

תכנות מתקדם 1

מבוסס על הרצאות של ד"ר מירי בן ניסן, אוניברסיטת בר-אילן 2013.

שיעור 1

תהליך קומפילציה

תהליך בו אנו משתמשים בתוכנה הנקראת מהדר, ("קומפיילר" *Compiler*) על מנת להפוך קוד מקור לשפת מכונה. מדוע חשוב שנכיר את תהליך הקומפילציה? זאת מפני שכדאי לנו להבין לעומק את ההבדל בין "זמן קומפילציה" ל-"זמן ריצה", ואת ההשלכות של השניים על תהליך העבודה והתוצר הסופי.

קדם-מעבד *Pre – processor*

זוהי תוכנית פשוטה האחראית על החלפת תבניות בקוד המקור, רגע לפני תהליך הקומפילציה. לקדם-מעבד ב-*C++* יש 4 תפקידים עיקריים: הכללת קבצים¹, הגדרת קבועים², הגדרת קטעי "מאקרו" עם פרמטרים³ וקומפילציה מותנית⁴.

במהלך הקומפילציה, הקוד עובד תהליכים שונים של "הבנת הקוד", *Parsing*, האחראים על בדיקת תקינות הכתיבה ומשמעותה. כמו כן, המהדר **עלול** להשפיע על הקוד עצמו בניסיון לשפר את הקוד: הזאת משתנים מקומיים, מיקום של פקודות, שינוי שמות של פונקציות וכד' (האופטימיזציות תלויות מהדר). מכיוון שהקוד עלול להשתנות, באגים עלולים להופיע רק בגירסה שעברה אופטימיזציה, ובמקרה זה יש לדבג גירסה זו.

מקשר *Linker*

תוכנית מחשב מורכבת בדרך כלל מרכיבים אחדים, ולכן סיבות אחדות: התוכנית משתמשת ברכיבי תוכנה סטנדרטיים. התוכנית גדולה ומורכבת ממשימות רבות, ולכן כתיבתה מחולקת בין מתכנתים אחדים, הכותבים כולם באותה שפת תכנות, או משתמשים בשפות תכנות שונות. חיבור כל הרכיבים הללו לתוכנית אחת נעשה על ידי המקשר, בתהליך הקרוי קישור (*linkage*). כאשר משתנה אחד מהרכיבים המרכיבים את התוכנית השלמה, אין צורך להדר את כל רכיביה, אלא די בהידור הרכיב שהשתנה, ובקישורו מחדש עם כל יתר הרכיבים. בתוכנות גדולות שיטה זו חוסכת זמן רב בתהליכי בניית התוכנה.

מפרש - *Interpreter*

מפרש היא תוכנה הקוראת תוכנית מחשב הכתובה בשפת תכנות ומבצעת אותה כמעט מידיית. לדוגמה, ב-*Java*, האינטרפרטר ממיר את הקוד לשפת ביניים, הנקראת *Byte – Code* אשר רצה כקלט לתוכנה אחרת הנקראת "מכונה וירטואלית" *Virtual – Machine*. מכונה זו מפרשת את הקוד, ממירה אותו לקוד מכונה מתאים ומריצה אותו. לעומת הקומפיילר, שמייצר קוד התלוי במכונה שעליו נרצה להריץ את הקוד,

¹ `#include <iostream>`

² `#define SCREEN_SIZE 80`

³ `#define SQUARE(A) ((A)*(A))`

⁴ `#ifndef ... #endif`

Byte – Code הוא גלובלי, וירוץ על כל מכונה וירטואלית. יש לשים לב כי התלות במכונה עוברת למכונה הווירטואלית, אשר תהיה חייבת להיות מותאמת למכונה.

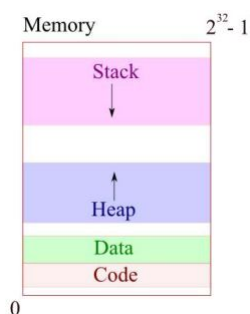
שיעור 2

משתנים

ישנם 3 סוגי משתנים שאנו מבחינים ביניהם:

1. משתנים אוטומטיים. - *Stack*
2. משתנים גלובליים + סטאטיים (המוגדרים לפני ה-*main*) - *Data*
3. משתנים דינאמיים - *Heap*

כמו כן, ישנו חלק רביעי, הקוד עצמו, שבו פרמטרים ומשתנים מקומיים מקבלים הקצאת מקום.



קובץ קונפיגורציה: על מנת להריץ את התוכנית עם פרמטרים שונים עבור ערכים מסויימים, למשל: ערך בוליאני שצריך לקבל כעת ערך *false* במקום *true*, לא נקמפל מחדש את התוכנית כי פתרון זה איננו יעיל ומבזבז משאבים. במקום זאת, ישנה אפשרות לטעון קובץ חיצוני לקריאה, המכיל ערכים ופרמטרים חדשים עבור המשתנים.

במהלך ריצת התוכנית, הקוד שאנו כותבים יושב במקום מיוחד בזיכרון. קוד מנופח ומסורבל, תופס יותר זיכרון. פונקציות *inline* לדוגמה, מנפחות את הקוד.

המחסנית - The Stack

המחסנית היא מבנה נתונים בשיטת *LIFO*, המתמלא בפרמטרי הריצה הנוכחיים לשמירתם במהלך ריצת התוכנית כשכבות ריצה שונות. קריאה לפונקציה פותחת "דף חדש" תוך שמירת הארגומנטים הנוכחיים ומצביע החזרה, כך שלמעשה מתקבלת הפרדה בין עומקי ריצה של פונקציות (ולפיכך גם הפרדה בפרטיות שלהם).

יתרונות: רציפות הזיכרון (פחות פרגמנטציה), וגודל הקובץ קטן. (הקצאות דינמיות על ה-*Heap* גדולות משמעותית מה-*Stack*)

מסגרת ריצה (גם: "פריים" Frame) שם לעומק ריצה מסויים, שבו מאוחסנים הפרמטרים השייכים לאותה סביבת ריצה.

הערימה - The Heap

למתכנת יש מקום גדול בזיכרון שבו הוא יכול לאחסן משתנים דינאמיים. במקום זה חייבים להצביע על המשתנים הללו בעזרת פוינטר, אחרת הזיכרון אובד. כאשר נשתמש באופרטור *new* בשפת ++C, אנו מקצים מקום על ה-Heap.

רמות זיכרון

ברמה הבסיסית ביותר נמצאת ה**חומרה**, *Hardware*, שהיא המהירה ביותר והקרובה ביותר למעבד. ברמה השנייה, יש את **מערכת ההפעלה** שמנהלת מערכת שלמה של זיכרון וירטואלי (Win32 APIs) ברמה השלישית, רמת ה**אפליקציה**, למתכנתים יש גישה לנהל בעצמם משתנים וזכרונות ולהגדיר מבני נתונים חדשים כרצונם. ברמה האחרונה נמצא ה**משתמש**, אשר אינו מקצה הלאה זיכרון, אלא רק מבקש.

Cache

זיכרון הנדרש בצורה תקופה, יאוחסן קרוב יותר למעבד. זמינות זו תורמת משמעותית לביצועי התוכנית, שכן הפעולות מתבצעות ברמה נמוכה של המעבד בעיקר, ולא מתבצעת קריאה חוזרת ונשנית מהזיכרון האיטי (דיסק קשיח מכני).

זיכרון וירטואלי - Swap (also: Page) file

מערכת ההפעלה ממפה את זיכרון ה-*Ram* בנוסף עם חלק מהדיסק הקשיח, על מנת להציג יותר זיכרון זמין למשתמש. מערכת ההפעלה אחראית לשיבוץ נכון של הזיכרון על מנת לשמור על רמת ביצועים סבירה⁵.

User space vs System space

הזכרון הוירטואלי מחולק לשני חלקים: מרחב המשתמש הוא איזור גלובלי שבו רצות האפליקציות של המשתמש הנוכחי (או משתמשים נוספים), ומרחב המערכת, שבה נמצאים ומאוחסנים משאבי מערכת ההפעלה.

⁵ שכן זיכרון הדיסק איטי משמעותית מזיכרון ה-*Ram*

Managed and unmanaged Systems

היכן נמצא כוח ניהול הזכרון? ב-Java, ניהול הזכרון הוא אוטומטי בעוד שב-C++\C ניהול הזכרון ע"י המשתמש. זהו כלי שנותן כוח למתכנת אך גם אחריות גדולה לניהול נכון של הזכרון. נשים לב כי אנו עוסקים בניהול זיכרון על ה-Heap!

שיעור 3

אופטימיזציה של זכרון

Char יכול להתחיל בכל מקום, מכיוון שגודלו הוא בית 1. נאחסן *Short* במקומות זוגיים. כל שאר המשתנים יכולים להכנס בכתובת שמתחלקת ב-4 או בגודל מילה. ברגע שסגרנו *Word* באמצעות ה-*Struct* שלנו, מערכת ההפעלה "תרפד" את שאר הבתים, כך שה-*Word* ים שלו יהיו מוגנים. **כלל אצבע לאופטימיזציה:** נתחיל עם הקצאת מקום למשתנים הגדולים ביותר, ובסדר יורד עד למשתנים הקטנים ביותר.

ניהול זיכרון הערימה Heap Management

שיעור 4

שיעור 5

קונסטרוקטורים ודיסטרוקטורים

קונסטרוקטורים משמשים ליצירת מופעים של אובייקט מחלקה. שמם של הקונסטרוקטורים צריך להיות זהה לשם של המחלקה. לכן, הדרך היחידה להבחין בין הקונסטרוקטורים השונים היא רשימת הפרמטרים שהם מקבלים.

Named Constructor Idiom

מוטיבציה: ב-C++ יש לקונסטרוקטורים משמעות שונה עבור רשימת פרמטרים שונה. אבל השם שלהם זהה. דבר זה עלול ליצור בלבול וחוסר בהירות בקוד. הפתרון המוצע הוא כדלהלן:

בעזרת סט של פונקציות סטטיות בעלות שם משמעותי, ניתן ליצור מעין מעטפת האחראית ליצירת האובייקטים. את הפונקציות הללו נשים בחלק ה-*public*, כאשר את שאר הקונסטרוקטורים נשים ב-*protected* או *private*.

דוגמה:

```

class Game
{
    public:
        static Game createSinglePlayerGame() {return Game(0);} //named constructor
        static Game createMultiPlayerGame() {return Game(1);} //named constructor
    protected:
        Game (int game_type);
};
int main(void)
{
    Game g1 = Game::createSinglePlayerGame(); //Using named constructor
    Game g2 = Game(1); //multiplayer game;without named constructor (does not
        compile)
}

```

בדוגמה לעיל, ללא שימוש ברעיון זה, קשה לנו להבין את משמעות הקונסטרוקטורים Game(0) ו-Game(1). רעיון זה הופך את התהליך לברור כשמשי!

הביצועים של קונסטרוקטורים ודיסטרוקטורים נמוכים בד"כ, עקב כך שהקונסטרוקטור או הדיסטרוקטור של אובייקט עלול לקרוא לקונסטרוקטורים או דיסטרוקטורים של אובייקטים פנימיים לאובייקט ואובייקט-על (Parent Objects). כל עוד הקריאות הללו הכרחיות, אין דרך לעקוף זאת, והמתכנת צריך להיות מודע להתבצעותן של קריאות שקטות אלו.

בשפת C, צריך להכריז על המשתנים בראש הפונקציה, מה שלא כן ב-C++. האחרונה נותנת, בדומה לשפת Java, אפשרות להצהיר על משתנים מתי שרק רוצים. ההגבלה היחידה היא, שצריכים להצהיר על משתנה לפני שמשתמשים בו. על מנת לשמור על עקרונות תכנות נכונים, נדאג להצהיר על משתנים בטווח (Scope) הקצר ביותר הניתן, ומיידית לפני השימוש במשתנים.

יצירת מערכים

כאשר ניצור מערך של אובייקטים, איזה קונסטרוקטור ייקרא? במקרה בו המערך הוא "ווקטור רגיל" קונסטרוקטור ברירת המחדל (הקונסטרוקטור הדיפולטי) ייקרט על מנת לבנות את האובייקטים. במקרה בו מדובר ב-"ווקטור ספרייה" (STL vector: Standard library vector) הקונסטרוטור הדיפולטי ייקרא פעם אחת, על מנת ליצור באופן זמני תבנית, ותבנית זו תועתק (בעזרת ה-Copy constructor) לכל התאים במערך.

שאלה: נניח והגדרנו $\text{std::vector}\langle T \rangle$, האם ל- T חייב להיות קונסטרוקטור דיפולטי?
תשובה: לא, ואנו צריכים רק אופרטור השמה, ('='), או assignment operator) או Copy constructor.

שאלה: האם כדאע להשתמש ברשימות איתחול⁶ או באופרטור השמה⁷?
תשובה: רשימות איתחול, כאשר היתרון הנפוץ ביותר הוא שיפור בביצועים. למעשה, למעט מקרה קצה שנרחיב בהמשך עליו, קונסטרוקטורים צריכים ככלל לאתחל את כל האובייקטים שלהם ברשימת האיתחול. במקרה בו כל האובייקטים הם פרימיטיביים, אין שום הבדל. כאשר חלק מהאיברים הם אובייקטים בעצמם, במקום להשתמש

⁶ לדוג': MyClass::MyClass():_capacity(15),_data(NULL),_len(0){}

⁷ MyClass::MyClass() _capacity=15;_data=NULL;_len=0;

באופן היצירה הרגיל, רשימת האיתחול מאפשרת ליצור אובייקטים ישירות לתצורה הסופית שלהם ולקבל את הערכים הסופיים. הבעיה ביצירה בעזרת ההשמה, היא שאובייקטים זמניים נוצרים בזמן ריצה ונהרסים ע"י דיסטורקטורים כאשר מגיעים ל-;.

שיעור 6

הנדסת תוכנה - עקרונות בסיסיים

הנדסת תוכנה מיישמת גישה שיטתית, מבוקרת וניתנת למדידה, לצרכי פיתוח, תפעול ותחזורה של תוכנה. הנדסת תוכנה מקיפה את מחזור החיים השלם של תוכנה, וכוללת ידע, שיטות וכלים עבור דרישות תוכנה, תכנון תוכנה, בניית תוכנה, בדיקות תוכנה, תחזוקת תוכנה, ניהול תצורת תוכנה ואיכות תוכנה. הנדסת תוכנה נועדה להפחית את המורכבות שבפיתוח תוכנה, לשפר את אמינות התוכנה המפותחת, ולהקטין את עלויות התפעול והתחזוקה. מאפיין בולט של הנדסת התוכנה הוא פיתוח מערכות מורכבות הכוללות חומרה, תוכנה ותקשורת.

ממה מורכבת הנדסת תוכנה?

1. תהליך פתירת בעיות שמציג הלקוח:

(א) זוהי מטרת העל של הנדסת תוכנה. לעיתים, פתרון לבעיה מסויימת יכול להתבטא ברכש מוצר. תוכנה חיצוני והימנעות מפיתוחו. תקשורת פעילה וטובה מסייעת לזהות ולהבין את הבעיות שעולות.

2. פיתוח והרחבה שיטתיים של מערכת:

(א) תהליך הנדסי המשלב יישום טכניקות המובנות היטב, ובנויות בצורה מסודרת ומוגדרת מראש. דבר זה יוצר סטנדרטים מקובלים שאפשר לעבוד לפיהם.
דוגמה לכך היא IEEE - Institute of Electrical and Electronics Engineers או ISO - International Organization for Standardization.

3. מערכת גדולה באיכות טובה:

(א) נדרשות טכניקות של הנדסת תוכנה כי מערכות גדולות לא יכולות להיות מובנות לגמרי על ידי אדם אחד. נדרשים עבודת צוות ויחסי גומלין. חלוקת העבודה ווידוא שכל חלק במערכת עובד ביחד הוא חלק קריטי בהנדסת התוכנה. המוצר הסופי צריך להיות באיכות מספקת.

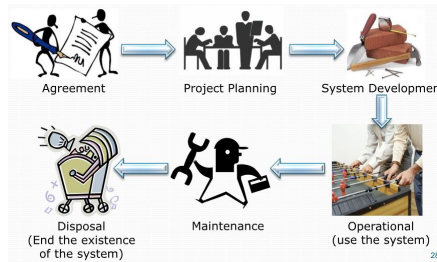
4. התחשבות באילוצי מחיר, זמן ואילוצים אחרים:

(א) המשאבים הם סופיים. רצוי שהרווח יהיה גדול מההשקעה. לזכור שהמתחרים מנסים לבצע את העבודה בצורה זולה ומהירה יותר. הערכה לא מדויקת של המחיר והזמנים גרמו להרבה פרויקטים להיכשל.

קשיים ומגבלות בהנדסת תוכנה

- **סיבוכיות** - מספר תרחישים גדול מאד אשר מקשה להבין את המכלול.
- **תאימות** - תוכנה נדרשת להתאים את עצמה למגבלות של החומרה ושל התוכנות האחרות בסביבתה.

- **דרישות לא מובנות / מוגדרות היטב** שקיים פער בתפיסתן ע"י בעלי התפקיד השונים.
- ניסיון **להמציא את הגלגל** במקום להשתמש ביישום זמין.
- **הערכת זמן** שגוייה לביצוע המטלות.
- **ניהול שינויים** והתמודדות עם זמינות המשאבים.



תכנון הפרוייקט - שלבים

תכנון הפרוייקט - שלב מקדים

1. קביעת הלו"ז
2. קביעת התקציב
3. קביעת הכלים והשיטות
4. זיהוי מרכיבי המערכת
5. זיהוי התוצרים
6. הגדרת אבני הדרך

פיתוח התוכנה

1. הגדרת הדרישות: הבנה, תיעוד ותעדוף הדרישות של המערכת המבוקשת. יש להיזהר בשלב זה לחלץ מהלקוח מה הבעיה, מבלי להיגרר לפתרון. מיומנות וניסיון עוזרים לזהות דרישות לא שלמות או סותרות.
2. ניתוח דרישות: ניתוח עלות מול תועלת, עלויות רכישה, עלויות התנעה, עלויות פיתוח הפרוייקט ועלויות תחזוקה שוטפת – מול תועלת לפרוייקטים אחרים, כניסה לשוק וכד'. האם הטכנולוגיה הדרושה קיימת, ניתוח סיכונים, בעיות חוקיות וביטחוניות. שקילת אלטרנטיבות.
3. שלב פירוט המערכת - Specification: בשלב זה מתארים בצורה מפורטת את התוכנה שרוצים לכתוב בצורה מוקפדת. בשלב זה מגדירים את הממשקים החיצוניים, וזה קריטי לפיתוח שהם יישארו יציבים. מחלקים את המערכת ל - ICSC - ים, רכיבי חומרה וממשקים. מגדירים מהם גבולות המערכת מבחינת משאבים וזמן, מה הקדימות שנתן הארגון לפרוייקט וכד'.
4. שלב הגדרת ארכיטקטורת התוכנה: הבנת קונפיגורצית המערכת והאילוצים הנובעים ממנה, מחשבים, ממשקים ותקשורות. הגדרת קשר לתוכנות נוספות, הבנת ההתנהגות הדינאמית של המערכת. המטרה היא ליצור ארכיטקטורת תוכנה יציבה. הארכיטקטורה כוללת: ממשקים, טכנולוגיות, טופולוגית המחשוב, מערכת הפעלה, שכבות, מנגנונים משותפים, קומפוננטות התוכנה.

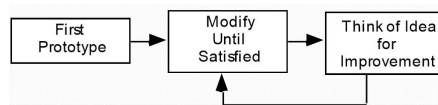
5. שלב תיכון המערכת: פירוט כל האלמנטים למימוש, עד רמת יחידות תוכנה. פירוט בדיקות: שלב המעבר בין המפרט ("מה?") למימוש ("איך?"). תכנון אופן המימוש של כל דרישות המפרט, עיצוב מבנה הקוד (ארכיטקטורה "פיזית" של התוכנה):

- (א) מילוי דרישות שלא באו לידי ביטוי במודל הניתוח.
- (ב) דרישות ביצועים.
- (ג) עיצוב ממשק המשתמש (דרישות הנדסת - אנוש) דרישות / הנחיות טכנולוגיות
- (ד) שלב המימוש: מעבר מהתיכון לקוד. מימוש וביצוע בדיקות יחידה. עבודה על פי נוהלי קידוד מוסכמים ואחידים. תיעוד.
- (ה) שלב האינטגרציה והבדיקות: שילוב מודולי התוכנה ווידוא שהתוכנה עונה על הדרישות והיא בשלה לקראת שילובה במערכת. שילוב התוכנה עם רכיבי החומרה, WF ותוכנות נוספות. ביצוע בחינות קבלה.
- (ו) שלב הדרכה ותמיכה: הדרכה של הלקוח כיצד להשתמש ביכולות המערכת. הדרכה נכונה תוריד את כמות התמיכה הנדרשת. הדרכה לא נכונה יכולה לגרום לחוסר שימוש במערכת על ידי הלקוח.
- (ז) שלב התחזוקה: מתבצע בהתאם להגדרות החוזה. תיקון באגים. הרחבות למערכת לפי בקשת הלקוח.

מתודולוגיית פיתוח תוכנה

מתודולוגיית פיתוח תוכנה היא סט מוסכם של עקרונות, תהליכים, פעילויות וכלים על פיהם מפותחות ומתוחזקות מערכות תוכנה. המתודולוגיות הללו מתחלקות לכמה משפחות עיקריות:

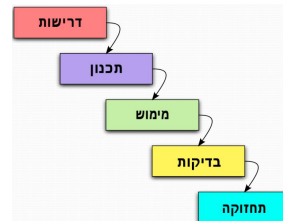
1. מתודולוגיות קוויות:



(א) תכנון ותקן - Build & Fix :

- i. יתרונות: כמעט ואין תקורה של התהליך, "מסתערים" על הפרויקט. מתאים לפרויקטים קטנים וממוקדים. מתודולוגיה זו מאפשרת הוספה מהירה של פונקציונאליות חדשה למערכת ולכן מתאימה לשלב התחזוקה.
- ii. חסרונות: אין התייחסות לחשיבות של הבנת הדרישות ולביצוע תוכנו לפני מימוש מערכת. אין דרך למדוד התקדמות, איכות או סיכונים. לאחר כל שינוי יש חובה של חיזוק המבנים הפנימיים של התוכנה מעת לעת וערכת בדיקות מקיפה, אחרת נגיע לשלב בו עלות הוספת פונקציונאליות חדשה שווה או גדולה לעלות בניית התוכנה כולה מחדש, שלאחריה אין הצדקה כלכלית להמשיך לתחזק את התוכנה.

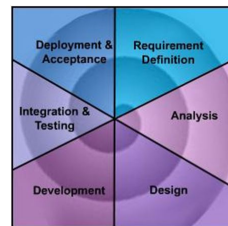
(ב) מפל המים:



- i. פיתוח התוכנה מתבצע בתהליך שיטתי ולוגי המורכב משלבים מוגדרים-היטב שאין לפסוח עליהם. השלבים מבוצעים בטור, אחד אחרי השני, ובכל שלב יש מיקוד במשימה עיקרית אחת בלבד. מתודולוגיית מפל המים שמה דגש רב על איסוף וניתוח של כל הדרישות כולן קודם לתחילת הפיתוח, וממליצה שתהליך הפיתוח לא יחזור לאחור לאחר ששלב מסוים בו הסתיים. השלבים העיקריים בשיטה זו הם: איסוף וניתוח דרישות, עיצוב תוכנה, תכנות, בדיקות, שילוב, התקנה ותחזוקה.
- ii. יתרונות: מתאים לצוות לא מנוסה. מודל סדרתי שקל לעקוב אחריו, בכל שלב אפשר לבחון אם להתקדם הלאה.
- iii. חסרונות: בד"כ אי-אפשר לחשוב על הכל מראש. התארכות התהליך, בשל הדגש על שלב הניתוח והעיצוב דבר שמגדיל גם את עלות הפרויקט. היווצרות נתק בין המפתחים למשתמשים, היות ששלב הפיתוח הוא ארוך חולף זמן רב עד שהמשתמשים מקבלים או רואים את המערכת החדשה. המוצר יוצא קשיח בהעדר משוב במהלך הדרך. ייתכן שבסוף התהליך יצא מוצר שהלקוח אינו מעוניין בו. חוסר נצילות בצוות: לא כולם עובדים בכל שלב. חוסר גמישות \ פתיחות לשינויים.

(ג) מודל אב - טיפוס מהיר, זהה למפל המים, תוך שימת דגש על רכיבים עיקריים בלבד על מנת לגבש אב - טיפוס ועל פיו להעריך את הדרישות, הזמנים, המשאבים וכו' הנדרשים למערכת השלמה. לרוב יופיע כשלב לפני מודל אחר. בד"כ לא יכלול רכיבים כמו כתיבה לקבצים, טיפול בשגיאות וכד'.

2. מתודולוגיות איטראטיביות:



(א) מודל הספירלה:

- i. מתחילים מפיתוח מערכת מינימאלית, ומבצעים את כל השלבים. לאחר סיום מעריכים את המוצר הנוכחי, מחליטים מה להוסיף ומה הסיכונים, וחוזרים על כל השלבים. מודל זה הוסיף למפל המים את המימד האיטראטיבי \ אבולוציוני. המתודולוגיה מספקת משוב איכותי הכולל הערכת לקוח, הגדרת יעדים וניתוח סיכונים, מגדירה מדיניות ולא תהליך בלבד, וניתן ליישם אותה לגבי המתודולוגיות האחרות.
- ii. יתרונות: לא צריכים לחכות עד להבנה של כל המערכת. בכל שלב המטרות מתבררות יותר. בכל איטרציה יש חלק מערכת בדוק ועובד. מפחית חרדות: האם אנו בונים את המוצר הנכון? האם יש לקוחות למוצר? האם ניתן לממש את המוצר בעזרת הטכנולוגיה הקיימת היום? מחר? השקעה כספית מדורגת - החזר השקעה מהיר. התקדמות זהירה יותר אל התוצאה דרך אבות טיפוס (כבר פחות מקובל).
- iii. חסרונות: סכנת אבדן שליטה. קשה לניהול - הרבה ניהול ותכנון, מצריך מומחיות בניהול סיכונים. מתאים למפתחים מנוסים, שיועדים איך להתקדם בו ולא מנסים בבת - אחת לפתח הכל. קשה להתאים מחוייבויות חוזיות למודל. בעוד שבמודל מפל המים אפשר להגיד אחרי כמה שלבים כמה אחוזים משלמים (כי חוזרים אחורה רק במקרה של בעיות ודרישות חדשות), פה לא יודעים מראש כמה איטרציות יהיו.

3. מתודולוגיות זריזות (Agile)

(א) זוהי מתודולוגיה שהותאמה לפיתוח תוכנה בצוותים קטנים תוך שימת דגש על יעילות, זריזות ואיכות. (גישת הסטראט-אפ). גישה זו לפיתוח תוכנה מניחה שלא ניתן להגדיר במלואה תוכנה מסוימת קודם לפיתוחה בפועל, ומתמקדת במקום זאת בשיפור יכולתו של הצוות לספק תוצרים

במהירות ולהגיב לדרישות העולות תוך כדי הפיתוח. בראש סדר העדיפויות הוא לספק מוקדם ובאופן רציף ככל האפשר תוכנה בעלת ערך ללקוח. מצפים בברכה לשינויים בדרישות התוכנה, אפילו בשלב מתקדם של הפיתוח. תהליכים מהירים ברי-שינוי מאפשרים יתרון משמעותי בתחרות הלקוח. אספקה של תוכנה עובדת באופן תדיר, ממספר שבועות עד למספר חודשים, עם שאיפה לסולם זמנים קצר. אנשי פיתוח ואנשי העסקים (המנהלים) חייבים לעבוד ביחד באופן יומי לאורך הפרויקט. תחזוקה של רשימת פריטי העבודה לביצוע, מסודרים לפי קדימויות. השלמת מנה קבועה של פריטי עבודה בסדרה של איטרציות קצרות המכונות 'מאוצים' (*Sprint*). משך כל מאוץ הוא 4 שבועות, ובסיומו מסופקת תוכנה עובדת למשתמשים. פגישת צוות יומית קצרה המכונה 'Daily Scrum'. בפגישה מציג כל אחד מחברי הצוות את ההתקדמות, העבודה המתוכננת וקשיים אפשריים. הפגישה מתקיימת לרוב בעמידה. פגישת תכנון מאוץ (*Sprint Planning*) קצרה שבה מוגדרים פריטי העבודה לאותו מאוץ. פגישת ניתוח מאוץ (*Sprint Retrospective*) קצרה להפקת לקחים מהמאורך הקודם. השיטה מעודדת ריכוז של כל חברי הצוות במיקום אחד.

XP-Extreme Programming (ג)

i. המתודולוגיה, מפרטת שורה של טכניקות בתחום התכנות ופחות בתחומים אחרים של הנדסת תוכנה. מערכות המפותחות לפיה גמישות מאוד לשינויים, וניתן להרחיבן בקלות ובאופן בטוח. השיטות המשמשות אותה הן מחמירות מאוד. משתמשת בשיטת פיתוח מונחה - בדיקות שעיקריה הם כתיבת דרישות המערכת כסט של בדיקות הניתנות להרצה, ופיתוח הבדיקות קודם לפיתוח הפונקציונאליות. שיטה זו דורשת הבנה טובה של עקרונות תכנות מונחה - עצמים ומשמעת עצמית גבוהה. מעצבי השיטה ניסחו 5 ערכים: משוב, פשטות, תקשורת, אומץ, כבוד. כל אחד מערכים אלו נלקח לקיצוניות. אינטגרציה מתמשכת - יש לבצע אינטגרציה של המערכת, הכוללת הידור וקישור לפחות פעם ביום. תכנות בזוגות - תמיד יושבים מול מחשב אחד זוג מתכנתים. אחד מחזיק במקלדת ובעכבר ומכונה 'הנהג'. הוא מבצע בפועל את עבודת התכנות. השני, המכונה 'הנווט', עוקב אחר הנהג, מיעץ לו לגבי דרך העבודה, ובעיקר אחראי על איתור תקלות בקוד: (תחביריות, לוגיות וכו'). את הזוגות מחלפים מדי פעם, בשביל שכל המתכנתים יכירו כמה שיותר קוד, ואילו במבט על קוד חדש יהיה מי שיחנך את התוכניתן שאינו מכיר את הקוד. מזניחים את שלב הדרישות והתיעוד. התנהלות צוותית - הצוות כולו מאוחד במטרה ומסונכרן כל הזמן. בתחילת כל יום מתבצעת 'פגישת עמידה' באורך של כ-15 דקות, בה כל הצוות נוכח, בעמידה, ומספר על התקדמותו ביום הקודם ותוכניותיו ליום הנוכחי. "משחק התכנון" - המתכנתים עובדים בשיתוף פעולה מלא עם הלקוח בתכנון המערכת. הלקוח כותב סיפורים והמתכנתים מנתחים אותם ומסדרים אותם על - פי סדר עדיפויות. לאחר מכן, הסיפורים הופכים למשימות פיתוח. גרסאות קטנות - שחרור של גרסאות קטנות ללקוח. PX תומכת בשחרור גרסאות כל 2-3 חודשים. הסיבות הן: קבלת משוב מהיר, תחושת השגיות של הצוות, הפחתת סיכונים, הגברת ביטחון הלקוח בצוות הפיתוח והתאמת התוכנה לדרישות.

Crystal Clear (ג)

Scrum (ד)

4. היתרונות בשימוש במתודולוגיות פיתוח הכוללות את מחזור החיים של הפרויקט: מסגרת לעבודה על המוצר. מכריח אותנו לחשוב על " התמונה הגדולה " ומאפשר להגיע אליה צעד אחר צעד. אחרת, החלטות שיילקחו אולי נכונות מקומית ואישית, אך מפספסות את המטרה הכללית. כלי ניהול, הבטחת איכות.

שיעור 7

Programming Frameworks

סוגי הגישות השונים לתכנון ותכנות המוצר, תלוייה במידה רבה באופי המוצר. ישנם מספר סוגים עיקריים של יישומי תכנות: כלים קצרי טווח, אפליקציות משרדיות, שרתים, אפליקציות רשת (*web*) ומערכות זמן אמת / מוטמעות (בחומרה).

שיעור 8

עקרונות פיתוח תוכנה

הקדמה

ישנן הרבה הערכות הקשורות לתכנות מונחה עצמים כגון: "כל משתני המחלקה צריכים להיות מוגדרים כ-*private*" או "צריך להימנע ממשתנים גלובליים" או "שימוש בזיהוי סוג המחלקה בזמן ריצה הוא מסוכן". מהו המקור להערכות אלו? מה הופך את הערכות אלו לנכונות? האם הם נכונות תמיד?

ישנן עקרונות כללים בפיתוח תוכנה:

1. Open - Closed principle

(א) פתוח להרחבה, אך סגור לשינויים. הרעיון הוא שינוי הפעולה של המודול⁸ בלי צורך לשנות את קוד המקור של המודול. הרחבה: התנהגות המודול ניתנת לשינוי לפי דרישות חדשות שעולות. סגור לשינוי: קוד מקור של מודול כזה אסור לשנותו והוא בחזקת קופסה שחורה.

(ב) כיצד מגיעים לכזה תכנון שכביכול מנוגד אחד לשני? התשובה היא בעזרת **אבסטרקציה** ובעזרת **פולימורפיזם** (שינוי בהורשה).

(ג) יתרונות: התוכנית גמישה וחסינה לשינויים מכיוון שאין צורך לשנות קוד כתוב. כמו כן יותר קל לבדוק אותה והיא נוטה להכיל פחות שגיאות.

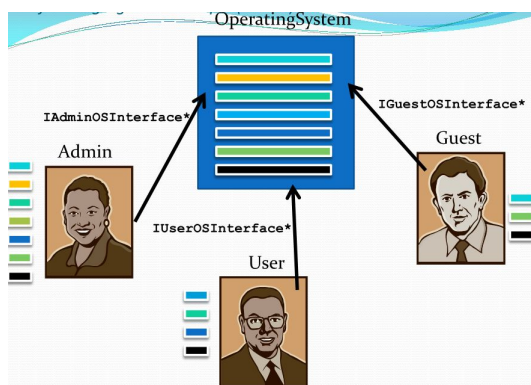
2. Single responsibility principle

(א) לכל אובייקט צריכה להיות אחריות יחידה אשר שייכת וממומשת במחלקה (*encapsulation*). תכנון כזה נותן לנו גמישות רבה יותר בתכנון, שכן נצטרך לשנות חלק קטן יותר מהקוד (רק את האובייקטים הרלוונטים לשינוי).

3. Interface segregation principle

(א) שימוש מבוזר בממשקים: העדפת ממשקים קטנים ומרובים המרכיבים את הממשק הגדול על פני שימוש בממשק גדול ואחיד הגורם ללקוח לממש פעולות שאין לו צורך בהם.

⁸ חלק בתוכנה. יכול להיות מחלקה, או אפילו חבילת מחלקות שלמה.



לדוגמה:

```

class IAdminOSInterface
{
    virtual operation1()=0;
    virtual operation2()=0;
    virtual operation3()=0;
    virtual operation4()=0;
    virtual operation5()=0;
    virtual operation6()=0;
    virtual operation7()=0;
};
  
```

```

class IUserOSInterface
{
    virtual operation2()=0;
    virtual operation3()=0;
    virtual operation5()=0;
    virtual operation7()=0;
};
  
```

```

class IGuestOSInterface
{
    virtual operation1()=0;
    virtual operation6()=0;
    virtual operation7()=0;
};
  
```

כאשר מערכת ההפעלה:

```

Class COperatingSystem :
public IAdminOSInterface ,
public IUserOSInterface ,
public IGuestOSInterface
{
public:
    virtual operation1();
    virtual operation2();
    virtual operation3();
    virtual operation4();
    virtual operation5();
    virtual operation6();
    virtual operation7();
};
  
```

1. Liskov substitution principle

(א) אחד מהעקרונות היותר מוכרים שברברה ליסקוב ניסחה, מוכר כ"עקרון ההחלפה". הרעיון הוא שאם מחלקה D יורשת ממחלקה B, אז עקרון ההחלפה אומר שבכל מקום בקוד בו מתייחסים ל-B (מחלקת האב), ניתן יהיה להחליף את B בתוכנית ב-D (או בכל צאצא אחר של B) - מבלי שיהיה צורך לשנות או לעדכן את הקוד. כלומר: קליינט המשתמש ב-pointer או ב-reference למחלקת בסיס B, לא צריך לדעת לאיזה תת-מחלקה האובייקט שייך. ההתנהגות של הבנים צריכה להיות דומה לשל האבא.

(א) עקרון זה אומר כי מחלקות גבוהות בהיררכיה לא תהיינה תלויות במחלקות נמוכות יותר, אלא שתיהן תסתמכנה על האבסטרקציה. כלומר, בהינתן מחלקות גבוהות בהיררכיה המצריכות שימוש במחלקות נמוכות יותר, נעדיף שימוש בשכבת אבסטרקציה החוצצת ביניהם. כך נפריד בין המימושים השונים ולא נצטרך לחזור על הבדיקות עבור המחלקה הגבוהה.

שיעור 9

איסוף "זבל" בג'אווה Garbage Collection in JAVA

נזכיר בנקודה זו כי איסוף זבל, GC, היא צורה אוטומטית של ניהול זיכרון. "זבל" במקרה זה, הוא זיכרון שאין אפשרות להגיע אליו מהתוכנית, ולכן הוא איננו שמיש על ידה. ישנן 2 גישות לשיטה לאיסוף.

1. **ספירת הקשרים:** שמירת מספר ההקשרים לאובייקט A מסויים. כאשר מספר זה הוא 0, אין יותר מצביעים לשם ונוכל לפנות אותו מכיוון שאיננו נגיש יותר.

2. **מעקב:** בדיקה אחורנית בעזרת גרף של כל האובייקטים, החל מהבסיס וסימונו. כאשר נסיים, כל אובייקט שלא סימנו יחשב כזבל.

המכונה הווירטואלית של ג'אווה, ה-JVM, אחראית על איתחול הערימה Heap, כאשר גודלה של הערימה גדל ע"פ ערך המינימום שלה, ערך המקסימום שלה, והשימוש הנוכחי. הערימה מתרחבת כאשר עוברים סף מסויים המוגדר מראש של אחוז שימוש. בג'אווה אין משתנים לוקאליים, וכולם מוגדרים על הערימה בעזרת האופרטור new(). בג'אווה אין שימוש בדיסטריקטור, ועל ה-GC מוטלת המשימה של שחרור הזיכרון.

לעיתים, המחלקה עלולה לחלק משאבים שה-GC לא ידע לשחרר, ועל מנת לטפל בכך ג'אווה מספקת מתודה הנקראת finalize(), שניתן להגדיר למחלקה שאנו בונים. כאשר ה-GC מוכן לנקות מחלקה מסויימת, הוא קודם יקרא למתודה זו, ורק אז הוא ימשיך בשחרור המשאבים. כלומר, שימוש במתודה זו מאפשר ביצוע ניקיון אחרון בזמן איסוף הזבל. עם זאת, שימוש במתודה זו הוא לא-דטרמיניסטי ועלול ליצור בעיות, שכן אנו לא יודעים מתי ה-GC יקרא למתודה, אם בכלל.

מכיוון שמנגנון איסוף הזבל הוא אוטומטי, לא ניתן לדעת אם וכאשר הוא ייקרא. ניתן להכריח ניקיון ע"י קריאה ל-System.gc().

ישנם 3 שלבים בתהליך איסוף הזבל: **החלטה** שאובייקט מסויים לא נגיש. **בדיקה** אם לאובייקט יש מתודה finalize(), ואם כן, האובייקט מתווסף לרשימת המתנה לשחרור. רק לאחר ביצוע המתודה עוברים לשלב הבא. השלב האחרון הוא **החזרת הזיכרון** שהאובייקט השתמש למאגר הפנוי.

ישנן 3 גישות לאופן איסוף הזבל: **עצור את העולם אני רוצה לרדת**, בו התוכנית איננה מורשית לרוץ עד שתסתיים פעולתו של ה-GC, **מדרג עולה**, בו מבצעים קצת מרשימת הפעולות של איסוף הזבל, עוצרים וחוזרים לתוכנית, עוצרים וחוזרים לאוסף הזבל - וחוזר חלילה עד סיום הפעולה של ה-GC (בדומה ל-Pipelining). כמו כן ישנה גישה **מקבילית** בה איסוף הזבל רץ במסלול נפרד¹⁰ יחד עם האפליקציה.

¹⁰ רץ כ-Deamon Thread - בשונה מעט מת'רדים "רגילים" של ג'אווה.

אזהרה: כאשר אין הקשר חזק, אנו עלולים לקבל חזרה *Null* מבקשות *get()* על אובייקטים, שכן מנגנון ה-*GC* מתייחס אל אובייקטים בעלי קישור חלש כבלתי מושגים *unreachable*. ניתן לראות זאת בדוגמה הבאה:

```
public class ReferenceTest {
    public static void main(String[] args) throws InterruptedException {
        WeakReference r = new WeakReference(new String("I'm here"));
        WeakReference sr = new WeakReference("I'm here");
        System.out.println("before gc: r=" + r.get() + ", static=" + sr.get());
        System.gc();
        Thread.sleep(100);
        // only r.get() becomes null
        System.out.println("after gc: r=" + r.get() + ", static=" + sr.get());
    }
}
```

שיעור 10