי צבע, פונט וכו' אך ניתן לבצע את השינוי מבלי לשנות את מסמך הדרישות, אך אם זו דרישת יותר קריטית שמשנה דרישת אחרת אז ניתן לשנות את הסעיף הספציפי במסמך דרישות הנוכחי ולרשום שזה מסמך דרישות גרסה שנייה, כmodoן שנשמר את הגרסה הראשונה לצורכי מעקב, כך במסמך תמיד ישאר מעודכן.
- סותרות זו את זו - Internal Consistency tests
- Completeness – לבדוק שכל הדרישות מטופלות ואין דרישות חלקיות או סותרות.
- Feasibility – שלב של חקר יסימות שדבר עלי בהרצתה 3.
- Testability – שיליה ניתן לבדוק את המערכת ללא מחולקות בין לקווות וקובלנים פנימיים או חיצוניים.

תוכן מסמך ה- PRD מחולק לשני קטגוריות המיצגות שתי **סוגי דרישות עיקריות** והם נקראים כך:

❖ Functional and Non-Functional Requirements

דרישות פונקציונליות – Functional Requirements

חוسبים שצוטוי הפיתוח צריכים כדי לישם את המערכת בצורה מאוד פרטנית:

- תיאור השירותים שהמערכת מספקת.
- תיאור כיצד המערכת מגיבה לקלטים שונים.
- הסבר של מה המערכת אסור לעשות.

Functional Requirements

דוגמאות:

- Describe the users and their usage characteristics
- Natural language can be used to enable better review of both users and managers

Example

1. Every user of the system can search about shoe-type and inventory in any store
2. The system will produce a daily reports about shortages according to stores
3. Every user will have a unique identifier having 8 digits and English letters.

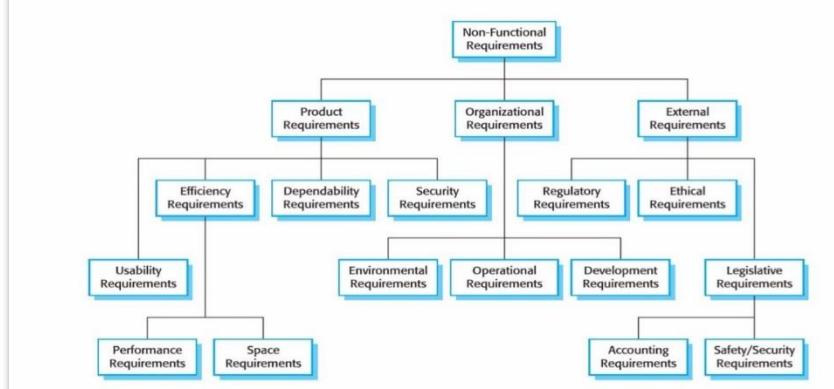
❖ דרישות שהם יותר חומרתיות – Non-Functional Requirements

- – ציון זמן התגובה של המערכת, באופן כללי.
- – ציון של אופן תהליכי ההזדהות (אם קיים כזה), למשל "Two Factor Authentication".
- – אילוצי משאבים, כולל מה דרוש מהמחשב ש郎 כדי שהמערכת תעבוד כנדרש.
- – תאימות עם אילוצים חיצוניים, למשל שפת מחשב מסוימת.
- – דרישות שהמערכת תפעל לפי כללים רגולטוריים ותעמוד בחוקים מסוימים שהוגדרו בחוק.
- – Safety – שהמערכת תהיה בטוחה לשימוש.

• IMPORTANCE OF NON-FUNCTIONAL REQUIREMENTS

- הדרישות הללו מואוד קשות לשינוי מכיוון שהן קשורות לדרישות מערכת שלמה ולא רק לנושא אחד ספציפי כמו בטיחות, אבטחה, משאבים, אילוצי תקשורת או אמינות וכו'
- הדרישות הלא פונקציונליות יכולות להיות הרבה יותר קריטיות מדרישות פונקציונליות כי היעדר דרישת אחת אפילו יכולה לעכב את השימוש במערכת.
- תיקון דרישת לא פונקציונלית עשוי לדרוש עבודה יותר נרחבת בהשוואה לתיקון דרישת פונקציונלית.
- דרישות לא פונקציונליות עשויות להשפיע על דרישות פונקציונליות, למשל דרישת אבטחה יכולה להיות מתורגם לדרישת לקיום טבלת מזהה ייחודית, המכילה שמות משתמש וסיסמות, הרשות וכו'.

Non-Functional Requirements Classifications



- **Ambiguity** – לאותו משפט שמתאר דרישת יכולה להיות משמעות שונה במוחם של המשתמש והפתח, סיטואציה זו יכולה לגרום לחוסר התאמה ולעיכובים משמעותיים בפיתוח המערכת ואף לביטולו.

- **כיצד נמנע מ-Ambiguity ?**

- הדרישות צריכה להיות ברורות ולא סטירות פנימיות.
- שימוש בשפה טבעית עלול להיות לא מדויק ובכך ליצור אי בהירות.
- למערכת גדולה ומורכבת יכול להיות הרבה בעלי עניין, שלכל אחד וקעימים שונים ודרישות שונות ובכך ליצור סטירות פנימיות.
- בעלי העניין לא מסוגלים לחזות את כל הדרישות של המערכת, ובכך לגרום לדרישות להיות לא שלמות.

מסיבות אלו ועוד צריך להשקיע זמן ומאזן על מנת להבטיח שמסמך הדרישות מלא את כל המטרות.

זהו בעיות בדרישות בשלב מאוחר בשלב היישום או לאחריו יכול לגרום לעליות הרבה יותר גבוהה מאשר אם היו מזהים אותם בשלב מוקדם.

- **מאפיינים של דרישות לא פונקציונליות (Non-Functional)**

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

הרצאה 5

:Software Engineering Development Methods •

:Build and Fix ♦

- בשיטת זו אנו מייצרים גרסה של המוצר מהר מאוד ומראים אותו ללקוח, הלקוח נותן פידבק, בעזרתו האינפורטטים של הלקוח אנו משפרים את הגרסה. אנו משפרים את הגרסה כל עוד הלקוח לא מרוצה, רק כאשר הלקוח מרוצה ומאשר את הגרסה אנחנו נצא מהלופ הזה.

בשלב הבא נעביר את הגרסה המאושרת לשלב ה- **Operate**, כלומר אנו משים ללקוח את הגרסה ופורטים אותה במערכות שלו ובכךאפשרים לו להשתמש בה, גם בשלב זהה יכולה להיות בעיה שהלקוח יצביע עליה ברמת ההטמעה או השימוש בגרסה כל עוד היא אצלו, במידה וזה קורה הגרסה תחזיר לשלב השינויים כל עוד הלקוח לא מרוצה.

רק כאשר הלקוח מרוצה מהמוצר יעבור לשלב הסופי – שלב ה- **Finalize** – שלב סיום הפרויקט.



- זהוי גישה שלרוב לא נוקטים בה.

יתרונות לגישה – קיצור הזמן עד שהלקוח רואה את המוצר (Time to Market).

חסרונות לגישה – הגעה מהירה ללקוח חלק מהיתרון, אך המוצר לא בהכרח נראה טוב או עובד טוב והלקוח יכול לוותר על העבודה שלנו כי המוצר "לא berhasil".

גישה שיכולה להתאים לסטרטאפים, ש.'**צריכים לצאת עם המוצר מהר או להראות ללקוח (POC)**'.

– **Plan Driven methods** ♦

טובה, רק כאשר מסייםים שלב אחד עוברים לשלב הבא:

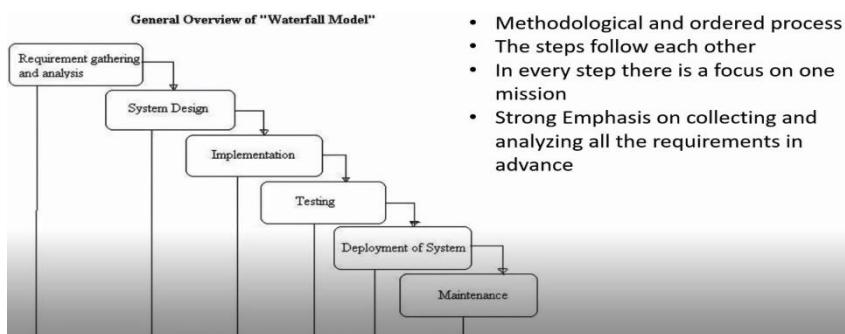
:WaterFall Model •

- **Requirement gathering and analysis** – שלב הגדרת הדרישות מהלקוח.

- **System Design** – שלב ההבנה איך המערכת אמורה להיראות בפנים ובחוץ, איך היא אמורה לתקשר עם רכיבים בתוך המערכת, איך היא

צריכה לתקשר מול העולם, מי מפעיל את המערכת ומה הוא צריך לעשות כדי שהיא תפעל נכון.

- **Implementation** – לפני שלב היישום, בניית דיאגרמת אובייקטים לפני שיתחיל לרשום אותם, לאחר מכן נתחיל ליישם את המערכת (רשום את הקוד).
- **Testing** – לרשום תסրיטי בדיקה, בצורה מדויקת איך אנו הולכים לבדוק את המערכת ורק אחרי זה לבצע את הבדיקות.
- **Deployment of System** – הוצאה לפועל של המערכת.
- **Maintenance** – שלב התחזוקה של המערכת.



◦ יתרונות לגישה:

- מערכת מתועדת היטב, כלומר קיים דוקומנטציה טובה מאוד למערכת.
- תכנון ויישום של המערכת כולל, כלומר כמקרה אחד של המערכת בתכנון ובביצוע, מביאה לאינטגרציה פשוטה יותר וקלה יותר של רכיבים של המערכת.

◦ חסרונות לגישה:

- תכנון הטוב הוא טוב, אך אם בשלב מסוים מתגלה בעיה ברמות קודמות של המפל אז זו בעיה גדולה כי העבודה של השלב הנוכחי נעדרת וחזרים אחורה עד לשלב שבו התגלתה הבעיה, לאחר מכן ממשיכים עם השלבים במפל מאותו השלב שחררנו אליו.
- הלוקות אינם שותף לתהליכי בזמן אמת אלא רק כאשר העבודה על המוצר מסתיימת.

◦ **V-Model** – גישה דומה למודל המפל, כל שלב תלוי בשלב הקודם ותייעוד כל השלבים, ההבדל הוא שבגישה זו בכל אחד מהשלבים מתכננים בסימון אליו גם בדיקות.

◦ **Accepting Testing** – בדיקות קבלה אלו הבודקות שהלוקות או מישהו מטעמו צריך לעשות על המוצר, לא כמישהו שבודק את המערכת כולא או בודק בדיקות ביצועים אלא בתור משתמש קצה, לחיצה על כל הceptors

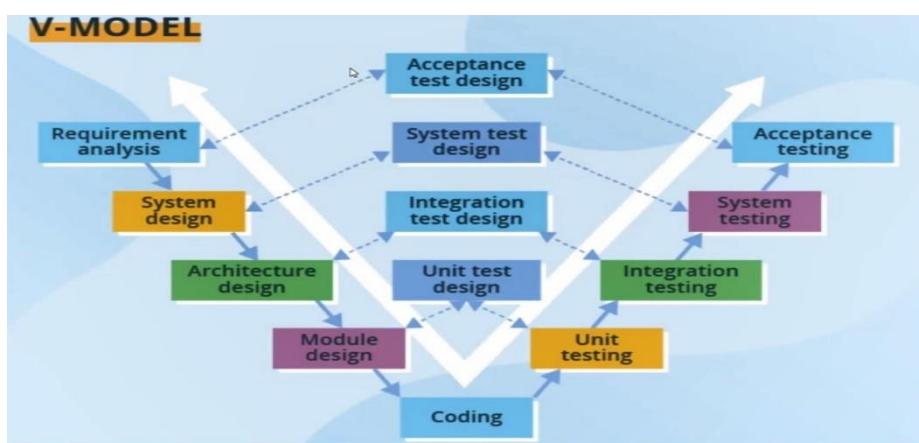
שים ולראות שהכל עובד כמו שימוש הkaza רוצה וכו', מסתכלים על הבדיקות האלו ברמת ה- Requirements בלבד, על כל אחת מהדרישות צריך להגיד תסיט בדיקה, נשים לב שכאשר אנו בשלב ה- Requirements אין באמת עדין מוצר, لكن אנחנו נקיים משיקולים אחרים שלא קשורים לשלב זהה.

.System Testing ○

.Integration Testing ○

.Unit Testing ○

נגע בכלל אלו כאשר נדבר על שלבי בדיקה.



יתרונות לגישה:

- Easy to understand and follow - מערכת מתועדת היטב, קלומר
- קיימ דוקומנטציה טובה מאוד למערכת.
- שלבי הבדיקה ברורים ומהותיים יותר.
- אוטם היתרונות של שיטת ה- WaterFall.

חסרונות לגישה:

- אוטם החסרונות של שיטת ה- WaterFall.

– מטרת השיטה האינקרמנטלית היא לשפר בעיות במודול Agile Model •

.WaterFall –

המניפסט של מודל ה- Agile :

- Individuals and interactions over processes and tools - מאמינים יותר באנשים (הצאות) מאשר שימוש בכלים אוטומטיים, צוותים מעורבים, קלומר צוות אחד גם של מפתחים וגם של בודקים וכו'.

- **Working Software over comprehensive documentation** – עדיפות תוכנה עובדת על פני תיעוד, ככלומר תראה ללקוח תוכנה עובדת ואז הוא ייתן לך פידבקים זה יותר טוב מדויקומנטציה מפורטת.
- **Customer Collaboration over contract negotiation** - אפשר לחתום על חוזה, אבל ניתן להיות אגילים לזה ולתת לו כל הזמן להיות מעורב בהתקדמות של הפרויקט ובסוף הוא יהיה יותר מרוצה.
- **Responding to Change over following a plan** – עדיף שניהיה מוכנים לשינויים מאשר שניצמד לתוכנית ולא נזוז ממנה.

איך עובדים בשיטת ה- Agile?

SCRUM – שם של צוותי הפיתוח בשיטה ה- Agile, צוותים אלו מכילים את כל סוג היכולות שצרכי כדי לבצע את המשימות שלנו (ארכיטקטים, בודקים, מפתחים וכו') בדרך כלל נع בין 9-4 אנשים, כל אחד מהאנשים הוצאות הוא קרייטי ולא ניתן לוותר על אף אחד מהם.

: Roles •

- **Product Owner (אחראי מוצר)** – הוא בעצם משקף את מה הלוקח רוצה וחושב על המוצר, הוא לא צריך לקחת משימות פיתוח.
מה התפקידים שלו?

1. מתעדף ביחד עם הלוקח איזה משימות כדאי לעשות כרגע מה- Backlog.
2. אם יש שאלות אז הוא ייצור קשר עם הלוקח ויבירר את הנושא.
3. אחראי על השגת היעד הפיננסי של הפרויקט.
4. בכל שלב שימושיים מטלה מסוימת מציגים את זה בפניו והוא מאשר את העבודה שנעשתה או נותן הערות לשיפור.

- **Development Team** – צוות הפיתוח שנקרא Scrum, הוגדר בסוף העמוד הקודם.

- **Scrum master** – הוא חלק מהאנשים שלוקחים משימות פיתוח. מה **התפקידים שלו?**

1. מייצג את צוות הפיתוח לפני חז.
2. שומר על רמה מסוימת של מקצועיות בתחום הוצאות שלו.
3. צריך לבדוק את הוצאות מהפרעות הקשורות במהלך הספרינט.

4. מקשר בין צוות ה- Scrum שלו לצוותי Scrum אחרים.
5. עוקב אחריו הוצאות וראה איך הם מישימים את המטלות שלהם.

- כל אלו הם הגדרות שמקבלים אנשים בצוות ה- Scrum, כל הגדרה כזו מוסיפה לאיש הוצאות אחראיות נוספות.

• מושגים חשובים:

- **Sprint** – זמן פיתוח, מוגדר בדרך כלל בין שבועיים לשלווה שלושה שבועות בהם הוצאות עובד על אישתו חלק מה מוצר. מה צריך להיות מוקן מסוף הסPRINT?

1. **דרישות (PRD)**.
2. **עיצוב.**
3. **אימפלמנטציה.**
4. **בדיקה.**
5. **אפשרות להציג ללקוח את החתיכה החדש מה מוצר.**



- **Product backlog** – רשימת כל המשימות שמרכיבות את המוצר.
- **Sprint backlog** – משימה אחת מרשימה המשימות, לוקחים אותה ועובדים עליה ב- Sprint אחד, ככלומר בסוף הסPRINT יש לנו חתיכה מהתוכנה שעבדת.

• Meetings – פגישות כחלק משיתת ה- Agile.

- **Sprint Planning** – הפגישה הראשונה, ישיבה שבה יושבים צוותי ה- Scrum ומתקנים את ה- Sprint שלהם, בוחרים משימה מה- Backlog Product לפי מה המשימה hei חשובה כרגע, או לפי המשימה שהוצאות יכול להתחייב שישים ושוברים אותה ל- Tasksanesים Chatikot קטנות של משימת הסPRINT (Sprint backlog), משיכים אנשים למשימות, 4 שעות בערך.

- **The Daily Scrum** – ישיבה קצרה של צוות ה- Scrum שנעה בין 15-10 דקות ומתקיימת כל יום.

מה צריך להיכל בפגישה?

1. על איזה משימה מתווך ה- Spring Backlog עבדתי ביום הקודם.
2. על איזה משימה אני עובד היום.

3. האם יש לי איזושהי בעיה והאם אני צריך עזרה מארגוני הוצאות, אם זה יקח יותר מ- 5 דקות להסביר את הבעיה אז צריך להגיד את זה לא חלק מה- Daily Scrum.

- The Sprint Review – ישיבה שקוראת פעם אחת בסוף ה- Sprint של בער שעתים, בישיבה זו נתונים פידבקים מ Każעווים, מה עבד טוב, מה לא עבד טוב, איזה משימות עשו ואיזה לא וכו'.

- The Sprint Retrospective Meeting – ישיבה שהמטרה היא לנסות ולדבר על דברים שהם יותר מנהלתיים, למשל לא עבד טוב החיבור בין אנשי הוצאות וכו'.



הרצאה 6

- **שלב מידול המערכת: System Modeling** – בשלב זה אנו נרצה להציג את המערכת שלנו כדיגרמה מסויימת.
 - **ישנים 4 סוגים יסודות ב- Modeling:**
- .1 – **Context Modeling** – מעنين אותנו המבנה הכללי של המערכת שלנו אל מול מערכות אחרות.
 - .2 – **Interaction Modeling** – מעنين אותנו איזה מין אינטראקציה יש עם המערכת שלי, מה מפעיל את המערכת שלי מבחן או מבפנים בין מודולים מסוימים.
 - .3 – **Structure Modeling** – מעنين אותנו מה הוא מבנה המערכת, כלומר נרצה לקבל דיאגרמה של מבנה המערכת.
 - .4 – **Behavior Modeling** – מעنين אותנו מה התנהגות של המערכת שלנו, איך היא מגיבה לאיורים שקיימים במערכת או איך היא מגיבה למידע שנכנס למערכת.

כל הדיאגרמות האלו מאוגדים בשם שנקרא

(UML) Unified Modeling language

- ישנו 3 סיבות עיקריות למידול המערכת:

1. זו דרך לעורר דיון הרבה יותר מڪצועי בקשר למערכת.
2. זו דרך נוספת לייצר תיעוד של המערכת עבור שלבים שהמערכת כבר הושקה.
3. זו דרך לייצר תיעוד בrama של פירוט עבור איך אנחנו הולכים לפתח את המערכת.

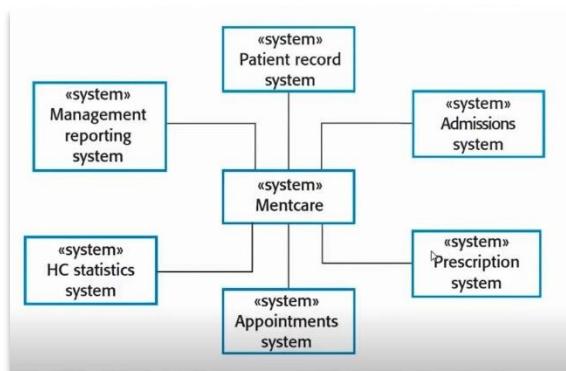
שלושת הסיבות הללו הן המוטיבציה שבגללה אנחנו רוצים לבצע מידול של המערכת שלנו.

- **Context Modeling** – המערכת שלם לעומת מערכות אחרות.

ניצג בעזרת הדיאגרמה הבאה:

○ **Activity Diagram**

Context Model



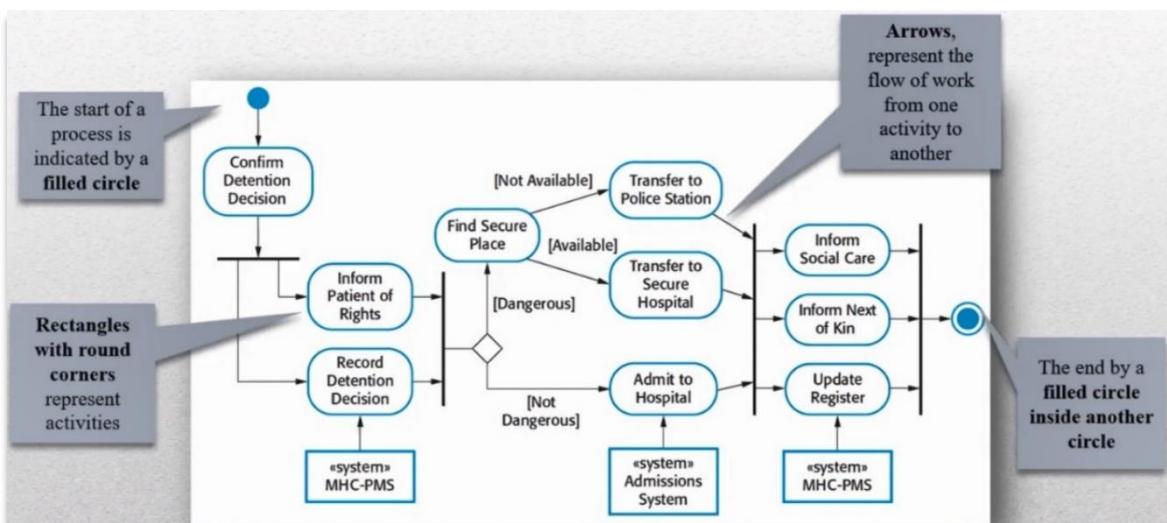
הכי High Level שיכל להיות. ←

لدיאגרמה זו מתלווה הדיאגרמה

התחטונה שיורדת יותר

לפרטים.

Activity Diagram



• **User** – נרצה לראות אינטראקציות עם המערכת שלנו, **Interaction Modeling**
 מול המערכת שלן או מערכת אחרת מול המערכת שלן.
 נציג מעזרת הדיאגרמות הבאות:

- **Use Case Diagram**
- **Sequence Diagram** - לאחר שאפיינו איזה cases use נמצאים לנו במערכת
 היינו רוצים לפרט כל אחד מהם, זה מה שהדיאגרמה הזו עשו.

- Use Case Diagram .1

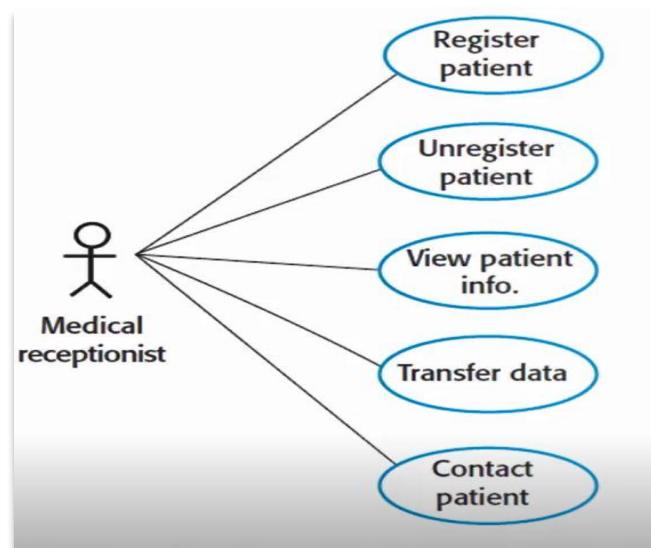
הדיאגרמה הבאה מציגה דיבור בין מערכות:

המערכת שמשתתפת באותה אינטראקציה → **אינטראקציה** → **המערכת שלנו**.



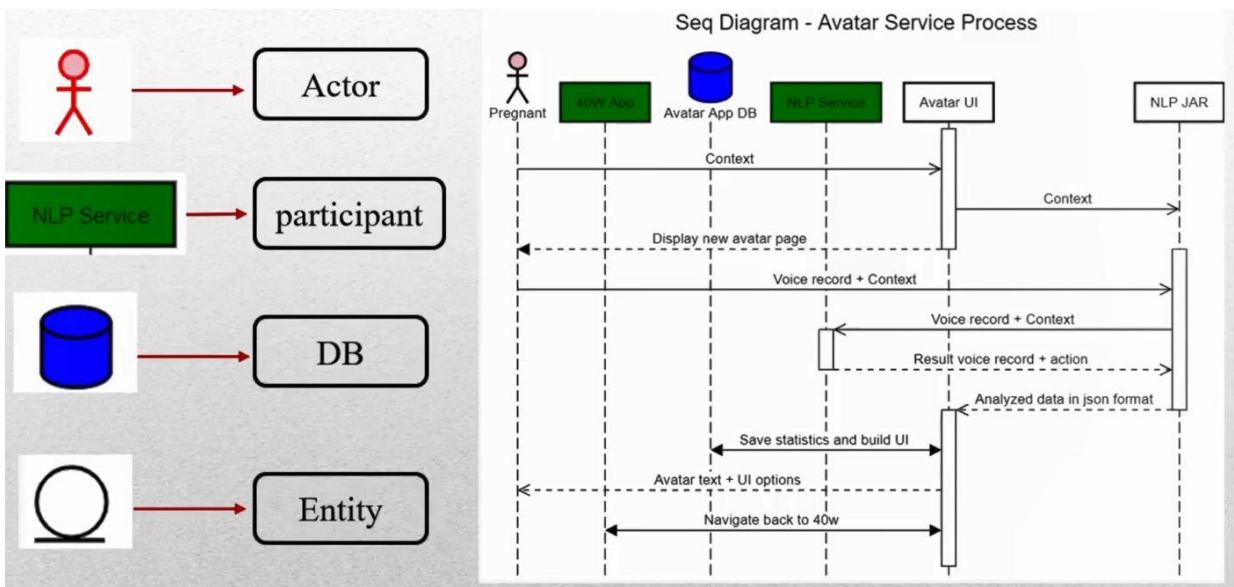
- Use Case Diagram .2

אם זה מתייחס ונגמר רק בתחום המערכת שלנו אז נראה רק את הדיאגרמה הבאה
 שמציגה רק את ה- Use Cases שבתוך המערכת שלנו, כלומר איזה פעולות קיימות
 בתוכה.



– Sequence Diagram

באה לאפיין אינטראקציה אל מול המערכת שלנו.



הרצאה 7

– Structure Modeling •

נייצג בעזרת הדיאגרמה הבאה:

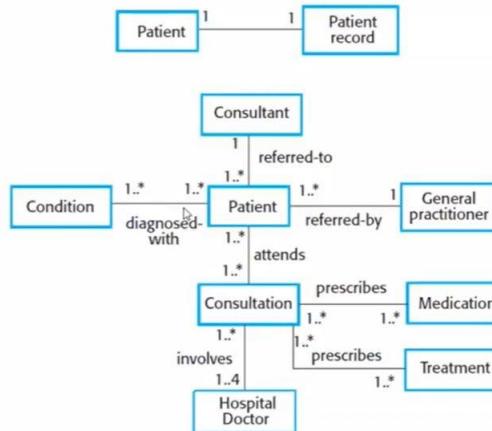
– Class Diagram ○

- יכול לייצג את סידור המערכת מבchinית וርכיבים שמייצגים את המערכת ואת הקשר ביניהם.
- יכול לייצג את המערכת כמודל סטטי שמציג את הסדר שבה המערכת רצתה.
- יכול לייצג את המערכת לפי הסדר הדינامي של חוטים במערכת.
- משתמשים בו כאשר אנו מייצרים מערכות שהיא Object Oriented CD' להציג את הקלאסים במערכת ואת הקשר ביניהם.

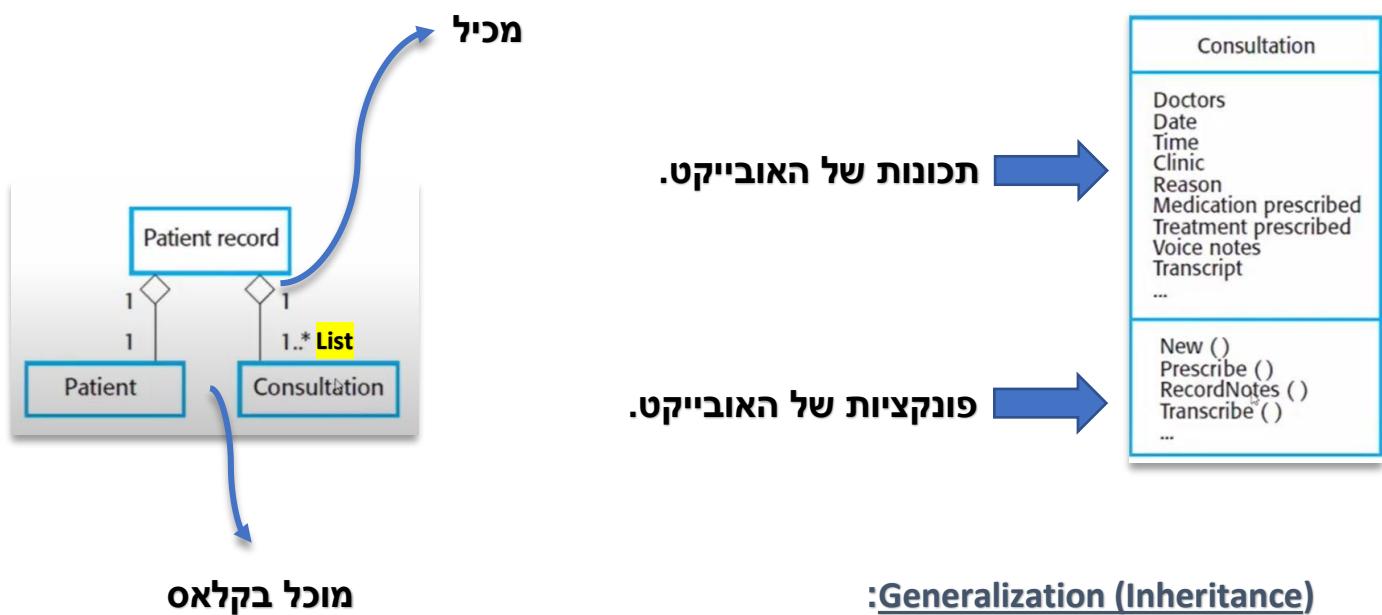
מייצרים את הדיאגרמה הזו כאשר אנו מדברים על עיצוב ועל הארכיטקטורה של המערכת.

Composition

- This figure show that objects of class Patient are also involved in relationships with a number of other classes.
- In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists.

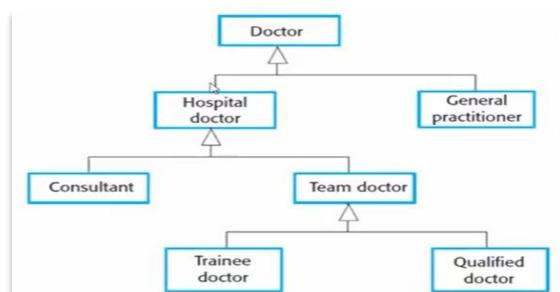


- You can define that an **exact number** of objects are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association.
- For example, the (1..*) multiplicity in that figure on the relationship between Patient and Condition shows that a patient may suffer from several conditions and that the same condition may be associated with several patients.



Patient record

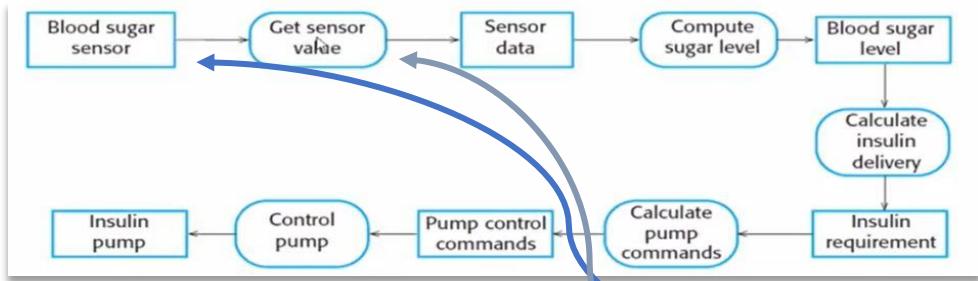
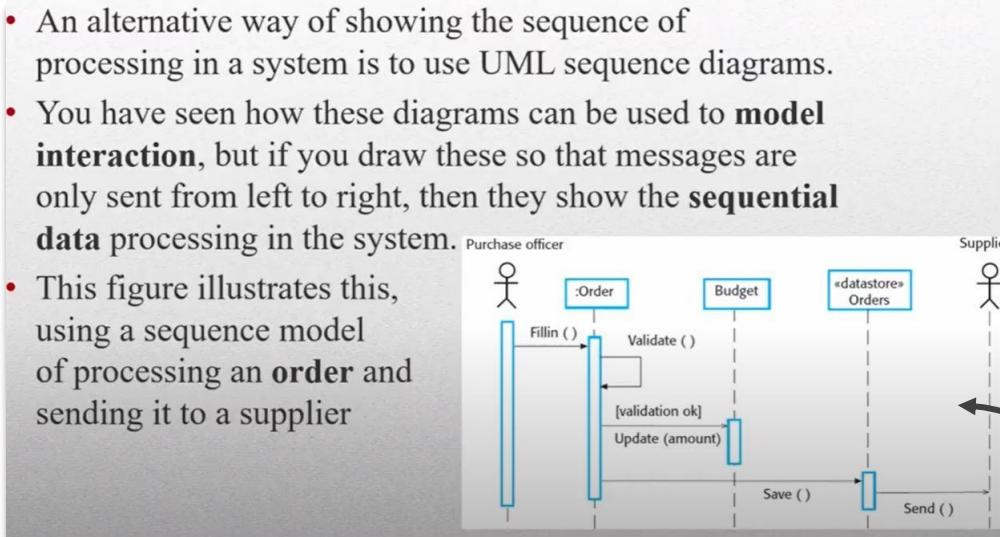
Generalization (Inheritance)



– לא לבחן. (כי זה דיאגרמה פחות נפוצה היום).

- **מתייחס למידע שעובר במערכת אויר המידע זהה מפעוף בין החלקים, בנוסף מתייחס לאירועים במערכת.**

- An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams.
- You have seen how these diagrams can be used to **model interaction**, but if you draw these so that messages are only sent from left to right, then they show the **sequential data processing** in the system.
- This figure illustrates this, using a sequence model of processing an **order** and sending it to a supplier



מערכות

סוגי מידע שעוברם

אירועים

- Architectural Design •

- שלב הארכיטקטורה, זהו שלב שבו מחליטים האם לבצע Re-use ל功德 מסויים או לא, אלו החלטות שיכولات להשפיע על המערכת שלנו.
- זהו שלב הראשון בעיצוב המערכת.
 - זהו שלב קרייטי המחבר בין עיצוב המערכת שלב האימפלמנטציה, שלב בו מזהים את הקשר בין שניהם.
 - **הפלט** של שלב זה הוא מודל ארכיטקטוני המתאר איך המערכת מאורגנת כסט של רכיבים שמתקשרים ביניהם לבין עצם.
 - יש צורך לעשות עיצוב ארכיטקטוני שהוא מיקרו ואחד שהוא מאקרו, כי יכול להיותשמי שקורא את זה לא מבין מה זה המודל המפורט ואולי יעזר לו לראות את זה מ- High level, הם נקראים כך:
- **Architecture in the small** – להסתכל על הארכיטקטורה של תוכניות אינדיבידואליות, איך רכיבים במערכת של' מדברות אחד עם השניה וכו'.
 - **Architecture in the large** – להסתכל על המערכת של' אל מול מערכות אחרות.

- Architectural Views •

- **A logical view** – נרצה לרדת לפרטים יותר נמוכים למשל מהם המחלקות שיהיו במערכת כיחידה לוגית, למשל מה התפקיד של המחלקה זו והמחלקה ההיא וכו'.
- **A physical view** – החובה של' בתור ארכיטקט להסתכל על החומרה שאנו צריכים כדי לפתח את המערכת של'.
- **A development view** – הסתכלות של' על המערכת ברמה של איזה קומפוננטות יהיה במערכת של', למשל האם צריך אישר רכיב AI שצריך להיות במערכת שלנו כדי לדעת האם יש צורך לאמן את המתכנתים שלנו, בעיקר כדי להיות מוכנים יותר טוב לרמה המקצועית הדרושה לבניית המערכת וכו'.
- **A process view** – מתייחסים לדרישות הלא פונקציונליות, למשל מה הפלטformance של המערכת מבחן (Run-time) ? כלומר כמה זמן עד שהמערכת מגיבה לפעולה מסוימת, כמה משתמשים יכולים להשתמש במערכת בו זמן ו איך לבצע את זה וכו'.
- **An abstract view** – נרצה לקחת החלטות בנושא של Re-use.

• **Architectural Patterns** – נרצה לעשות שימוש בידע קיים, תבנית

ארქיטקטונית היא תבנית שמודרגת היבט, כמו למשל מבנים קוד שאם אני לא משתמש בהם אז המערכת שלנו תהיה הרבה פחות טובה.

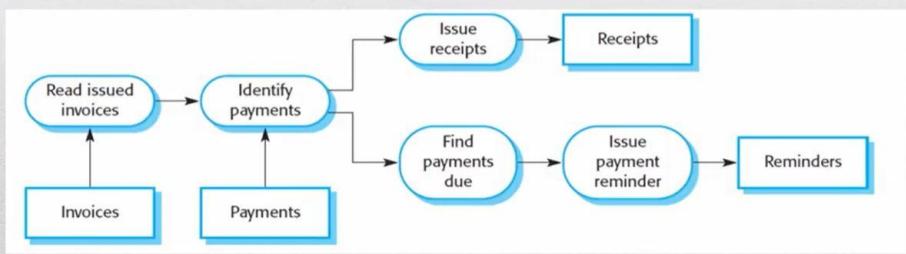
נגדיר זאת כפתרונות לביעות מוכחות מעולם התוכנה (פתרונות אופטימליים). אם נדע להתאים את התבנית לבעה שלנו אז המערכת שלנו תהיה הרבה יותר טובה מאשר אם לא נצליח.

○ **Pipe and Filter Architecture** – יש מערכות שכאשר אנחנו מקבלים

קלט למערכת שלנו וזה מתחילה לעשות עיבוד לקלט זהה, בכל שלב בתהליך עושים לקלט איזשהו העשרה, כמו מרוחקים לו מידע לאורך כל הדרך או שמשתמשו מבנים שהקלט הזה לא תקין ועשויים לו סיכון וזורקים אותו החוצה, ניתן לחשב על זה כמו על צינור עם יציאות, כאשר קטע קיבל מידע אז ניתן להמשיך למקטע הבא בו נאגור מידע נוסף שיעזר לנו להמשיך למקטע הבא אחריו.

Pipe and filter architecture

- Pipe and filter systems are best suited to batch processing systems and embedded systems where there is limited user interaction. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed.
- While simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections. It is difficult to implement this as a sequential stream that conforms to the pipe and filter model



הרצאה 9

שכיחה במערכות Web, העיקרון – **MVC (Model View Controller)** ○
החשיבות ביותר זה ליצור הפרדה מוחלטת בין השכבות ב-MVC, כלומר שאות
את מהשכבות האלה לא מכירה את השכבות האחרות ולא מכירה את
הлогיקה אחת של השניה, **הן מדברות אחת עם השניה באמצעות API (Application Programming Interface)**

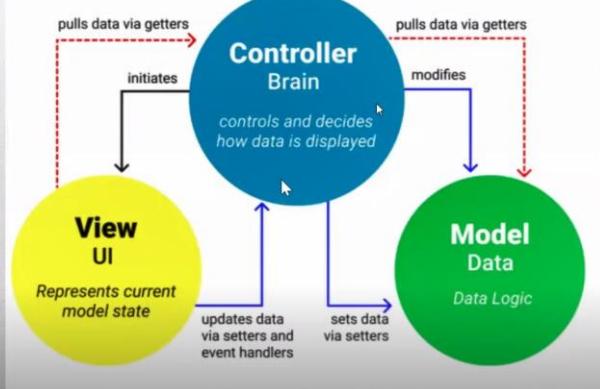
- ה- **Controller Brain** הוא כמו מרכזיה, מעביר מידע מה- **VIEW** ל- **Model Data** והפך.

- תפקידו של ה- **Model Data** הוא להביא מידע.
- תפקידו של ה- **View** הוא לשקוף את ה- **User Interface** למשתמש.

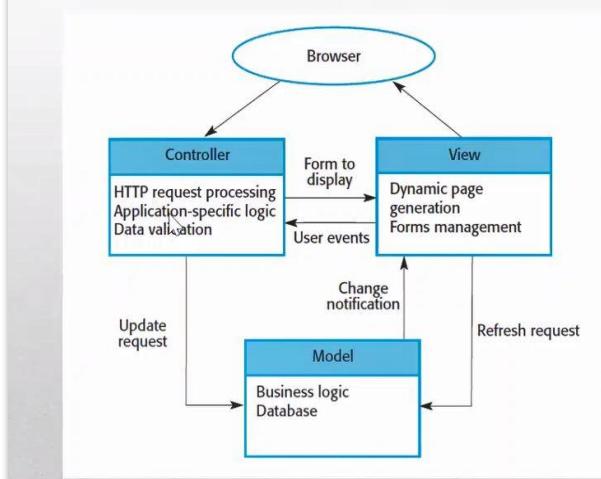
- ✓ היתרונו בתכנון זהה הוא שאם נרצה לשנות את ה- **View** למשל, אז לא נצטרך לשנות את ה- **controller** או את ה- **Model data**, ויש פה חיסכון של הרבה מאוד כסף לחברה ויכולת להתחדש בו- וו.
- ✓ אם נרצה לפתח מערכת מ- 0 ויש לנו את החלוקת הבורווה הזה אז אפשר לתכנן את ה- **View** בלי שייהי לנו **- Controller** ו- **Model data** כי אנחנו כתבים רק את המ██ים, וזה נכון גם עבור כתיבה של ה- **Controller** או ה- **Model data** בפרד.

MVC New

MVC Architecture Pattern



Architectural patterns - MVC (Old)

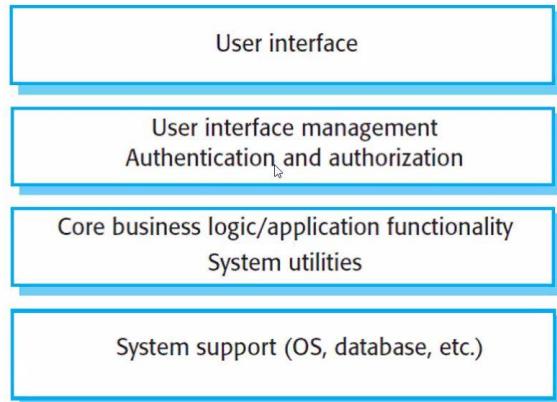


• **Layered Architecture Pattern** – עיקרון הפרדת השכבות, MVC הוא מקרה

פרטי של זה.

- שכבות שיש להן תפקיד מוגזם מאוד במערכת כאשר כל שכבה מדברת עם שכבה אחרת באמצעות API.
- אפשר למערכת להיות יותר גמישה לשינויים ותיקונים.

- This pattern is shown in below figure here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

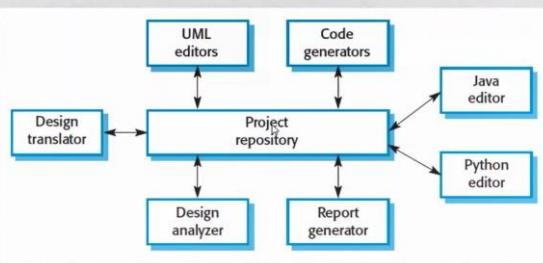


• **Repository Architecture** – ארכיטקטורה שבביס שלה יש לנו Repository שמננו אנו רוצים לספק יכולות מעל המידע עצמו, כל היכולות שיכולה לעניין את המשתמש.

- **Repository** – שם כללי למקום שנשמר בו מידע.
- **חסרון בארכיטקטורה** - זה מיועד לאפליקציות Stand-Alone

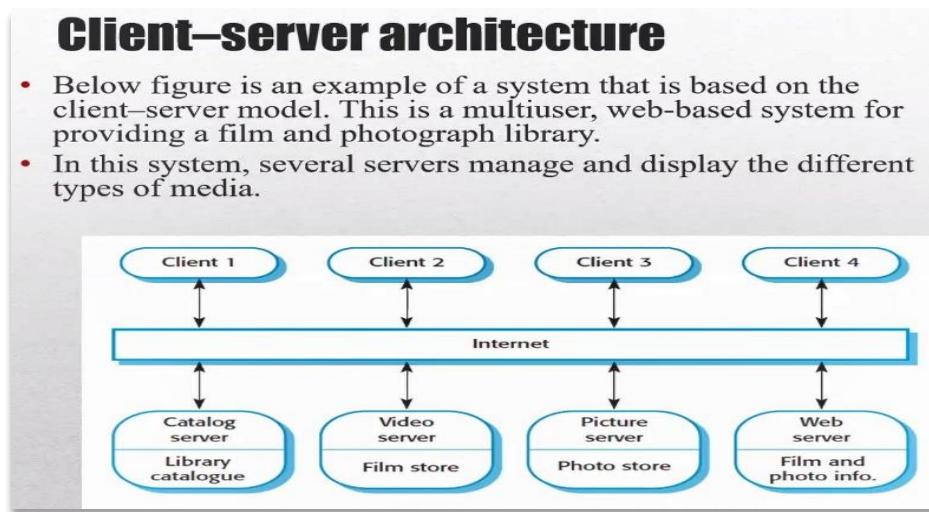
Repository architecture

- Below figure illustrates a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development.
- The repository in this case might be a version-controlled environment that keeps track of changes to software and allows rollback to earlier versions.



Client-Server Architecture • ארכיטקטורה של Client-Server מבוססת על האינטרנט.

- משתמשים באינטרנט כדי להנגיש תוכן למשתמש.
- כל Server יוצר רק שירות אחד, אם אתה יוצר יותר שירות אחד ב- Server אז אתה נגד את הארכיטקטורה הזאת.



הרצאה 10

Application Architectures • ניסיון לחלק את הממערכות לשני קבוצות:

1. Transaction Processing Applications – מערכות דedata ביבס, כלומר

מבצעות פעולות על גבי איזשהו בסיס נתונים, נקרא כך כי כאשר מדברים על DATA ביבס אז מדברים על טרנסקציות (מבצעים טרנסקציות בDATA ביבס).
○ Transaction system – נמצא תחת הקטגוריה של Information system, Web system application, processing Application, Web system application, לרוב זה יהיה processing Application.
כלומר רוב הפעולות שלו תבוצע על גבי ה- Web.

2. Language Processing Systems – מערכות שעושות איזשו עיבוד על המידע והופכים את המידע מצורה מסוימת לצורה אחרת, למשל קומפיאילר או NLP (Natural Language Processing)

הרצאה 11

• - Design and Implementation

זהו השלב בו אנו הופכים את התכנון שלנו לקוד, כלומר שלב הביצוע.

• – Object Oriented Design

- פה אנו מדברים על יחסים בין אובייקטים.
- כדי לפתח ב- Object Oriented אנו צריכים שפה שתומכת Oriented.
- גישה ישרה לאובייקט יכולה להתבצע אך ורק מתוך האובייקט עצמו ולא מחוץ לו.
- כאשר יש לנו מערכת שרצה אז בעצם אובייקטים נוצרים באופן דינמי מההגדרה של הקלאס.
- לכל אובייקט יש דאטה (List וכו') ויש פעולות (פונקציות) שבאמצעותם ניתן לבצע מניפולציות על הדאטה, וכן ניתן לחשב עליהם (הקלאסים) ולשנות אותם בתור ישות בפני עצמה.

– עיקנון ראשון וחשיבותו בשלב זה

- כאשר אנו רוצים לדעת אם כתבנו את המערכת בצורה טובה אז חשוב על אם אנו רוצים להרחב את המערכת או לשנות אותה ויש לנו הרבה מאד עבודה כדי לעשות את זה או אם יש לנו הרבה מאד בעיות כדי לשנות אותה אז כתבנו מערכת בצורה לא טובה, אחרת, כאמור אם אנו יכולים לבצע את זה בצורה קלה ומהירה אז כתבנו את המערכת בצורה טובה.
- אסור לשנות באובייקט מסוים ישפיע על אובייקטים אחרים.

• System Context and Interaction – כאשר אנו מאפיינים את המחלקות שלנו

נחשב על שני דברים:

- מהו המבנה של אותה מחלוקת.
- Aiזה שימוש המחלוקת עושה בפונקציה של מחלוקת אחרת (למשל בירושה בין מחלוקות), בנוסף נחשב על הכליה של מחלוקת מסוימת במחלוקת אחרת ומדובר נרצה לעשות זאת.

ישנם 3 גישות לאר לעשות את העיצוב הזה של המחלקות שלנו:

1. **Grammatical Analysis** – נספר לעצמנו סיפור על איך המערכת שלנו

אמורה להיכתב כדי להבין איזה תהליכי יש לנו במערכת.

2. **Tangible Entities**

3. **Scenario-Based Analysis** – נחשב על תסרים בתוך המערכת

שלנו כדי להבין מין ישוות נמצאות במערכת שלנו.

• **Object Class Identification** – שלב בו אנו מגדירים מה המשתנים ומה הfonקציות

שיש לנו בכל קלאס וקלאס.

• **Interface Specification** – כל' חשוב בשלב הפיתוח, מאוד נרצה להימדד לשינויים

שנקראות **Interface**, כולל חתימה של מה השיטה (fonקציה) מחזירה או מקבלת,

כך אנו קובעים איך המערכת שלנו תתנהג, ניתן לעשות זאת באמצעות מחלקות

אבסטרקטיות.

הרצאה 12

• **Design Patterns** – ישם 4 Design Patterns בשלב ה- **Implementation** וهم:

○ **Observer Pattern** – מרכיב משני שכבות, **Subject** ו- **Observers** כל אחת

מהשכבות מורכבת משני חלקות או יותר.

הבעיה שאotta תבנית זו באה לפטור זו בעיה של היתר רוצה להאזין לתוך המערכת של לאיושה מחלוקת (אובייקט) שמסיבה כלשהי האובייקט הזה

משתנה (בעיקר המידע הפנימי שלו כולם) – "אינסטנסס" שלו או ניתן

להגדיר זאת כ- "המצב של האובייקט", הצגה של מידע פנימי של

האובייקט משתנה בצורה גרפית כלשהי, למשל גרפ עמודות או בצורה

פא.

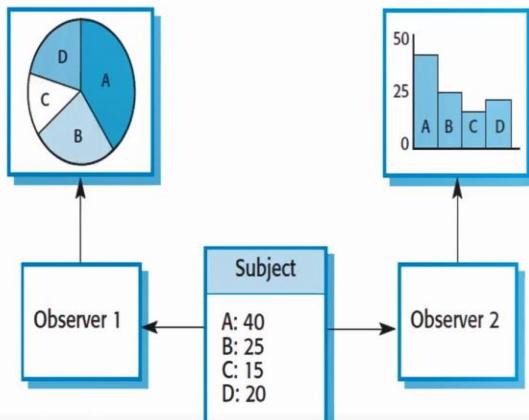
נרצה לקבל עדכון על שינוי כלשהו באובייקט כדי שנוכל להגיב בצורה בה היינו רוצים.

האובייקט המשתנה נקרא **Subject** והאובייקטים שרצו להאזין לאותו שינוי נקראים **Observers**, יכול להיות יותר אחד כזה.

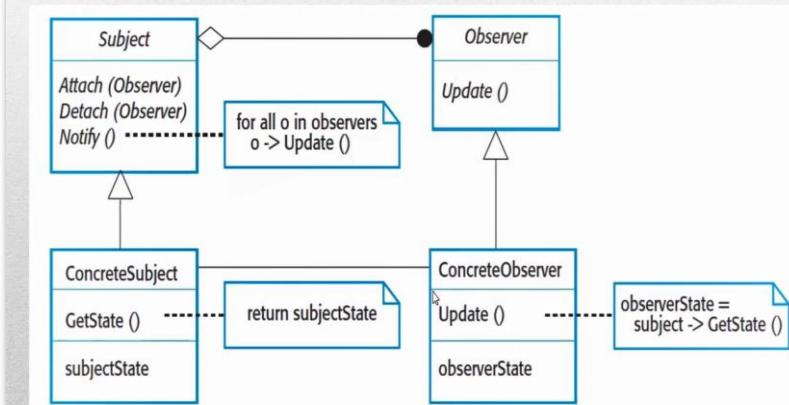
נרצה בעצם להציג את המידע בצורה שונות כדי לפלח את המידע בצורה שונות, אם יש סוג אחד של **Subject** ונרצה להציג את המידע ביותר מצורה אחת אך נציג **Observer** לכל הצגה.

Observer Pattern - Example

Two different graphical presentations of the same dataset



Observer Pattern - UML

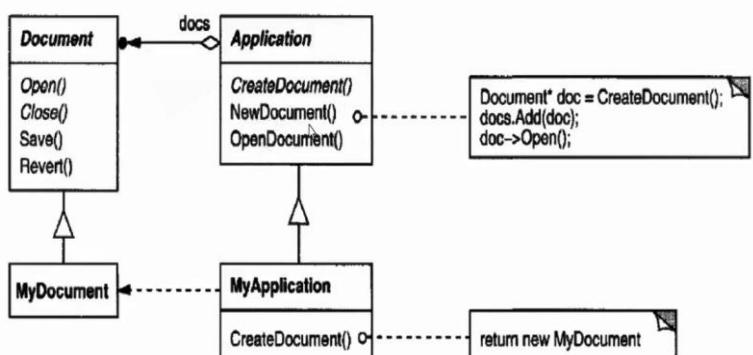


- **Observer** מכיל **Subject**
- **Subject** מכיל יותר מ- **Observer** אחד (כלומר רשיימה של **Observers**).
- **Subject** יורש מ- **Concrete Subject**
- **Observer** יורש מ- **Concrete Observer**
- קו ישר זה סוג של הפעלה.

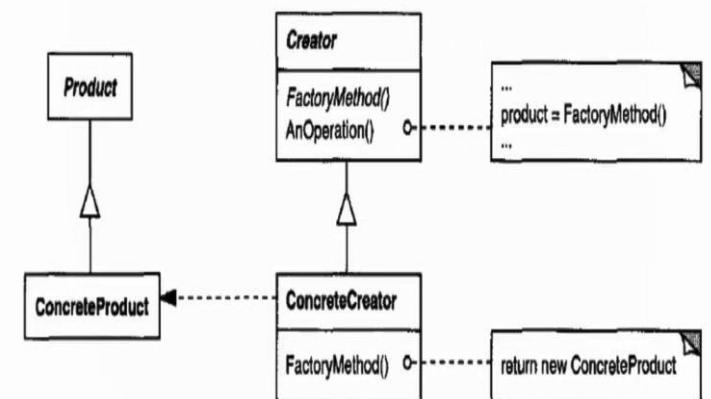
— תבנית שמרתת ליצר אובייקטים בזמן ריצה, **Factory Method Pattern** ○
 כלומר ליצור אובייקטים בזמן ריצה על פי הדרישה של המשתמש, כולל אובייקטים מסווגים שונים.
Product Creator - התבנית כוללת שני שכבות,
Creator יש פונקציה שנקראת **FactoryMethod()** שמייצא אובייקט שנקרא **Product**.
 ישנה מחלוקת **Concrete Creator** שבה יוצרים **Concrete Product** הוא איזשהו מחלקה אבסטרקטית או אינטראפיס ומילוי שירש ממנו **Concrete Product** זה
 כך אנו מאפשרים למשתמש ליצור כל פעם אובייקט ספציפי שהוא בוחר.

Factory Method Pattern - Example

It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.



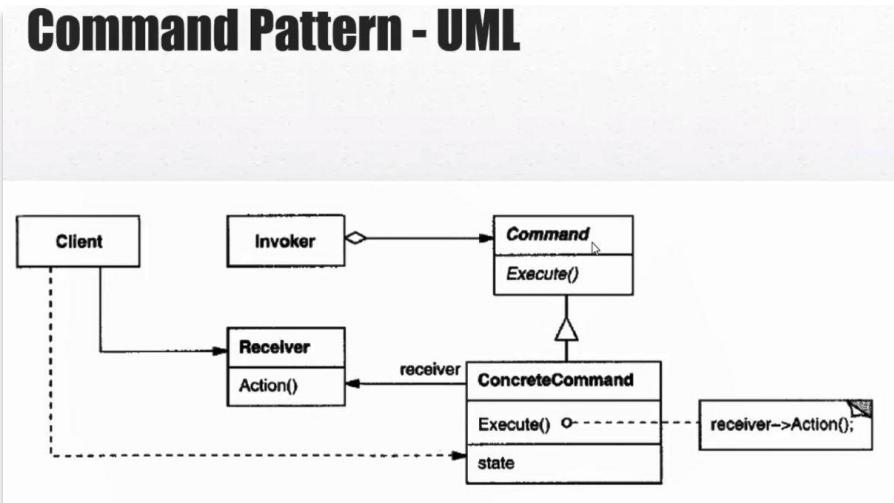
Factory Method Pattern - UML



- בדוגמה- ישנה מערכת שיעודת לייצר קבצים: **CreateDocument** היא המתודה של ה- **Factory** שמחזירה **Document**, ברגע שבפונקציה מחזירה **Document** אז זה יכול להיות כל **Document** ולא קובץ ספציפי אחד.

- – תבנית שבאה לפתור את הבעיה שבה אנו רוצים להסתייר איזושהי פעולה שבנינו מהמשתמש (תוכניות וכו'), מסתירים בעצם את האימפלמנטציה של אותה פעולה וחושף לך רק את זה שאתה מקבל איזשהו **Command** (פקודה).

Command Pattern - UML



- – תבנית שבאה לפתור את הבעיה שבה אנו רוצים שהמשתמש יוכל לייצר רק אובייקט אחד מחלוקת מסוימת, דיברנו על זה באחת הרצאות הקודמות.

Class Singleton:



- – אחת התבניות הידועות, ***לא למחוק***.

▪ מה צריך להיות לכל **Design Pattern** –

1. השם שלו בעצם צריך להיות בעל משמעות.
2. צריך להיות לו הגדרה של מה הוא הולך לפתור.
3. צריך להיות לך דרך לתאר את הפתרון.
4. הסבר של מה הוביל לבעה הזו וכייך אנו הולכים לפתור אותה.

הרצאה 13

Implementation issues • נושא מרכזיים בשלהי היישום

ומהם היכולות שלנו להתמודד איתם. נדבר על 3 בעיות או נושאים מרכזיים וهم:

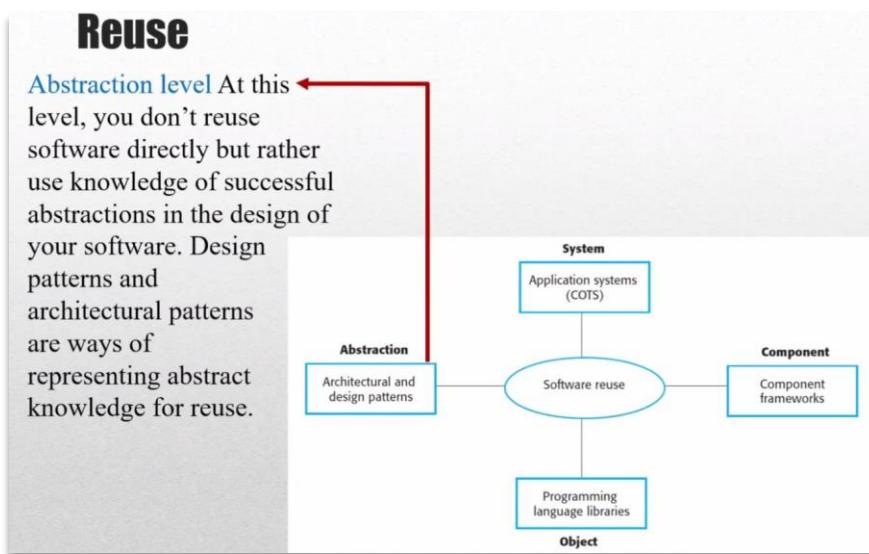
- Re-use
- Configuration management
- Host-target Development

1. – בין השנים 1990-1960 אנשים פיתחו תוכנה מאפס ולא היה אפשר לעשות use Re-use כי לא היה פלטפורמות לשיטוף קוד, היום המצב שונה ונרצה להשתמש בקוד שכבר כתבו ואפילו נאמר **חייב**.
יתרונות לפיתוח מוצר מהפץ זה שהוא לא צריך להתאים את עצמנו לקוד כתוב. בנוסף נראה שבמהלך כתיבת המוצר אנו עושים Re-use גם בלי לשים לב, למשל בכתיבת פונקציות.

ישנים 4 רמות ל- Re-use וهم:

Design level – לעשות Re-use לידע, לדוגמה שימוש ב- Abstraction level ○

ספציפי הוא שימוש חוזר ברמת הידע.



בכל שלבים הבאים יהיה ממש use-use Re-בקוד:

- **Object Level** – עושים שימוש בקוד בرمאה של אובייקט או בספרייה, למשל אם נרצה לשלוח מיילים אז נשתמש בספרייה Java-mail כדי לשלוח מיילים, נדרש לשלוח מייל שהוא רלוונטי למערכת שלנו אך אנו לא צריכים לכתוב את התשתיות כדי לשלוח מיילים. זהה, חוסך הרבהocabi ראש.
- **Component Level** – שימוש בהרבה יותר מספרייה, למשל הרבה ספריות או יכולת מורכבת כמו מנגנון של הרשות והזדהות. יש המון קוד שרשום ב- Open source שמאפשר לנו להמשיכן יכולות שכבר כתבו.
- **Application System** – משך לקיחת מוצר שלם ושימוש בו בתוך המערכת שלנו למשל מוצר שלם של מערכת בנקאות שימושים בה כדי לקבל שירותים בנקאים דרך האינטרנט.

▪ יתרונות ל- Re-use :

- חוסך המון זמן.
- מונע באגים, הקוד יהיה הרבה יותר אמין, יציב וחוסך בעליות. אך כל זה רק במידע ונעשה שימוש נכון בקוד.

▪ חסרונות ל- Re-use :

- דרוש זמן כדי לעשות אדפטציה לקוד זהה.
- דרוש זמן ללמידה של המוצר שאנו הולכים להטמייע במערכת שלנו.
- דרוש זמן כדי לבצע מחקר כדי למצוא את אותו סגמנט של רוד שאנו רוצים להשתמש בו.

ברוב המקרים נרצה לעשות Re-use.

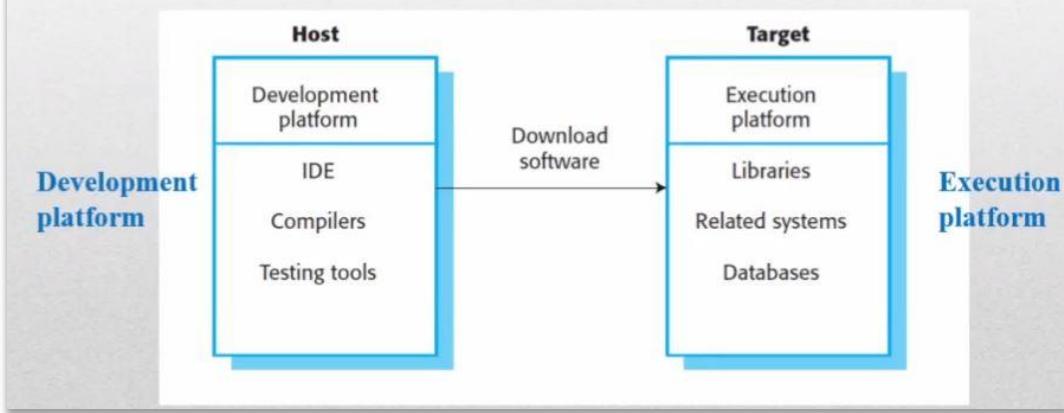
2. Configuration management – ניגע בזה בהמשך.

3. **Host-target Development** – הבעיה בנושא זהה היא שהיכולת שלנו לפתח מההו ולבדק אותו בתוך החברה שלנו היא שונה מהמצב בו נרים את התוכנה שלנו במחשב של הלוקוט. זאת מכיוון שיכל להיות שיכל להיות קונפיגורציות

שונות בין המחשבים והמערכות שאנו משתמשים בהם בתוך החברה לבין הקונפיגורציות של הלוקו, או למשל שימוש במערכות הפעלה שונות וכו'.

Host-target development

Most professional software development is based on a host-target model. Software is developed on one computer (the host) but runs on a separate machine (the target).



פתרונות לבעה זו:

א. פתרון WEB-י, כלומר לאנדרויד התקינה במכשיר של הלוקו אלא יוצר מוצר שהוא WEB-י לחילוטין שמותקן בתוך השרת שלנו והлокו יתחבר אליו רק דרך ה- Browser.

ב. באמצעות סימולטוריים, כמו לינוקס או וינדואס או איזושהי תוכנה (מכシリים) ולבסוף שם את התוכנה שלנו, כלומר על פלטפורמות שונות. בכך אנו מודדים את הסיכון לכך שה מוצר שלנו יגע לאיזושהי פלטפורמה שלא בדקנו בה את המוצר שלנו, בכך אנו מודדים את הסיכון שלנו לביעות עם המוצר בפלטפורמות שונות.

ג. שימוש במכונות ווירטואליות כמו VM שדיברנו עליו בעמוד 4.

ד. לעשות הרבה בדיקות.

ה. פתרון נוסף וחשוב הוא שימוש ב- IDE- Integrated Development Environment.

Integrated Development Environment (IDE)

- An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface.
- Generally, IDEs are created to support development in a specific programming language such as Java.
- The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.
- A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together.

– זו בעצם גישה בה מפרסמים את הקוד מקור של

ה מוצר שלנו כמו שהוא מבלי להציגו לנו כדי שייהי אפשר ל千古ת אותו כמו שהוא
ולהשתמש בו, או ל千古ת אותו ולשנות אותו. בעצם הקוד מקור יהיה נגיש לכל הקהילה
המקושרת ל千古וד המקור הזה ולהשתמש בו או לעדכן אותו.
מתקשר ל- Re-use .

ישנם פלטפורמות כמו GitHub שמאפשרות לקהילת ה- Open source לגדול.
היום ישנים המון מוצרים שכותבים ב- open source .
ה מוצר הראשון שמעצם גרם לכל השינוי הזה היא מערכת הפעלה Linux, היררכיות
של הקהילה גרמה למערכת הפעלה זו להיות טובה ואמינה לשימוש.



.MySQL ,Eclipse ,Java Open source

למה להשתמש ב- Open source ?

- חשיפה לכהילה.
 - משתמשים יכולים לדוח על באגים ובערך לגרום למוצר להיות טוב יותר (ניתן לחשב על זה בתור QA בחינם).
 - משתמשים יכולים לשפר את הקוד ולהרחיב את המוצר הקיים.
- ישנם כל מיני הסכמים מסחריים ב- open source שcadai להכיר שקיימים.

הרצאה 14

בדיקות – ניגע בכמה נושאים בכל הקשור לבדיקות המערכת והם:

- הבנת בדיקות בשלבים שונים החל תיקון תוך פיתוח לבדיקות קבלה ובדיקות משתמש
- הצגת שיטת לבחירת מקרי בדיקה לזריהו תקלות בפיתוח
- הבנת נושא TDD - Test Driven Development
- הכרה של שלושת סוגי הבדיקות – בדיקות יחידה (unit), בדיקות מערכת (system testing), ובדיקות גירסה (release testing).
- הבחנה בין בדיקות פיתוח לבדיקות משתמש

מטרות הבדיקות:

- להוכיח כי המערכת מבצעת את כל תפקודיה לפי מפרט הדרישות
- יש להגדיר בדיקה אחת לפחות לכל אחת מהדרישות כדי שהן מופיעות במפרט הדרישות של המערכת
- מומלץ להגדיר בדיקות שיבדקו שילובים של מספר דרישות כדי לוודא שאין תקלות שנובעות משילובים אלה
- למצוא קלטים או תסרים שבهم המערכת לא מבצעת את מה שמוגדר עבורה ככל מרغTAGיות או לא רצויות של המערכת.
- מטרות הבדיקות הינה לזהות התנהגויות אלה ולשרש אותן (ז"א שייהו חלק מתסרייטי הבדיקה גם בהמשך).
- סוג ההתנהגויות שהן במיוחד בעיתיות הקשורות לנפילות (crash) או אינטראקציה שגوية עם נתן מערכות חיצונית אחרות

TDD (Test Driven Development) – גישה שאומרת שהבדיקות צרכות להתבצע במקביל לשלב האימפלמנטציה (שלב היישום), מתבצע לרוב בשיטת פיתוח AGILE פונקציית הבדיקה נכתבות במקביל לכתיבה הקוד, באופן הטהור הבדיקה נכתבת לפני פונקציית הביצוע.

חרוגנות

- מחייב קיומו של כלי אוטומטי לרשום הבדיקות וביצוען (למשל JUnit)
- החיסרונו המרכזי נובע מההשקעה המרובה בהגדרת הבדיקות (כמעט כל שורת קוד מחיבבת שורת בדיקות מקבילה)

יתרונות

- קיימות בדיקות לכל שורות הקוד במערכת באופן מובנה – CISCO מלא.
- אם אין לך להגיד את הבדיקה – אין לך להגיד את פעולה המערכת.
- באופן מובנה קיימות בדיקות רגסיה – ככלומר כאשר מבצעים שינויים במערכת ניתן בקלות להריץ שוב את כל הבדיקות הקודמות ולודא שלא ניגרם כל נזק לפעולת המערכת.
- מסיע בדיבאג – ניתן לזרות את שורות הקוד שגרמו לביצוע לא תקין של המערכת וגם לודא שכאשר מבוצע תיקון – הבדיקות עוברות בהצלחה.

בדיקות וריפיקציה – נתונים למערכת קלטים שהמערכת לא יודעת להתמודד איתה כדי לראות איך המערכת מגיבה (למשל בדיקות מקרי קצה) ובמקרה והמערכת קורסת אז לתקן את זה בזירת הودעת שגיאה, אנו לא רוצים שהמערכת תקרוס.

בדיקות ולידציה – בדיקות הפונקציונליות של המערכת, בדיקות בהם בודקים שהפונקציונליות של המערכת עובדת בצורה שהיא נדרשת.

Inspection Tests – בדיקות פיקוח, בדיקות שבהם בודקים את המערכת לא כשהיא עובדת אלא כשפתחים אותה, למשל Design review או Code review, זהה לא בדיקה קונקרטית של שורת קוד ספציפית. זהה לא בדיקה בהם מתחשים או מוצאים באגים!

- כל בדיקה שנרשום תילך עם התוכנה לאורך כל הדרך בה המערכת קיימת (למשל גם בגרסאות מתקדמות יותר), למה? לפעמים הבאג הספציפי שהבדיקה בודקת חוזרת על עצמה כאשר מרחיבים את התוכנה لكن נרצה שכבר יהיה לנו את התסריט בדיקה זהה.
- **כל המערכת חשובה יותר** כך רמת הבדיקות תהיה ברמת אמינות גבוהה יותר.

- נרצה שהבדיקות יהיו יעילות מבחינות זמן, כלומר לא להריץ בדיקה על 1000 קלטים, אלא להריץ את הבדיקה עם קלטים מהקיצון או למשל חלוקה באפס, כך בעצם מקרים את זמן הבדיקות.

ישנים מספר סוגי בדיקות:

- **Unit Test** – לבדוק את היחידה הקטנה ביותר שאנו יכול לבדוק במערכת, **כלומר לבדוק את הפונקציות של המערכת, מתבצע לפני Acceptance Testing.**
- **זיהוי** – ספרייה בה משתמש כדי לבצע בדיקות אוטומטיות, כדי לבדוק איזושהי פונקציה נוצרת לצורך לפחות מופע אחד של אותה המחלקה.
- **בקוד הבדיקה נשתמש ב- Assertion.** כדי שהטסט ירוץ כטט נוצרת לרשום בתחילת כל בדיקה את ה- **Annotation** הבא:

 - מגדיר שהקוד הבא ירוץ רק בבדיקה, צריך לקרוא לספרייה של **@Test** כדי להשתמש בה.
 - בדיקות שניינו איזשהו ערך בתוך המחלקה עצמה יכולות להשפיע על הבדיקות הבאות ולכן ניתן להגדיר איזשהו קטע בדיקה שרצ לאחר ה-Test כדי שלא ישפיע לנו על הבדיקה הבאה, למשל לפחות את הערכים של המשתנים במחלקה, או ליצור מחלקה חדשה כדי ליצור דף חדש. **קורה לאחר כל בדיקה.**
 - ניתן להגדיר גם בדיקות שרצות לפני הבדיקה שרצה תחת ה- **Annotation של Test @, קורה לפני כל בדיקה.**
 - ניתן להגדיר איזשהו קטע בדיקה שרצ פעם אחת לאחר שכל הבדיקות הסתיימו.
 - ניתן להגדיר איזשהו קטע בדיקה שרצ פעם אחת לפני **@BeforeClass** שהבדיקות התחילהו.

לכל ה- Annotations האלו נקרא SETUP Methods ואנו משתמשים בהם כדי שבדיקה אחת לא תשפיע על בדיקה אחרת.

- **Acceptance Test** – המטרה שלהם זה איך המשתמש קצה יבחן את המערכת, כלומר לחשב על איזה קלטים המשתמש יכנס.

- **Sub System Integration Test** – בדיקות אשר משלבות בדיקה של שני מחלקות או יותר, למשל כאשר ישנו X מחלקות שմדברות אחת עם השנייה ניתן לחשב עליהם בתור רכיב במערכת ולכן לבדוק את המחלקות הללו בתור רכיב אחד.
- **System Integration Test** – בדיקות אשר בודקות את המערכת עצמה, אם יש כמה מערכות אז נבדוק כל מערכת בנפרד.

בדיקות ירושה:

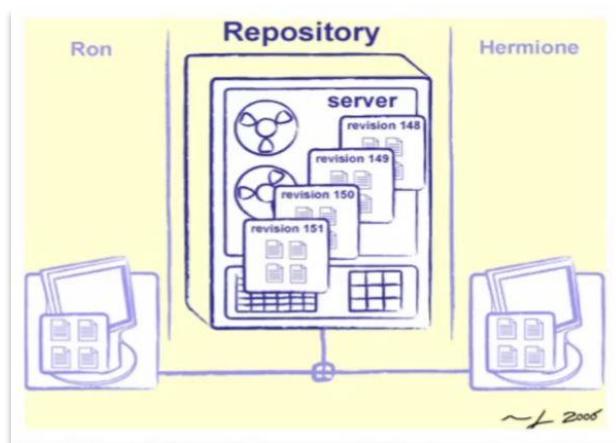
- בדיקה של מחלוקת ראשית לא מספקיה כדי לבדוק מחלוקת ירושה גם אם מדובר בשיטה של מחלוקת האב
- הסיבה היא ששיעור של מחלוקת ראשית יכולה לקרוא לשיטה של מחלוקת ירושה יכולה למשוך ולכן יכול להיות ששיעור של מחלוקת ראשית אינה נכונה במספר מקרים של מחלוקת ירושה
- יש להזכיר תשומת לב מיוחדת לבדיקות של שיטות כאשר מדובר במבנה של ירושה

הרצאה 15

:Version Control- Configuration Management

זהו השלב של התחזוקה של המערכת, ככלומר כבר השכנו גרסה רשמית ללקוח וicut נרצה לתחזק את אותן גרסאות, נציגו שאמו שומרים את כל הגרסאות של התוכנה.

כאשר אנו עובדים בקבוצה אז הרבה מאוד אנשים נוגעים בקוד ולכן נדרש למצוא שיטה שתאפשר לנו לסנכרן בין כל אותן אנשים שנוגעים בקוד כך שלא יקרה מצב שבו אנו הורסים קוד עובד.



תכונות של VCS- Version Control System:

1. הקבצים נשמרים בתוך Repository.
2. ה- Repository יכול להיות המקומי (על המחשב של המשתמש) או מרוחק (בשרת מסויים).
3. המשתמש עובד על עותק של הקובץ שנקרא Working Copy.
4. השינויים מתעדכנים לתוך ה- Repository כאשר המשתמש עשה פעולה Committed.
5. משתמשים שעובדים באותה מערכת יכולים לראות את השינויים שנעשו בקובץ לאחר שהוא עודכן על ידי משתמש אחר.
6. ניתן להשתמש המערכת גם עם יותר מ- 2 משתמשים.

- בסופו של דבר יש לנו עז גרסאות כאשר הענף המרכזי של העז אשר נקרא Muster מכיל את הגרסאות הרשמיות שהחברה הוציאה.
- הסתעפות בעז הם גרסאות אשר עובדים עליהם והם אינם מוגמרות, סיבה טוביה לפטישה של הסתעפות זה כאשר מגיע לקו מובוס מבוחינה כספית וمبקש לפתח לו גרסה מיוחדת רק עבורו עם פיצרים נוספים.
- ניתן לティיג גרסאות כדי שנדע לחזור אליהם, **לרוב נרצה לティיג גרסאות רשמיות**.

ישנים שני ארכיטקטורות של VCS:

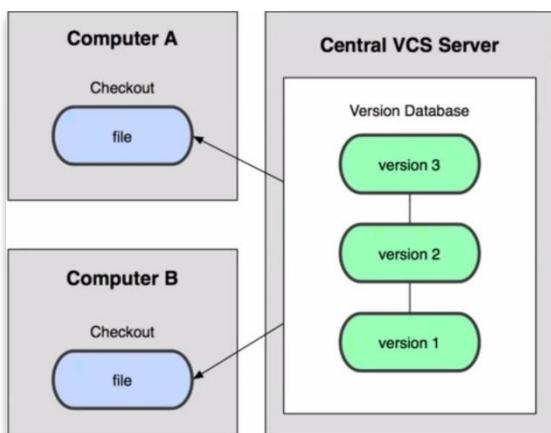
– Centralized Version Control

- יש שירות מרכזי שמותקן ברוב המקרים בתחום החברה עצמה, מתחברים לשרת וכל Client שמחובר לשרתעובד

עם פקודות Check-IN ו- Check-OUT כאשר אחד מוציא קובץ מהשרת והשני מחזיר קובץ אל השירות. בכל שלב רק מישחו אחד יכול לעבוד על גרסה מסוימת.

יתרון –

- ניהול קל של גרסאות.



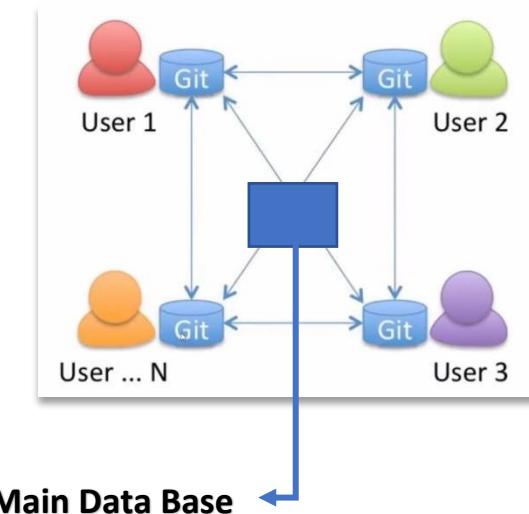
חסרונות –

- יש נקודה אחת מרכזית שמנהלת הכל ולכן יש פה **בעיה של Single point of failure**
- העבודה איטית בגלל שהכל מנוהל במקום אחד.

– **Distributed Version Control**

בשיטת זו הגרסה, כלומר ההסתעפות בעץ גרסאות נמצאת אצל כל אחד ואחד מהמשתמשים בזיכרון המקומי של המשתמש (על המחשב של המשתמש), כלומר יש גישה אחת לדאטה ביבס בו המשתמש עושה **Copy** של הקובץ המקומי **Repository** שלו והוא עובד עליו בצורה לokaלית, בסוף היום הוא יעשה **Commit** לקובץ ויחזיר אותו לדאטה ביבס הראשי.

ישנה פעולה שנקראת **Merge**, פעולה זו מבצעת איחוד בין גרסה מסוימת שנייה לקוביות עבדו עליה במקביל ושניהם עשו **Commit**.



יתרונות –

- עבודה מהירה
- **Single Point Of Failure**
- אין

חסרונות –

- פעולה ה- **Merge** היא פעולה כבדה.

נכתב על ידי אוֹהֶד אַמְסָלִם, תשפ"ב